Seamless Database Evolution for Cloud Applications

Aniket Mohapatra[1,3], Kai Herrmann[2], Hannes Voigt[2], Simon Lüders[3], Tsvetan Tsokov[4] and Wolfgang Lehner[2]

[1]*Technische Universität Kaiserslautern, Germany*

[2]*Technische Universität Dresden, Germany*

[3]*SAP SE, Walldorf, Germany*

[4]*SAP Labs Bulgaria EOOD, Sofia, Bulgaria*

Keywords: Database, Schema Evolution, Migration, Seamless Deployment.

Abstract: We present DECA (Database Evolution in Cloud Applications), a framework that facilitates fast, robust, and agile database evolution in cloud applications. Successful business requires to react to changing wishes and requirements of customers instantly. In today's realities, this often boils down to adding new features or fixing bugs in a software system. We focus on cloud application platforms that allow seamless development and deployment of applications by operating both the old and the new version in parallel for the time of development/deployment. This does not work if a common database is involved, since they cannot co-exist in multiple versions. To overcome this limitation, we apply current advances in the field of agile database evolution. DECA equips developers with an intuitive Database Evolution Language to create a new co-existing schema version for development and testing. Meanwhile, users can continuously use the old version. With the click of a button, we migrate the database to the new version and move all the users without unpredictable downtime and without the risk of corrupting our data. So, DECA speeds up the evolution of information systems to the pace of modern business.

1 INTRODUCTION

Business success largely depends on the ability to quickly adapt to the ever-changing needs and wishes of customers. In our age of digitalization, almost all businesses are permeated by IT systems and the pace of evolving the software system determines the pace of evolving and adapting the business. The software technology community applies agile development methods for easy and robust evolution of applications. Modern cloud application platforms facilitate the seamless testing and deployment of new versions: Both the old and the new version run in parallel for a while, so developers can work with the new version while users are still using the old one. Once the new version is approved, users can be migrated with the click of a button. This reduces the overall downtime and the risk of faulty changes.

However, established platforms, such as Cloud Foundry (www.cloudfoundry.org), explicitly exclude database applications, since the database as single-point-of-truth cannot co-exist both in the old and in the new version at the same time. Whenever we evolve the schema of the database, we have to evolve all the currently existing data and all applications that access the schema as well. Without platform support, this has to be done manually by implementing a data migration procedure, writing adapters for not yet updated users, etc. This makes the database evolution very time-consuming, expensive, and error-prone and thereby a limiting factor for the fast and continuous evolution of businesses.

Thanks to new research results on database evolution, this assumption is not necessarily true anymore. There are tools and concepts that foster agile database evolution and co-existing schema versions within one database (Rahm and Bernstein, 2006). We recently introduced INVERDA, a tool that allows developers to easily create new schema versions by evolving existing ones with a simple descriptive language interface. The new schema version then truly co-exists with other previously created schema versions.

In this paper, we present DECA (Database Evolution in Cloud Applications)—a framework that applies INVERDA for the deployment of database applications in cloud platforms. We tailor INVERDA to the

Mohapatra, A., Herrmann, K., Voigt, H., Lüders, S., Tsokov, T. and Lehner, W.
Seamless Database Evolution for Cloud Applications.
DOI: 10.5220/0006845101970207
In *Proceedings of the 7th International Conference on Data Science, Technology and Applications (DATA 2018)*, pages 197-207
ISBN: 978-989-758-318-6

specific requirements of cloud platforms and evaluate its feasibility in this special but very important area. Our contributions are:

- **Architectural Blueprint for Database Evolution in Cloud Applications.** We propose an architecture to integrate seamless database evolution into existing tools. Applying results from the literature, we guarantee that no data will be corrupted during the evolution and migration. (Section 3)
- **Extensive Evaluation.** We analyze the performance characteristics to equip developers with a guideline for the use of row- or column-stores and we highlight the necessity of statement-wise triggers to gain a significant speed up. (Section 4)

In a word, DECA now facilitates agile and seamless database evolution for cloud application platforms to keep the pace with ever-changing business decisions.
Outline: In Section 2, we discuss modern cloud application development and detail on current advances in database evolution. In Section 3, we apply the advances in database evolution for agile cloud deployment: We present the design of DECA and show how to integrate it in a common cloud platform with a relational database. After an extensive evaluation in Section 4, we conclude the work in Section 5.

2 RELATED WORK

Evolving applications as quickly and as robustly as possible is an omnipresent challenge in software development. To this end, cloud application platforms such as Cloud Foundry implement **Blue-Green Deployment** (https://docs.cloudfoundry.org/devguide/deploy-apps/blue-green.html) to reduce the risk and downtime during the continuous development of cloud applications. Figure 1 shows the general process. The currently running application version (1) is called the blue version. While the new/modified green version is developed, the blue remains live and active (evolution phase). Once the green version is ready, it is deployed and thoroughly tested (2). Once the testing is done, we migrate all users to the green version (migration phase). Now, the requests are routed to the green version so that we can take the old blue version offline (3). However, this approach is not applicable for database application as it will lead to data discrepancy between the green and the blue database version.

To evolve existing applications in an easy, controlled, and robust manner, modern agile software development heavily relies on **Refactorings** (Ambler and Lines, 2012; Fowler and Beck, 1999). Scott Ambler (Ambler and Sadalage, 2006) adapts this approach to the evolution of a production database by proposing more than 100 **Database Refactorings** ranging from simple renamings over changing constraints all the way to exchanging algorithms and refactoring single columns to lookup tables. One essential characteristic of these refactorings is that most of them couple the evolution of both the schema and the data in consistent and easy-to-use operations. The overall goal is to make database development just as agile as software development by changing the game from big upfront modeling to an incremental continuous evolution of the database based on refactorings.

