NoSQL Schema Evolution and Big Data Migration at Scale

Meike Klettke
University of Rostock
Rostock, Germany
meike.klettke@uni-rostock.de

Uta Störl
University of Applied Sciences
Darmstadt, Germany
uta.stoerl@h-da.de

Manuel Shenavai
University of Applied Sciences
Darmstadt, Germany
manuelshenavai@hotmail.de

Stefanie Scherzinger
OTH Regensburg
Regensburg, Germany
stefanie.scherzinger@oth-regensburg.de

Abstract—This paper explores scalable implementation strategies for carrying out lazy schema evolution in NoSQL data stores. For decades, schema evolution has been an evergreen in database research. Yet new challenges arise in the context of cloud-hosted data backends: With all database reads and writes charged by the provider, migrating the entire data instance eagerly into a new schema can be prohibitively expensive. Thus, lazy migration may be more cost-efficient, as legacy entities are only migrated in case they are actually accessed by the application.

Related work has shown that the overhead of migrating data lazily is affordable when a single evolutionary change is carried out, such as adding a new property. In this paper, we focus on long-term schema evolution, where chains of pending schema evolution operations may have to be applied. Chains occur when legacy entities written several application releases back are finally accessed by the application. We discuss strategies for dealing with chains of evolution operations, in particular, the composition into a single, equivalent composite migration that performs the required version jump.

Our experiments with MongoDB focus on scalable implementation strategies. Our lineup further compares the number of write operations, and thus, the operational costs of different data migration strategies.

Keywords—NoSQL Databases, Schema Evolution, Data Migration Strategies, Lazy Migration, Lazy Composite Migration, Incremental Migration, Predictive Migration

I. INTRODUCTION

With agile development, applications evolve continuously. This clashes with the tradition of declaring a fixed database schema up front. Let us consider the example of an online role playing game, with player entities stored in the production data store. Even though the data store itself may be schema-flexible, the currently serving application code libraries [2]: For simply adding the points property, a Java attribute is added to the declaration of class Player. This trivial code change is shown in Figures 1(a) and 1(b). Thus, whenever a legacy player is loaded into the application, and later written back to the data store, this player has been successfully migrated to the latest schema.

Yet tackling data migration via object-NoSQL mappers inevitably introduces latency, due to the inherent performance overhead of carrying out changes within the application code. Moreover, developers need to take great care that all object mapper class declarations are backwards-compatible with all earlier schema versions, to avoid runtime exceptions during lazy migration [3]. Also, with more complex schema changes, the data migration code becomes increasingly complex, and technical debt piles up accordingly.

In this paper, we explore new approaches. One conjecture is that it is more sustainable to tackle data migration within the data store, rather than within the application code. A second conjecture is that when chains of pending evolution operations are to be applied lazily, it makes sense to combine them into composite operations, to more efficiently bridge the version jump to the latest schema version.

Contributions: This paper makes these contributions:

- We discuss data migration as an optimization problem in the context of the four dimensions time, amount, operation execution, and location.
- We consider basic schema changes such as adding, deleting, or renaming a property, and also more advanced schema changes such as moving or copying a property between entities. This is the scope of our schema evolution language, which we have introduced.

...
in earlier work [4]. We study chains of pending schema changes that are to be applied lazily, and declare rules for deriving lazy composite migrations.

- We experimentally compare three different implementation strategies for lazy data migration carried out in the popular NoSQL data store MongoDB: (1) We implement schema changes within the application code, (2) we use the native update API provided by the data store, and (3) we use database stored procedures. We compare the runtime, throughput, and the number of physical writes when chains are executed in sequence (lazy stepwise), or as lazy composite migrations.

- Based on simulation analysis, we compare different data migration strategies: Eager, lazy stepwise, and lazy composite migration. Furthermore, we blend lazy and eager migration into incremental and predictive migration. Our in-depth analysis reveals major impact parameters, such as the length of version jumps to be bridged, and the application access pattern.

**Structure:** This paper is structured as follows. In Section II, we sketch the scenario of a schema being imposed by the application code onto an otherwise schema-flexible data store. We further discuss data migrations in the context of four dimensions. In Section III, we introduce the formal preliminaries underlying our work. Section IV introduces version jumps and the composition rules for evolution operations. In Sections V and VI, we evaluate different strategies for implementing data migration, using experiments and simulation-based analysis. Section VII references related work. We then conclude with Section VIII.