Database researchers pick up on this refactorings-based principle and propose structured **Database Evolution Languages (DELs)**. DELs provide a set of **Schema Modification Operations (SMOs)** which are refactorings on the database schema. SMOs couple the evolution of both the schema and the data with intuitive and consistent evolution operations, e.g., partitioning or joining existing tables. SMOs are way more compact than comparable SQL scripts (Curino et al., 2013; Herrmann et al., 2017). Since manual schema evolution is a heavy, error prone and expensive operation, it is beneficial to have operations like SMOs which can do the required task in a clean and consistent manner without the risk of corrupting data. One of the most advanced SMO-based DELs, we are aware of, is **PRISM** (Curino et al., 2009) that includes eleven SMOs that allow to describe most practical database evolution scenarios (Curino et al., 2008). **PRISM++**, which extends PRISM, includes Integrity Constraint Modification Operations (ICMOs) that allow to create and drop both value and primary key and foreign key constraints with the same charm and simplicity as SMOs (Curino and Zaniolo, 2010; Curino et al., 2013). Further, PRISM++ facilitates update rewriting. This enables developers to automatically adapt applications working on an old schema version to correctly access the data of a new SMO-evolved schema version.

BiDEL is a more recent extension of PRISM that allows to create, drop, and rename both tables and columns as well as splitting and merging tables both vertically and horizontally. BiDEL is shown to be relationally complete (Herrmann et al., 2015) and bidirectional. The latter facilitates co-existing schema version with a system called **INVERDA** (Herrmann et al., 2017). INVERDA automatically makes multiple schema versions accessible in one database—write operations in one version are immediately visible in all other schema versions as well. Further, INVERDA guarantees data independence: No mat-

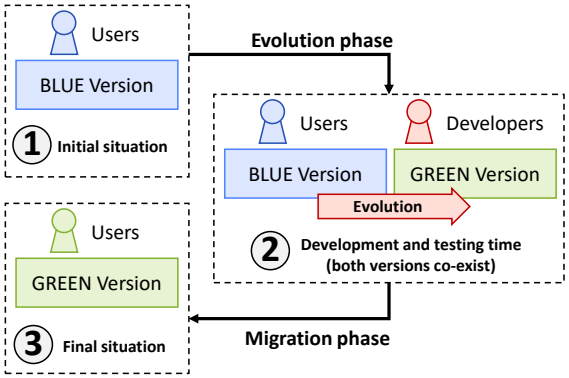


Figure 1: Blue-green deployment of a cloud application.

ter which schema version is physically materialized, every single schema version behaves like a regular single-schema database. Since not all evolutions are information-preserving, InVerDa manages auxiliary information to keep the otherwise lost information. This functionality is exactly the missing piece of the puzzle to implement blue-green deployment of database applications. Our contribution is to show and evaluate the feasibility of InVerDa's approach for the seamless evolution of cloud applications.

In comparison to manually evolving a database with standard SQL, an SMO-based DEL is way easier to learn and to use. However, it still requires a technical understanding of the database evolution process. Therefore, the **CRIUS** project provides simple database evolution where non-technical users are equipped with an intuitive graphical user interface to easily extend a running database (Qian et al., 2010). The database evolution is restricted to adding tables and columns which at least allows to grow the database with the application and prevents from common mistakes. Another interesting compromise between expressiveness and user-friendliness is model-driven database evolution with **MeDEA** (Domínguez et al., 2008). MeDEA equips developers with an editor to specify schema changes in an Entity-Relationship diagram and the respective changes are automatically mapped to the underlying relational database model. Integrating those approaches in our tool is promising future work. We focus only on the recent literature for database evolution here, but we point the interested reader to bibliographies from Roddick (Roddick, 1992) and from Rahm et al. (Rahm and Bernstein, 2006).

Summary. There is plenty of related work for the agile and seamless evolution of application code. Database evolution is an emerging topic in research and there are promising solutions that use SMO-based DELs to make the database evolution faster and more robust. To our best knowledge, there are no existing solutions that apply those results for the seamless evolution of database applications in cloud platforms.

3 DESIGN CONCEPTS

DECA adapts the principle of seamless evolution of applications with blue-green deployment to the evolution of databases in cloud application platforms. Recent advances in database evolution research allow to easily create new schema versions; developers merely write declarative and intuitive evolution scripts with Database Evolution Languages (DELs), such as PRISM++ or BiDEL (Curino et al., 2013; Herrmann et al., 2017), consisting of Schema Modification Operations (SMOs). InVerDa builds upon these languages and automatically generates co-existing schema versions within the database—InVerDa verifiably ensures that no data is lost in any schema version by managing auxiliary information automatically without any developer being involved (Herrmann et al., 2017). We take this powerful result and use it to let the database schema co-exist in both the old and the new version during the evolution and migration of a cloud application. The design concepts presented in this section are generic so they can be adapted for all common database systems—we also highlight specific requirements and nice-to-haves such as statement-wise instead-of triggers.

DECA is a database evolution and migration tool for cloud application platforms. We obtain two limitations from the common blue-green deployment in these platforms: First, DECA is *single-branch*, so we evolve exactly one source schema version to one target schema version. Second, the evolution moves *forward only*, i.e., after the migration, we do not support the blue version anymore.

In the following, we describe the process of the seamless database evolution with DECA in a cloud platform in Section 3.1. Upon this, we detail the user interfaces in Section 3.2 and present our system integration in Section 3.3.

3.1 The DECA Process

DECA generally adapts blue-green deployment from cloud application platforms to database evolution and has two phases: In the first phase, the new green schema version is already fully accessible but the data remains physically in the original blue schema version and all applications continue working on the old blue version. Merely the developers and test users can already access the new green schema version to develop and evaluate new features, bug-fixes, etc. In

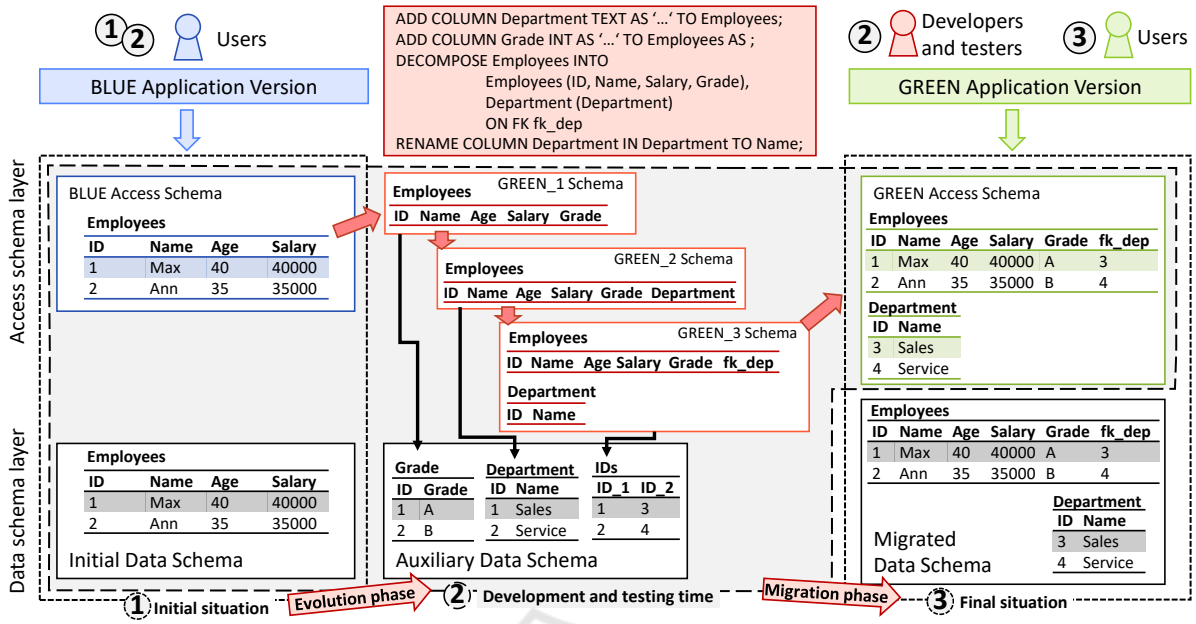


Figure 2: Blue-Green database evolution and migration process.

the second phase, the data is physically migrated to the new green schema version. After completion, the new green version can be accessed by all applications and we take the old blue schema version offline.

Please consider an exemplary human resource software as shown in Figure 2, which is deployed in a cloud application platform such as Cloud Foundry. In the *initial situation (1)*, there is a simple table for the employee details as shown in the blue access schema on the left side of the figure. User applications access this blue schema version on the upper *access schema layer* just as any other regular database. The access schemas are implemented with updatable views—the data is persisted in tables in the *data schema layer*. We will evolve and eventually migrate this database.

Evolution Phase. Developers evolve the application and the database schema to create a new green version by executing DEL-scripts. Assume new features require to add the two new columns "Department" and "Grade". Developers express this intended evolution merely by executing two ADD COLUMN SMOs as shown in the upper middle of Figure 2. While the first ADD COLUMN SMO produces the intermediate schema version GREEN_1, the second ADD COLUMN SMO results in GREEN_2. During testing, the team decides to normalize the Employees table and split away the Department column while creating a new foreign key—this results in the GREEN_3 schema version. For better readability, the team renames the Department column into Name to finally end up in the GREEN access schema.

We are now in the *state of development and testing (2)*, where both the blue and the green schema version are fully accessible. Users can still use the old version, while the developers and testers can use the new version just as usual. DECA uses the code generation of InVerDa (Herrmann et al., 2017) to create the green access schema with views—any write operation on these views is executed by instead-of triggers that propagate the write operation back to the blue data schema. Since not all information can be stored in the blue data schema, DECA automatically manages auxiliary tables that keep all the otherwise lost information. Both the blue and the green schema version are now guaranteed to behave like regular single-schema databases even though the data is still stored only in the blue schema version.

Migration Phase. Till now, users did not notice any change as they continuously used the blue version. When the developers and testers confirm that the new green version works correctly, we enter the migration phase by instructing DECA to change the physical data schema to the green schema version. This is essentially done with the click of a button. On the database side, DECA creates tables in the data schema according to the green schema version and populates them with data by simply reading the data in the green access schema. Afterwards, the data schema from the blue version as well as the auxiliary data schema can be cleared so that only the new green data schema remains. The simultaneous migration of the application is realized by common cloud application platforms such as Cloud Foundry. After running the physical migration we are in the *final state (3)* and users use

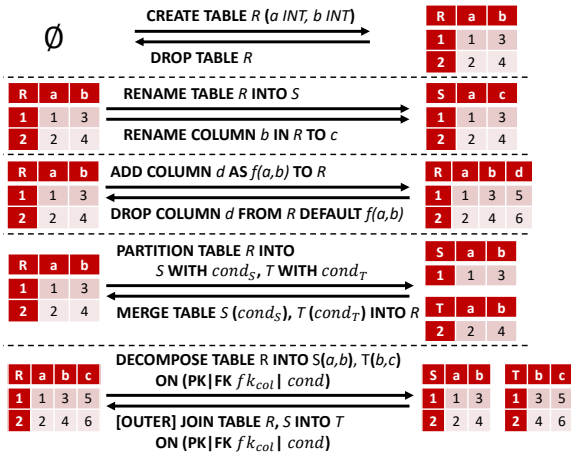


Figure 3: SMOs of DECA's DEL.

the new green version now—the data access schema is called GREEN(migrated) then. At this state, the deployment cycle is finished and DECA is ready to start the next evolution in the same way.