II. SCHEMA EVOLUTION IN AGILE DEVELOPMENT

We consider a common scenario in agile software development, where applications are deployed continuously against a NoSQL production data store such as MongoDB.

A. Application-imposed Schema

The application code expects to load and store entities according to a given data model, or schema. With each new release of the application code, this schema may change. We therefore distinguish different schema versions.

There are different scenarios how the current application schema can be made available:

1) In professional development, applications frequently use object-NoSQL mappers for the marshalling of entities into objects of the application space. Then, annotated class declarations implicitly declare a database schema. Figure 1 shows the declarations of classes `Player`, `Mission`, and `Stats` in a first application release. All classes carry a designated attribute `version` to keep track of the schema version.¹

2) The schema may also be explicitly declared: For instance, the schema-flexible data store MongoDB allows for an optional schema to be registered. MongoDB then ensures that all entities validate against this schema [6].

3) Alternatively, the schema may be extracted from the data instance in a reverse-engineering step (c.f. [7]).

Figure 2 summarizes these scenarios. The arrows denote the data flow. To the top, we see the application deployed to the production environment, due to a new release. The application version changes from $v$ to $v+1$. In the middle, we see that the current schema, as expected by the application code, changes accordingly. The NoSQL data store, however, is schema-flexible and contains entities in different versions.

As the application code changes to version $v+1$, all entities persisted in the data store so far become legacy entities, and may not adhere to the new schema $S_{v+1}$. We assume that the mapping from schema $S_v$ to its successor schema $S_{v+1}$ is described by schema evolution operations. Again, the mapping may be explicitly declared, or derived by comparing the different versions of the application code.

---

¹Assuming that class declarations carry a dedicated `version` attribute is reasonable. Empirical analysis of open source projects shows that maintaining timestamps or versions in persisted entities is common practice [5].
In this paper, we capture schema evolution operations in a language that we have already introduced and employed in earlier work [4], [8], [9]. Figure 3 shows the grammar. The schema evolution language contains single-type operations which add, delete, or rename properties from all entities of a given class (or entity type, as defined later in Section III), e.g., adding a property points to all players. The language further allows multi-type operations to copy or move properties between classes, e.g., copying a player’s points to all of his or her Mission entities. We assume that all multi-type operations that are executed are safe [4], so their result is well-defined and does not depend on the execution order.

Copying is an important schema change with NoSQL data stores, where schema denormalization is common.

B. Dimensions of Data Migration

We consider four orthogonal dimensions of data migration triggered by schema evolution:
1) Time
2) Amount
3) Operation Execution
4) Location

These dimensions are now described in detail.

Time: Within the time interval starting with the introduction of a new schema version until the newly released application accesses a persisted entity, the data migration has to be completed. In this paper, we consider different data migration strategies. Eager migration transforms all legacy entities when a new schema version becomes available. Lazy migration changes an entity only when it is actually accessed by the application. Eager and lazy migration may be blended, as discussed in Section VI-C, in order to reduce the average latency of accessing an entity.

Amount: We further consider the amount of data that we migrate. If an application requests an entity and there is a pending copy or move operation, additional entities are likely to be affected. Now the question is whether these affected entities should be migrated as well, possibly causing further, cascading migration operations. This tradeoff requires a realistic cost model.

Operation Execution: By operation execution, we describe how schema evolution is carried out. For instance, when a legacy entity from several versions back needs to be migrated, the pending schema changes may be either applied as stepwise or as composite migrations (c.f. Section IV).

Location: The fourth dimension is more technical and describes the location in the software stack where the migration is carried out. For instance, orchestrating migration from within the application code can be expected to be slower than migration via database-provided update mechanisms or database-internal stored procedures. At the same time, depending on the complexity of the migration operations and the functionality of the NoSQL data store, it may not even be possible to realize all migration operations with native database operations. We refer to Section V for a discussion.

III. PRELIMINARIES

In the following, we provide the formal foundations for the rest of this paper. We begin with an introduction of some core terminology and then proceed with the schema evolution and migration operations.