Currently, the migration phase causes limited availability of the application for a short period of time, since copying the data from the blue to the green data schema locks the data tables. Hence, data cannot be written but can be still read. To achieve a zero-downtime migration, we need proper copy mechanisms that migrate the data silently without blocking the operations and keep the already migrated data in sync. Further, the amount of data to migrate can be reduced by combining SMOs and in-place migrations. These opportunities are left for future research.

Summary. We follow the blue-green deployment process: While the blue production version remains accessible, developers create a new green schema version that co-exists and allows independent development and testing. Finally, we migrate to the green schema version and drop the blue version.

3.2 DECA's Interfaces

Developers interact with DECA two times: once in the evolution phase to create the new green schema version and once to trigger the migration phase. In the latter phase, data is copied from the blue data schema to the green data schema—this process is fully automated and does not require any further interaction on the developer's side. During the evolution phase, developers use the bidirectional SMO-based DECA-DEL to describe the evolution of both the schema and the data from the old blue to the new green version as well as the propagation of write operations from the new green back to the old blue version. The DECA-DEL is equivalent to BiDEL (Herrmann et al., 2017).

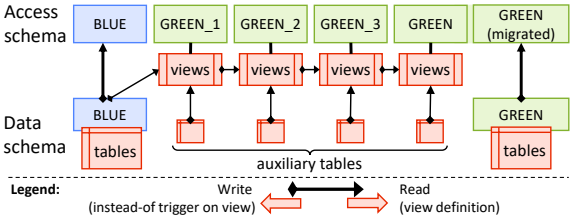


Figure 4: Generated views and triggers for example (Fig. 2).

Besides DECA-DEL's relational completeness, this guarantees that both the blue and the green schema version behave like regular single-schema databases, so we take this for granted.

The DECA-DEL is summarized in Figure 3. According to their counterparts in SQL-DDL, there are SMOs to create, drop, and rename both tables and columns. Further, we can partition a table horizontally. The partition criteria may overlap and must not necessarily cover the whole relation. To ensure bidirectionality the inverse SMO for merging two tables horizontally also takes such partition criteria—this is required to propagate new records in the green schema version back to the intended partition in the blue one. Similarly, the last two SMOs allow to decompose or join tables vertically. Either way, we need to specify how records from the decomposed side are joined—this may be according to the primary key, a specified foreign key or any evaluable condition. Further, the join can have inner or outer join semantics.

Summary. Developers have interfaces to trigger both the evolution and the migration phase; for the evolution phase, DECA equips developers with a powerful SMO-based DEL to easily describe the evolution from the blue to the green schema version.

3.3 System Integration

DECA is an independent component adjacent to a DBMS. DECA is not a part of the DBMS but generates and deploys database artifacts that are then executed in the database to evolve, test, and migrate it. As can be seen in Figure 2, we separate the database into an access schema layer and a data schema layer. The access schema layer consist of one or two schema versions which comprise of views and triggers and the data schema layer consists of persistent data in the form of tables. The application interacts with the access schema instead of the data schema. This decoupling avoids directly modifying the data schema and to achieve concurrently active access schemas.

Figure 4 shows the access schema and the data schema for the example in Figure 2 with the generated views and triggers. The green access schema version is implemented with a sequence of views starting at

the blue data schema. The views are made updatable with instead-of triggers that propagate write operations also from the green to the blue schema version. Since SMOs may add information capacity during the evolution, we use auxiliary tables to prevent information loss. Consider our example, where we add a new column named Salary to the employees table. Since this column does not exist in the blue version, we create an auxiliary table which stores the data of the new column. The view in the green schema version now joins the blue data table and the auxiliary table. Write operations on the green schema version are propagated by an instead-of trigger back to both the data table and the auxiliary table.

For each view, we define three instead-of triggers, i.e., for inserts, updates, and deletes. In most database systems, these triggers are row-wise triggers by default, i.e., they would execute one row at a time. This causes a significant propagation overhead when multiple records are updated at a time. Unfortunately, many DBMSes do not support statement-wise instead-of triggers. However, we show that it would be worth it to implement **statement-wise instead-of triggers** as they can significantly speed up the propagation of write operations from the green access schema to the blue data schema, as we evaluate in Section 4. To explore the benefits, we simulate their behavior via stored procedures that take the arguments of the write operation as parameters. Revisiting our example from Figure 2, assume we want to delete several employees because some departments of the company got sold or we increase the salary of all employees by 10%. When we execute such write operations on the green schema version before the migration, triggers are used to propagate them from the green access schema through the SMOs back to the blue data schema. With row-wise triggers, this would cause one trigger call on each intermediate version for each single affected employee. A statement-wise instead-of trigger could instead fire one delete operation on the blue data schema as well as on each affected auxiliary table. Since all these executions are done on a bulk of records at a time, it is very fast compared to the row-wise execution.

Summary. DECA generates the co-existing green schema version with views and triggers upon the blue data schema. Especially write operations of multiple records benefit from statement-wise triggers.

4 EVALUATION

DECA easily realizes complex evolutions and migrations of the database in cloud applications. We shorten the downtime and made the migration process more predictable. A new green version co-exists with the old blue version to allow extensive testing—when the developers are sure that the new version works correctly, we instruct DECA to physically migrate the database with the click of a button. Developers specify the evolution as sequences of intuitive SMOs that allow to generate the co-existing green schema version without changing the data schema and without restricting the availability of the blue schema version. There is no hand-written data propagation between schema version, hence we eliminate a common source of failures. The same holds for the migration code, which is completely generated from the SMO-based evolution script. There is no hand-written code involved that could fail or delay the migration.

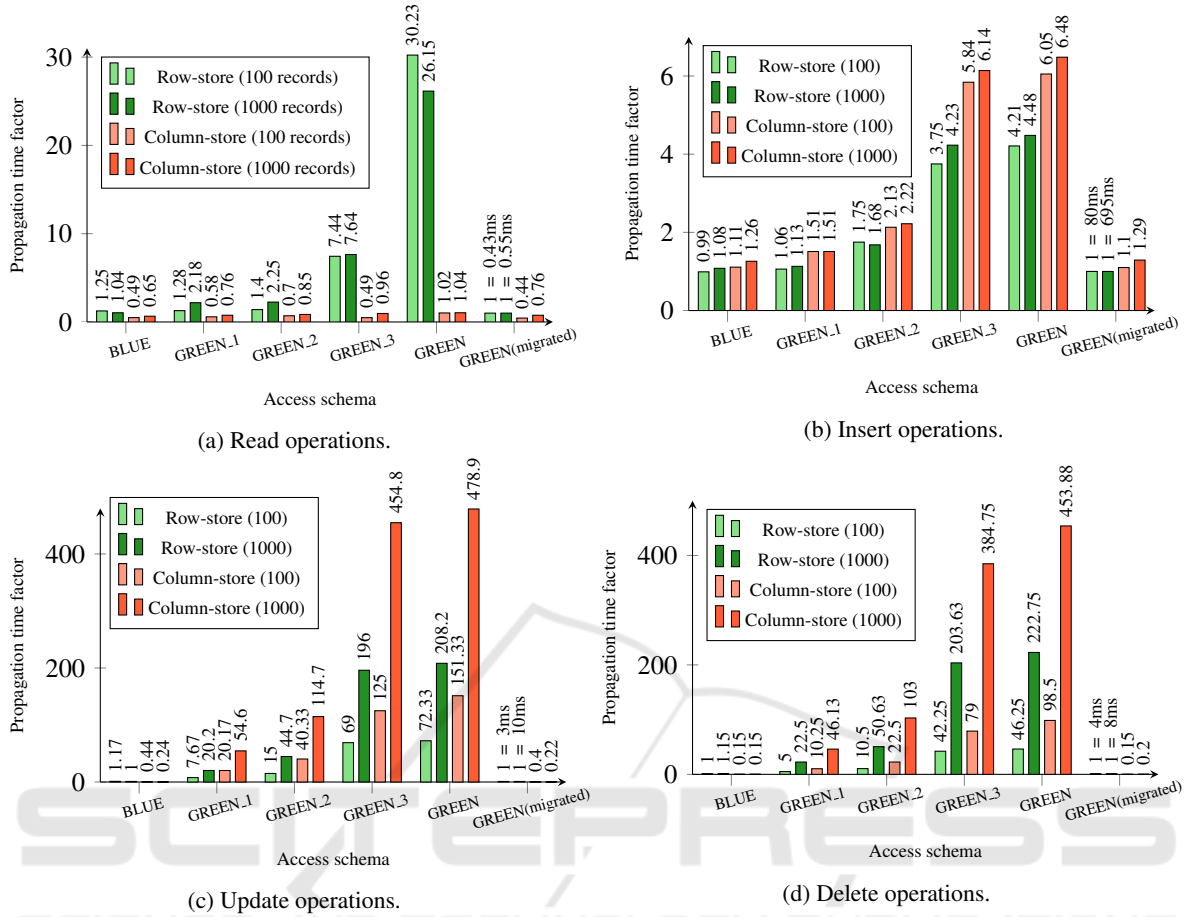
Beyond this functional contribution, we now present an extensive evaluation of DECA. We analyze the overhead caused by DECA to show that it is reasonable. Further, we focus on the impact of row/column stores and the use of row- or statement-wise triggers to further reduce this overhead. In Section 4.1, we evaluate our introductory example in detail. Afterwards, we zoom into the effect of single SMOs in Section 4.2.

4.1 Exemplary Scenario

We evaluate the exemplary evolution and migration from Figure 2 on Page 4 to analyze DECA's benefits.
Setup: We load the employees table with 150,000 sample records and evaluate both read and write operations on both the blue and the green schema version as well as on the intermediate schema versions as shown in Figure 4 to analyze the impact of the evolution's length on the performance overhead. In Section 4.1.1, we measure the read/write performance for batch sizes of 100 and 1000 records in both a row- and a column-store with common row-wise triggers. In Section 4.1.2, we evaluate statement-wise triggers as a promising alternative that can significantly speed up the propagation of write operations between the schema versions. Finally, we analyze the time for executing the actual migration of the data schema from blue to green in Section 4.1.3. All experiments are conducted in the SAP HANA in-memory DBMS on a Linux workstation with 2x6 Core Intel Xeon CPU (3.5GHz) and 96GB main memory.

4.1.1 Propagation Overhead and Store Layout

The general expectation is that the propagation overhead increases with a growing number of SMOs in the evolution. To quantify this overhead, we compare the use of DECA to not using it. An evolution done



(a) Read operations.

(b) Insert operations.

(c) Update operations.

(d) Delete operations.

Figure 5: Comparing Row-store vs. column-store.

without DECA would require to manually implement an evolution and migration program, e.g. in SQL, which would ultimately result in GREEN(migrated). Therefore, we consider the readings on the migrated green schema version as the baseline for our measurements. We represent all measurements as a factor of its respective baseline. For instance in Figure 5a we see that a SELECT with batch size 1000 on GREEN_2 in a row-store takes 2.25 the time of the baseline, which is a SELECT with batch size 1000 on GREEN(migrated) in a row-store. The batch size is the number of rows selected, inserted, updated, or deleted by a single query. We use row-wise triggers for these general experiments and will reduce the apparently high overheads with statement-wise triggers in the next section.