A. Entities, Entity Types, and Schemas in NoSQL Databases

An entity \( e_{t,v} \) is a persisted object in a NoSQL database. It carries a type identifier \( t \) (denoting, for instance, a Player or a Mission entity), and further a set of properties. Properties consist of a name and a value. A value can be atomic (e.g., a string or numeric), or structured (an object or an array). Structured properties can be nested. Each entity carries a schema version number \( v \). In a document oriented NoSQL data store such as MongoDB, we regard a (JSON) document as an entity. In a column-family database such as Cassandra, each row is an entity.

An entity type \( E_{t,v} \) declares the structure of all entities that share the type identifier \( t \) (in our example, Player or Mission) and the schema version number \( v \). In the entity type, structural constraints of the entities associated with this entity type are declared. Thus, the entity type declares a set of common properties and also whether a property is required or optional. In document oriented NoSQL data stores, the entity type may be declared using JSON schema or some database-proprietary schema language. In a column-family data store, the entity type is usually represented by the tabular structure.

The schema \( S_v \) expected by an application release \( v \) is the set of all entity types \( E_{t,v} \).

B. Schema Evolution and Migration Operations

The schema evolution operation \( \text{evolve}_{v+1} \) defines the transition of the schema \( S_v \) in version \( v \) into the schema \( S_{v+1} \). Thus, \( \text{evolve} \) operates on the level of entity types. We write:

\[
S_{v+1} \leftarrow \text{evolve}_{v+1}(S_v)
\]

The syntax of schema evolution operations as used in this paper is shown in Figure 3.

The single-type operations add, delete, and rename apply to one entity type at a time. The multi-type operations move and copy operate on two or more entity types and are vital in NoSQL data stores, where denormalization is common when joins are not natively supported.

Migration operations carry out the structural changes at the level of the actual entities. We assume that given an evolution operation \( \text{evolve}_{v+1} \), a corresponding migration operation \( \text{migrate}_{v+1} \) can be derived. As the actual definition of the migration operation depends on the interfaces of the underlying NoSQL data store, we treat migration operations on an abstract level. A single-type migration operations
operation (add, delete, or rename) transforms an entity from version \( v \) to version \( v + 1 \). We therefore write

\[
\epsilon_{t,v+1} \leftarrow \text{migrate}_{v+1}(\epsilon_{t,v}).
\]

Figure 4(a) visualizes the relationship between the operations \( \text{evolve} \) and \( \text{migrate} \) for single-type operations.

The multi-type operations move and copy contain a join condition that involves entities of more than one entity type. Here, a prerequisite is that all affected entities are migrated from schema version \( v \) into version \( v + 1 \). The copy operation updates the entities of one entity type, whereas the move operation updates the entities of two different entity types. For instance, the following migration operation can encode a move operation:

\[
(e_{t1,v+1}, e_{t2,v+1}) \leftarrow \text{migrate}_{v+1}(e_{t1,v}, e_{t2,v})
\]

Figure 4(b) shows a multi-type schema evolution and the related migration operation. The downward facing arrow between the abstraction levels expressed by operation \( \text{evolve}_{v+1} \) and \( \text{migrate}_{v+1} \) illustrates the following: For each schema evolution operation, a corresponding migration operation can be generated that realizes the structural changes at the level of the entities.

IV. VERSION JUMPS AND COMPOSITE MIGRATIONS

When a legacy entity from several versions back has to be migrated to the most up-to-date version, we talk of version jumps that need to be bridged.

Lazy composite migrations can be effective short-cuts in performing version jumps, as illustrated in Figure 5. In a first step, we compose the schema evolution operations; in a second step, a data migration operation is derived from the composite schema evolution operation. Thus, the composition at the level of schema evolution operations affects the migration operations at the level of entities as well: Instead of applying each operation one at-a-time, a single operation may be executed that yields the same result.

Let us consider a legacy entity \( e_{t,v} \), to be migrated by a chain of pending single-type migration operations into version \( v + x \):

\[
e_{t,v+x} \leftarrow \text{migrate}_{v+x}(\ldots(\text{migrate}_{v+1}(e_{t,v})))
\]

The benefit of applying lazy composite migration, rather than lazy stepwise migration, is twofold:

1) We only need to persist the latest version \( e_{t,v+x} \) in the data store. Any intermediate versions do not need to be stored. This simple optimization reduces the number of physical (and usually billable) database writes.

2) Sometimes, it is possible to compose several operations into a single operation, not even materializing intermediate versions in main memory. As an effect, we reduce the latency imposed by lazy migration.

With our rule-based composition that we introduce in Section IV-B, we combine schema evolution rules and write \( \text{evolve}_{i-k} \) to denote that the evolution operations \( i \) through \( k \) are combined. As an example, the composed operation \( \text{evolve}_{4-5} \) translates a schema from version 3 into version 5, and is defined as follows:

\[
\text{evolve}_{4-5}(S_3) = \text{evolve}_5(\text{evolve}_4(S_3))
\]

The composite operation generates the same result that we get if we successively execute each of the single-step schema evolution operations. Next, for each of these composed schema evolution operations, a matching data migration operation is derived. The operations \( \text{migrate}_{2-5} \),
Preconditions always refer to the current version, whereas we have omitted the version numbers from entity types. The sequence of two operations, each with pre- and postconditions, is shown in Figure 7: The numerator states a property in the source, is equivalent to performing a copy. However, first copying a property from the source to the target, and then deleting it at the side of the target, undoes the effect of the copy. However, first copying a property from the source to the target, and then deleting it at the side of the target, is equivalent to performing a move. Out of this reason, the copy operation is listed twice as op1 in Figure 6.

In addition to the rules listed, sequences of single-type evolution operations of the same kind can be composed. For instance, two add operations in sequence, on the same entity type, can be composed to a single composite add.

C. Algorithm for Optimizing Data Migrations

We sketch an algorithm for composing a chain of schema evolution operations. The evolution operations are managed in a list. Naturally, the order of the operations in the list determines the order of their application.

Two operations evolvei and evolvem can be composed if evolvei matches with op1 and evolvem matches with op2 in the table from Figure 6, and if there is no operation evolvek that operates on any of the entity types that occur in op1 or op2 and that is in the list between evolvei and evolvem.

If evolvei and evolvej are composed to evolvei−j, then operation evolvei is removed from the list and the operation evolvej is replaced by evolvei−j. This process continues until no further composition can be made.

migrate3−5, and migrate4−5 can be applied for migrating the entities from version 1, 2, and 3 into version 5.

A. Motivating Example

Before we state our composition rules, we provide an illustrative example. Let us consider a schema S1 with an entity type EPlayer. Two schema evolution operations are pending, namely the addition of a new property points, followed by renaming this property to score:

\[ \text{evolve_1: add Player.points = 42} \]
\[ \text{evolve_2: rename Player.points to score} \]

The operation evolve_2 delivers the schema in version 2, and evolve_3 produces the schema in version 3.

We employ a notation similar to the Hoare Calculus [10], where schema evolution operations carry pre- and postconditions. A pre condition prop1 \( \notin E_{t,v} \) declares that an operation can be applied if the property prop1 is not defined in an entity type in version v, whereas precondition prop2 \( \in E_{t,v} \) requires that the property prop2 has to be defined in the entity type in version v.

\[
\begin{align*}
\text{precond:} & \{ \text{points} \notin E_{\text{Player},1} \} \\
\text{evolve:} & \text{add Player.points = 42} \\
\text{postcond:} & \{ \text{points} \in E_{\text{Player},2} \} \\
\text{precond:} & \{ \text{points} \in E_{\text{Player},2} \land \text{score} \notin E_{\text{Player},2} \} \\
\text{evolve:} & \text{rename Player.points to score} \\
\text{postcond:} & \{ \text{points} \notin E_{\text{Player},3} \land \text{score} \in E_{\text{Player},3} \}
\end{align*}
\]

Combining the pre- and post-conditions, we derive the composition below:

\[
\begin{align*}
\text{precond:} & \{ \text{points} \notin E_{\text{Player},1} \land \text{score} \notin E_{\text{Player},1} \} \\
\text{postcond:} & \{ \text{points} \notin E_{\text{Player},1} \land \text{score} \in E_{\text{Player},3} \}
\end{align*}
\]

Composition of operations is possible in several cases. In the next section, we will define a set of composition rules.

B. Composition Rules

An example of a composition rule with its pre- and postconditions is shown in Figure 7: The numerator states a sequence of two operations, each with pre- and postconditions. The denominator states the composite operation, again with its pre- and postcondition. For better readability, we have omitted the version numbers from entity types. The preconditions always refer to the current version, whereas the postconditions are defined for the version that will be generated by the operation. Owing to space limitations, we use a tabular arrangement to more compactly present our composition rules in Figure 6, and refer to the unabridged rules in the Appendix in Figure 13.