Figure 5a shows the propagation overhead for read operation for both row- and column-stores. The main observation is that the propagation overhead increases from BLUE to GREEN for both batch sizes especially in row-stores—in column-stores the effect is less dominant. As expected, the propagation time increases with each SMO in the evolution.

Further, the time taken by a SELECT operation in a column-store compared to row-stores is significantly smaller. We attribute this to the heavily read-optimized storage structures in column-stores. For many SMOs, the propagation logic joins auxiliary tables with the data tables, which works perfectly with the query processing patterns of column-stores.

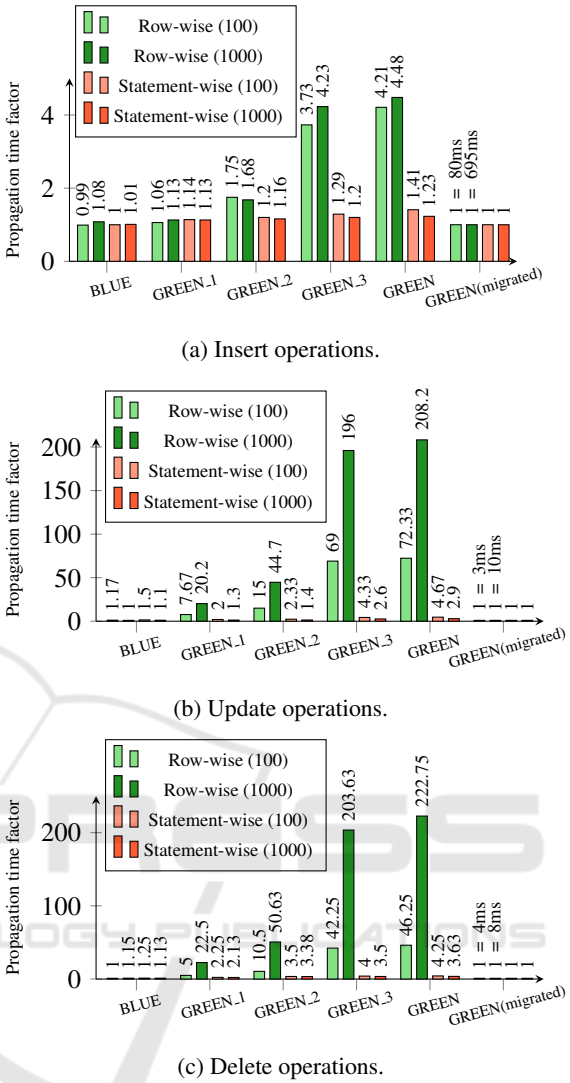
We conduct the same evaluation for insert operations and present the results in Figure 5b. Again, the propagation overhead increases for both row- and column-stores when moving from BLUE schema to GREEN schema. The complex decomposition with a foreign key between GREEN_2 and GREEN_3 requires the generation of new surrogate key values and causes a steep rise in the propagation overhead. Between GREEN_3 and GREEN the rise is not that high as we merely rename a column. In GREEN(migrated), the data schema matches the green access schema; hence, the propagation is no longer required and the performance goes back to normal. Comparing row- and column-store, we now see that row-stores perform better for insert operations: In the worst case scenario, the propagation overhead for

inserts in a column-store is around factor 6.5 while it is only factor 4.5 for row-stores.

Figures 5c and 5d show the propagation overhead for update and delete operations. Again, the propagation time increases with each SMO in the evolution and drops down again after the migration. In comparison to the GREEN(migrated) schema version, the worst case propagation overheads for both row- and column-stores are two orders of magnitude higher. Further, the worst case for an update operation in a column-store is roughly 2.3 times higher than in a row-store. Similarly, it is around 2 times higher for a delete operation.

Finally, a word on the effect of the number of records on the propagation overhead. For an insert operation, the propagation overhead grows roughly the same for 100 and 1000 records; however, the baseline value for 1000 record is already 8.6 times higher. Whereas, a select operation generally causes a higher overhead with growing batch sizes—since the query execution is often below millisecond, there might be a measuring error attached. The update and delete operations show roughly similar overhead growths for the two batch sizes since both operations require to find the affected records before writing. While the blue access schema allows deleting and updating the whole batch with one single statement, the row-wise triggers are fired for every affected row in the green version.

Summary. As expected, the propagation overhead increases with the number of SMOs in the evolution. Further, column-stores facilitate significantly faster reading but row-stores allow faster writing. Hence, the common advantages and disadvantages of row- and column-stores also hold for the fixed data access patterns determined by the used SMOs.



(a) Insert operations.

(b) Update operations.

(c) Delete operations.

Figure 6: Comparing row-wise and statement-wise triggers.

4.1.2 Row-wise vs. Statement-wise Trigger

The propagation of write operations through sequences of SMOs is implemented with instead-of triggers. We show that statement-wise triggers are more feasible than common row-wise triggers, since they cause a significantly smaller propagation overhead. Figures 6a to 6c show the propagation overhead for insert, update, and delete operations respectively. Again, GREEN(migrated) is taken as the baseline. For instance, in Figure 6a the duration of inserting 100 records at GREEN_3 with row-wise triggers is 3.73 times higher than at GREEN(migrated).

Figure 6a shows the performance comparison of row-wise vs. statement-wise triggers for insert operations. In both the cases, as we move along the SMOs in the evolution, the propagation overhead increases. The worst case propagation overhead for statement-wise trigger for batch size of 1000 is 1.23 times that of the baseline value. With row-wise trigger, the propagation overhead is 4.28 times which is still around 3.6 times higher than that of statement-wise triggers. Hence, insert operations can greatly benefit from statement-wise instead-of triggers.