Our rules are data store independent, and can thus be applied to different NoSQL data stores. They are further inspired by the rules introduced by Abiteboul et al. [11] for relational databases, and have been adapted and extended for the use case of NoSQL evolution operations.

The table in Figure 6 reads as follows. The operations are defined on entity types. To improve readability, we omit the version numbers of entity types. The first column shows the first schema evolution operation op1. As a succeeding operation, we choose operation op2 from table header row. The evolution operation obtained by composing op1 and op2 is stated in the corresponding table cell.

For instance, the first rule which is defined in the table states that “add E.B.y = default”, followed by “rename E.B.y to z”, can be replaced by the composite evolution operation “add E.B.z = default”.

The entry “\(|-|\)” declares that we cannot compose the operations, while “noop” declares that as an effect of composition, operations op1 and op2 cancel each other out. This holds for an add or copy, followed by a delete.

If op1 is a copy operation, we distinguish whether the succeeding operation op2 affects a property of the source entity type or a property of the target entity type. For instance, first copying a property from the source to the target, and then deleting it at the side of the target, undoes the effect of the copy. However, first copying a property from the source to the target, and then deleting it at the side of the source, is equivalent to performing a move. Out of this reason, the copy operation is listed twice as op1 in Figure 6.

In addition to the rules listed, sequences of single-type evolution operations of the same kind can be composed. For instance, two add operations in sequence, on the same entity type, can be composed to a single composite add.
D. Extended Example

We illustrate the composition rules using a more complex example, and continue with the gaming application, involving with Player and Mission entities. The property points has already been added to Player entities (see evolve2 in Section IV-A). We follow up with the next four operations. First, property points is renamed to score. Later, this property is copied from Players to their Missions. During development, the property is further renamed to amount within the Mission entity type. And finally, the property amount is yet again moved from the Mission to the Stats entities. Below, we show these schema evolution operations in sequence.

\[
\begin{align*}
\text{evolve}_3: \quad & \text{rename Player.points to score} \\
\text{evolve}_4: \quad & \text{copy Player.score to Mission.score} \\
\text{where Player.id = Mission.pid} \\
\text{evolve}_5: \quad & \text{rename Mission.score to amount} \\
\text{evolve}_6: \quad & \text{move Mission.amount to Stats.amount} \\
\text{where Mission.id = Stats.mid}
\end{align*}
\]

Let us assume that entities are migrated lazily. When the application in release 6 is about to load a Mission entity, the pending migration operations need to be applied. We discuss this scenario at the level of evolution operations: (1) To evolve from schema version 4, we can compose operations evolve5 and evolve6 to operation evolve5–6:

\[
\text{evolve}_{5-6}: \quad \text{move Mission.score to Stats.amount} \\
\text{where Mission.id = Stats.mid}
\]

(2) To evolve from schema version 2, the composition of evolve3 through evolve6 yields

\[
\begin{align*}
\text{evolve}_{3-6}: \quad & \text{copy Player.points to Stats.amount} \\
\text{where Player.id = Mission.pid} \\
\text{and Mission.id = Stats.mid}
\end{align*}
\]

Based on the composite evolution operations, we can derive migration operations accordingly. Assuming that each evolution operation matches one migration operation, we can now discuss the effectiveness of composition at the level of migrating individual entities: With lazy composite migration, we only execute one migration operation migrate\(_{5-6}\) instead of the two operations of lazy stepwise migration. For even longer version jumps, we can apply migrate\(_{3-6}\) to migrate entities from version 2 into version 6 with lazy composite migration. In comparison, lazy stepwise migration requires four operations.

V. EXPERIMENTAL EVALUATION

We experimentally evaluated implementations for carrying out lazy data migration with MongoDB. We conducted our experiments using our schema evolution middleware which actually supports a choice of NoSQL products.

Setup: We used MongoDB 3.2.1, Java version 1.8.0_74 with MongoDB Java Driver version 3.2.2, running on Ubuntu 14.04 LTS. MongoDB was used as-is, without any tuning. The experiments were run on a typical NoSQL commodity machine: The Dell PowerEdge C6220 has 2 Intel Xeon E5-2609 processors (4 cores each), 32 GB RAM, and 4 \times 1 \text{TB} \text{SATA} 7.2 \text{k HDs}.

Again, we refer to the scenario of players and their missions. Each Mission is currently played by exactly one Player, and each Player is pursuing 50 Missions on average. In the following, we focus on two dimensions of data migration in particular, namely operation execution and location (c.f. Section II).

A. Impact of Operation Execution

Over five releases, properties are added to Players. We compare two migrations strategies using MongoDB update operations: In lazy stepwise migration, each add operation is executed individually, in sequence. In lazy composite migration, five migration operations are composed into a single operation. Figure 8(a) shows the accumulated runtime increasing with the number of Player entities processed. The
factor between both approaches increases from 1.08 to 4.51, and converges to the theoretical limit of 5.

Figure 9 visualizes the throughput for this experiment. Besides the performance benefit of composition, we see the advantage of migrating more than one entity at-a-time, i.e., migrating groups of entities in bulk. This is an argument in favor of predictive migration, as discussed later in Section VI-C.

B. Impact of Location

In the previous experiment, the execution of migration operations was delegated to the database using the MongoDB update language (termed “update based” in Figures 8 and 9). However, NoSQL data stores differ in their update operations. Therefore, it may not always be possible to capture the migration operations within the update language, especially in the case of copy and move operations.

Let us consider a concrete example, where a property is moved from each Player to the corresponding 50 Mission entities. We compare three location alternatives in Figure 8(b): One option is to carry out migration within the application code. This implementation method can always be chosen, yet not surprisingly, is expensive even for considerably small inputs. A second option is to use update operations provided by the database (as done in Figure 8(a)). This approach is more efficient. However, depending on the expressiveness of the update API (often, lacking joins), it may nevertheless be necessary to carry out parts of the migration within the application code. Thus, this approach is not always appropriate when facing large volumes of legacy data. The third approach examined uses a “custom coded” MongoDB JavaScript function, comparable to a database stored procedure. This is the most efficient approach w.r.t. the runtime overhead in our lineup.

VI. SIMULATION-BASED ANALYSIS

We continue with an analysis based on simulations, where we contrast different migration strategies.

A. Eager vs. Lazy Forms of Migration

We next compare eager migration, lazy stepwise migration, and lazy composite migration.

We assume a sample application, starting with a data instance of 100 million entities in version 1. We apply four evolution operations, that successively transform the schema into version 2, 3, 4, and finally 5. We further assume conservatively that from each schema evolution operation, exactly one migration operation is derived, which then writes one entity in the NoSQL database. In order to make our results comprehensible, we make the simplistic assumption that no entities are added by later releases.

Eager Migration: Eager migration affects all legacy entities and migrates them into the next version. Figure 10(a) visualizes the effect over time. The horizontal axis facing the front shows five schema versions (labeled “schema releases”), and represents the progress of releases over time. The second horizontal axis (labeled “entity versions”) captures which schema version the entities in the database actually comply with. The vertical axis shows the number of entities (in millions) in the database, at the time of a given schema release, that are in a given entity version.

The initial release declares schema version 1. Accordingly, all 100 million entities are in version 1 as well.
the schema version changes to 2, all entities are eagerly migrated into entity version 2. This pattern carries forward to all schema releases. Obviously, with eager migration, all entities always match the schema of the current release.

Figure 10(d) shows the accumulated number of migration operations in our simulation. Each operation updates one entity. At the time of schema release 5, 100 million migration operations per schema release have been carried out. We would like to point out that in the current version of the cloud-hosted database Google Cloud Datastore, performing a total of 400 million writes costs up to 720 USD [1]. With weekly and possibly daily releases, this is a considerable burden on the long-term operational costs.

Lazy Stepwise Migration: In lazy migration, we only migrate entities on demand, upon access by the application. For our calculation, we assume that in each schema release, 25% of the entities are accessed and therefore migrated. In our simulation, we assume a uniform distribution of entity access. We apply this distribution because it is very easy to calculate and an estimation of the worst case for lazy migration. With other access distributions, for instance the normal distribution, the number of migration operations in lazy migration is even lower.

Initially, all entities are in version 1. This is captured in Figure 10(b), where at the time of schema release 1, all 100 million entities belong to entity version 1. With schema release 2, 25 million entities are migrated to this next version, while the majority of entities remains in version 1. With schema release 3, 25 million entities are migrated lazily into version 3, originating from versions 1 and 2. At the time of schema release 5, the data store contains entities in five schema versions.