Figures 6b and 6c show the same measurements for update and delete operations. Similarly, the update and delete operations show an increasing propagation overhead when moving along the sequence of SMOs. The worst case propagation overhead with row-wise triggers is two orders of magnitude more than the baseline value for both the update and the delete operation. The propagation overhead for updating the GREEN schema version with batch size 1000 is 208 times the baseline values—doing the same with statement-wise triggers is only a factor of 2.9, which is roughly 69 times below row-wise triggers. Similar

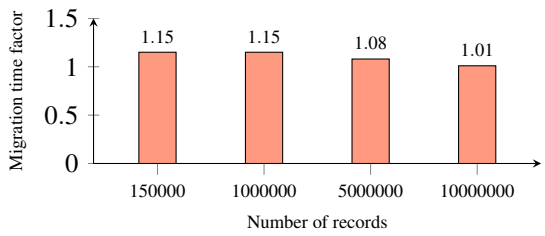


Figure 7: Migration Overhead via DECA.

behavior can be seen for the delete operation as well.
Summary. The comparison between a row-wise and statement-wise trigger showed that the latter performs significantly better. Instead of propagating the write operation for every single affected record with row-wise triggers, we merely propagate different write statements that can affect multiple records. Hence, we deem statement-wise instead-of-triggers as essential to reduce the propagation overhead of write operations on the green version to an acceptable level.

4.1.3 Migration Overhead via DECA

After the developers and testers confirm the new green version, we use DECA to actually migrate the database. At the same time, the cloud application platform—Cloud Foundry for instance—moves all users to the already deployed green version, so we can turn off the former blue version. During the migration phase, there are many operations carried out apart from the data migration. In order to understand the actual impact of the database migration time, we isolate the actual data migration. We compare the result to a simple table migration operation where the same amount of data is copied from one table to another table with the same schema. We start our migration for 150,000 records and then move to 1,000,000 then 5,000,000 and finally to 10,000,000 records. For each record set size, we run the migration five times, exclude the highest and the lowest migration times, and take the average of the rest.

Figure 7 shows the times taken for the migration via DECA as a factor of the simple migration. For DECA, we record a slightly higher migration time because it needs to read the values via the sequence of SMOs before writing them in the migrated data schema. The migration time with DECA is close to that of a naïve migration and it scales nicely with the number of records. The more records the higher the cost for the actual data movement and the smaller the impact of the data access propagation for reading.
Summary. The migration time with DECA is mainly determined by the actual transfer of the data—the overhead introduced by the migration through the sequence of SMOs is negligibly small.

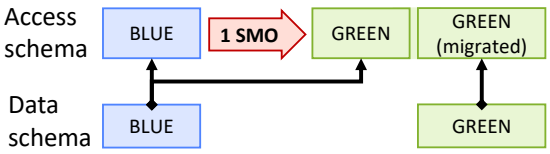
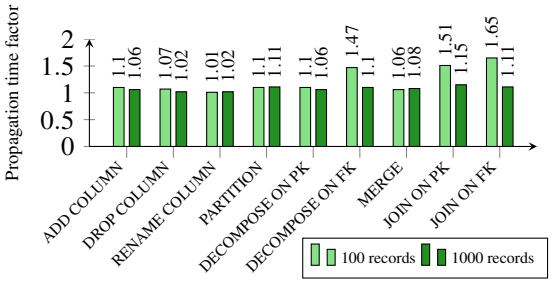


Figure 8: Scenario with a single SMO.

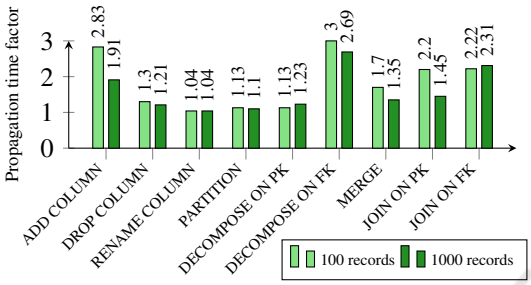
4.2 Single SMO Evolution

After the general evaluation of our exemplary scenario, we now deep dive into the propagation overhead of single SMOs and analyze them in isolation. The knowledge about the different characteristics of the different SMOs helps developers to design and plan evolutions/migrations more profoundly.
Setup. We use a row-store with statement-wise triggers. The general setup is shown in Figure 8. The green version contains one or two simple tables that are loaded with 150,000 sample records. We obtain the green version by applying exactly one SMO at a time. Again, we execute read and write operations of 100 or 1000 records on the green schema version. The baseline is the execution time with data being already migrated to the green data schema. Before the migration, the operations take more time on the green schema version, as they need to be propagated to the blue data schema first. The factor of this measured time compared to the baseline is the overhead that is actually caused by the respective SMO. The overhead for read operations is in the range below milliseconds and thereby subject to measuring errors; hence, we focus on the write operations here. E.g., in Figure 9a, propagating an insert operation with 100 records through the ADD COLUMN SMO from the green to the blue schema version takes 1.1 times as long as executing it directly on the migrated green schema.

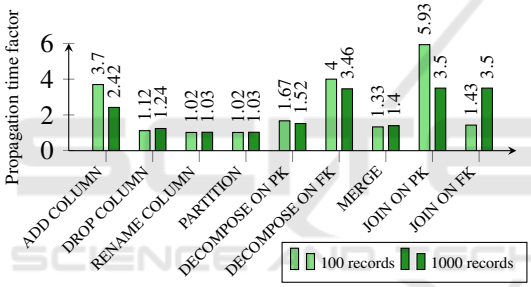
Figure 9a shows the overhead of propagating an insert operation through the different SMOs. In general, the overhead caused by insert operations are very small—usually below 10%. Merely, the SMOs that involve writing to multiple tables show higher propagation overhead. Assume we join two tables from the blue schema to one table in the green schema. Whenever we insert data to the joint view in the green schema version, we need to write to the two data tables. Similarly, the partitioning and the decomposition on a foreign key involve auxiliary tables that cause a noticeable overhead. Another interesting observation is that the relative overhead decreases when we write more records. We use a statement-wise trigger: the time for transforming the statement from the green to the blue version are independent of the number of records. Hence this overhead is negligible small compared to writing more and more records.