The migration of entities from version 2 into 3 can be done with one migration operation whereas the migration of entities from version 1 into version 3 requires applying two migration operations. The accumulated number of migration operations executed is shown in Figure 10(e). Overall, we execute fewer migration operations than with eager migration (c.f. Figure 10(d)), since entities that are not accessed by the application are not migrated.

The migration effort rises with each schema release: With advanced schema releases, the length of the version jumps tends to increase. This drives up the number of migration operations with lazy stepwise migration, causing the slope of the curve to increase.

Lazy Composite Migration: We next focus on the effect of composing migrations, as proposed in Section IV. Figure 10(c) visualizes the distribution of entities in different versions as the application is repeatedly released. It is noteworthy that the distribution is identical to that of lazy stepwise migration, as shown in Figure 10(b).

The difference between the approaches becomes evident when we consider the number of migration operations executed: As may be expected, the accumulated number of migrations, shown in Figure 10(f), is persistently lower than with lazy stepwise migration, shown in Figure 10(e).

B. Cascading Migration Operations

For the calculations in the previous section, we assumed single-type operations. With multi-type operations, the cost model is more involved. The reason are cascading migration operations. Let us reconsider the example from Section IV-D. When loading a Stats entity in the current version 6, it may also be necessary to migrate Missions, because both are joined in the move evolution operation evolve6. Furthermore, this can also trigger the migration of Player
entities \(evolve_4\) and \(evolve_5\). To counter such cascading
effects, we next introduce hybrid migrations strategies.

C. Blending Lazy and Eager Migration

Hybrid migration strategies blend concepts from lazy and
eager migration. They aim at reducing cascading migration
operations and therefore also the latency overhead.

**Incremental Migration:** In incremental data migration,
we handle some releases with lazy migration, and others
with eager migration. This choice can be made depending
on the disruptiveness of a schema change (e.g., performing
simple changes lazily), and the heterogeneity of legacy data
in the production database. In our experience, this approach
best matches how development teams proceed in practice.

Figure 11(a) shows a concrete scenario. For the first five
schema releases, lazy stepwise migration is applied. By then,
the database contains different versions of legacy entities,
and is therefore quite heterogeneous. Up to version 5, this
strategy requires the same number of migration operations
as lazy migration (see Figure 10(e)). To transfer the data
instance into a structurally homogeneous state, the migration
into version 6 is done eagerly. The resulting data instance
may be seen as a schema-consistent snapshot, from which
we again continue with lazy migration. In Figure 11(a), we
therefore see a distinguished, single spike at the time of
schema release 6, when all entities are in version 6. By the
time of schema release 8, the production database contains
entities in three versions.

The accumulated number of migration operations is
shown in Figure 11(b). Again, we see an increase between
schema releases 5 and 6, caused by the effort of eager
migration. For the succeeding schema releases, the gradient
again decreases, as lazy migration needs to span versions
jumps of length at most 2.

**Predictive migration:** An inherent downside of lazy
migration is the latency overhead when legacy entities are
accessed by the application. To reduce the average latency,
we employ predictive migration, and proactively migrate
legacy entities that are likely to be accessed by the applica-
tion in the near future. We can carry out predictive migration
in the background, concurrently to lazy migration.

In Figure 12, we simulate this strategy for our running
example. We predict 25 million entities and migrate them
into the latest version. Whereas the development and val-
idation of a forecast function is part of our future work,
we assume for the simulation that in 50% of cases, the
function delivers correct predictions. A prediction may turn
out to be incorrect, because this function uses heuristics.
Figure 12(a) contrasts lazy stepwise migration, as discussed
earlier, with the predictive approach: It shows the distribu-
tion of entity versions at the time of release 5. Predictive
migration increases the number of migration operations (see
Figure 12(b)), but decreases the average latency, since more
entities reside in younger schema versions. Another advan-
tage of predictive migration is that the update operations
can be executed in bulk (as discussed in Section V).

Whether predictive migration is combined with lazy step-
wise migration, lazy composite migration, or incremental
migration, it can effectively reduce latency, provided that a suitable prediction function is employed.