(a) Performance of single SMOs (insert)



(b) Performance of single SMOs (update)



(c) Performance of single SMOs (delete)

Figure 9: Write propagation through single SMOs.

Figures 9b and 9c show the same evaluation for update and delete operations respectively. Again, SMOs that involve multiple joins or write operations on multiple tables have a higher propagation overhead. For instance, propagating an update through a DECOMPOSE ON FK SMO involves executing a join and updating both data tables and auxiliary tables. As a result, it is 3 times slower than the baseline for batches of 100 records. Further, e.g., the propagation of a delete operation through a JOIN ON PK involves deleting records from both the tables in the blue data schema, which causes a 5.9 times higher execution time for 100 records.

Summary. Generally, the overhead for propagating write operations through single SMO is very small. Merely SMOs that involve multiple joins or writing to multiple tables, can cause higher overheads. These insights help developers to better plan resources for development, testing, and migrations.

5 CONCLUSION

Reacting to changing wishes and requirements of customers as fast as possible is crucial for economic success. The pace of implementing business decisions is often limited by evolving the company's IT-system accordingly. We can quickly evolve applications with agile refactoring-based software development methods and deploy them seamlessly with blue-green deployment in cloud application platforms. However, this does only work as long as no database is involved, which is rarely the case. Evolving both the database schema and the data without corrupting the data is still an error-prone and expensive challenge—not to mention co-existing schema versions for the blue-green deployment.

Our goal was to make the database evolution in cloud application platforms just as simple and robust as the evolution of the application code. Therefore, we discussed current advances of agile database evolution in the literature in Section 2. Researchers adapt the refactoring-based evolution from application code to databases by equipping developers with SMO-based DELs. SMOs carry enough information to evolve and migrate the database automatically and to let schema versions co-exist in a database—writes in one version are immediately visible in all other versions and each schema version is guaranteed to behave like a regular single-schema database.

This is exactly, what we needed to extend the seamless blue-green deployment of cloud applications to the database layer. In Section 3, we presented an architectural blueprint of our tool DECA. Developers write SMO-scripts to easily create a new green schema version that co-exists with the blue production version. This allows intensive development and testing without limiting the availability of the application. After testing, developers migrate the database with a click of button, hence DECA eliminates the risk of faulty evolutions and corrupted data during both the evolution and the migration of the database.

Finally, we presented an extensive evaluation of DECA in Section 4. There is an overhead for propagating read/write operations from the green access schema to the blue data schema. However, we have shown that this overhead can be reduced to a minimum by using a row-store database with statement-wise instead-of triggers. Developers can safely use DECA to evolve and migrate the database very quickly without the risk of corrupting the data and with a reasonable performance overhead. In sum, the blue-green deployment of cloud applications including a database now facilitates fast and solid implementations of spontaneous business decisions.

REFERENCES

Ambler, S. W. and Lines, M. (2012). *Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise*. IBM Press.

Ambler, S. W. and Sadalage, P. J. (2006). *Refactoring databases: Evolutionary database design*, volume 4. Addison-Wesley Signature.

Curino, C., Moon, H. J., Deutsch, A., and Zaniolo, C. (2013). Automating the database schema evolution process. *The VLDB Journal*, 22(1):73–98.

Curino, C., Moon, H. J., Ham, M., and Zaniolo, C. (2009). The PRISM Workwench: Database Schema Evolution without Tears. In *International Conference on Data Engineering (ICDE)*, pages 1523–1526. IEEE.

Curino, C., Tanca, L., Moon, H. J., and Zaniolo, C. (2008). Schema evolution in wikipedia: toward a web information system benchmark. In *International Conference on Enterprise Information Systems (ICEIS)*, pages 323–332. Springer.

Curino, C. and Zaniolo, C. (2010). Update rewriting and integrity constraint maintenance in a schema evolution support system: PRISM++. *Proceedings of the VLDB Endowment (PVLDB)*, 4(2):117–128.

Domínguez, E., Lloret, J., Rubio, Á. L., and Zapata, M. A. (2008). MeDEA: A database evolution architecture with traceability. *Data and Knowledge Engineering (DKE)*, 65(3):419–441.

Fowler, M. and Beck, K. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley.

Herrmann, K., Voigt, H., Behrend, A., and Lehner, W. (2015). CoDEL – A Relationally Complete Language for Database Evolution. In *European Conference on Advances in Databases and Information Systems (ADBIS)*, pages 63–76. Springer.

Herrmann, K., Voigt, H., Behrend, A., Rausch, J., and Lehner, W. (2017). Living in Parallel Realities: Co-Existing Schema Versions with a Bidirectional Database Evolution Language. In *International Conference on Management Of Data (SIGMOD)*, pages 1101–1116. ACM.

Qian, L., LeFevre, K., and Jagadish, H. V. (2010). CRIUS: user-friendly database design. *Proceedings of the VLDB Endowment (PVLDB)*, 4(2):81–92.

Rahm, E. and Bernstein, P. A. (2006). An online bibliography on schema evolution. *ACM SIGMOD Record*, 35(4):30–31.

Roddick, J. F. (1992). Schema Evolution in Database Systems – An Annotated Bibliography. *ACM SIGMOD record*, 21(4):35–40.