VII. RELATED WORK

For eagerly evolving the schema in relational or NoSQL data stores, the developer community has produced a range of practical tools: For instance, Flyway and Liquibase (for relational databases), or Mongeez (for MongoDB) trigger easier migration at application startup. Eager migration requires careful management, to avoid application downtime. Google’s data store F1 is one of the few systems custom-tailored to avoid downtimes in eager migration [12]. However, eager migration may be costly, causing high numbers of billable reads and writes against a cloud-hosted database. In such settings, lazy migration may be a preferable strategy.

A first set of experiments studying the performance overhead of lazy migrations in NoSQL data stores can be found in [13]. There, it is shown that the overhead of lazy migration is affordable, when one single-step evolution operation is applied (i.e., an add, rename, or delete). Our work goes further and considers composite migrations, executing several pending schema changes for a legacy entity in one go, rather than one after the other. Thus, we can effectively bridge what we call version jumps.

We would like to point out that composition in the context of lazy migration is, to our knowledge, a novel twist to an established research area: In database theory, the composition of eager migrations (or schema mappings) in the relational domain is a well-studied topic. We refer to [14] for a fundamental introduction. In the context of lazy migration, however, legacy entities in different schema versions may co-exist in the same data store.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we examined strategies for scalable long-term lazy schema evolution in NoSQL data stores (lazy stepwise, lazy composed, predictive, and incremental data migration). In particular, we proposed a rule-based composition for chains of pending schema evolution operations. Our experiments and our simulation-based analysis show that we can effectively reduce the number of migration operations. This immediately manifests in fewer (billable) database writes and is particularly interesting in setups where the database is provided as-a-service. We further analyzed hybrid migration strategies that blend release rounds using lazy migration with rounds using eager migration.

One of our future tasks is to develop a holistic cost model that captures the tradeoffs involved in choosing a migration strategy. The cost model is a basis for deciding how schema evolution and data migration for a given application are best realized. A major challenge here is to capture the risk involved with migrating legacy entities: From the viewpoint of the DevOps team, it is desirable to deploy continuously. This speaks in favor of migrating legacy entities lazily, rather than eagerly (since the latter is challenging to do with zero downtime). At the same time, when lazy migration has to bridge version jumps, the latency imposed by lazy migration can be an incalculable risk to the smooth operation of the application. Having a holistic cost model would allow a proper risk assessment, which is preferable to the status quo, where decisions about the schema evolution strategy merely rely on the experience of the DevOps team.

We regard the work presented in this paper as a foundation for crafting a comprehensive cost model.

REFERENCES


add-rename \{x \not\in E_A\} add A.x = \text{default} \{x \in E_A\}, \{(x \in E_A) \land (y \not\in E_A)\} rename E_A.x to y \{(x \not\in E_A) \land (y \in E_A)\}
\{(x \not\in E_A) \land (y \not\in E_A)\} add A.y = \text{default} \{(x \not\in E_A) \land (y \in E_A)\}

add-move \{x \not\in E_A\} add A.x = \text{default} \{x \in E_A\}, \{(x \in E_A) \land (y \not\in E_B)\} move E_A.x to E_B.y cond \{(x \not\in E_A) \land (y \in E_B)\}
\{(x \not\in E_A) \land (y \not\in E_B)\} add E_B.y = \text{default} cond \{(x \not\in E_A) \land (y \in E_B)\}

add-delete \{x \not\in E_A\} add A.x = \text{default} \{x \in E_A\}, \{x \in E_A\} delete E_A.x \{x \not\in E_A\}

rename-rename \{(x \in E_A) \land (y \not\in E_A)\} rename E_A.x to y \{(x \not\in E_A) \land (y \in E_A)\}, \{(y \in E_A) \land (z \not\in E_A)\} rename E_A.y to z \{(y \not\in E_A) \land (z \in E_A)\}
\{(x \in E_A) \land (y \not\in E_A) \land (z \not\in E_A)\} rename E_A.x to E_A.z \{(x \not\in E_A) \land (y \not\in E_A) \land (z \in E_A)\}

rename-copy \{(x \in E_A) \land (y \not\in E_A)\} rename E_A.x to y \{(x \not\in E_A) \land (y \in E_A)\}, \{(y \in E_A) \land (z \not\in E_A)\} copy E_A.y to E_B.z cond \{(y \not\in E_A) \land (z \in E_B)\}
\{(x \in E_A) \land (y \not\in E_A) \land (z \not\in E_B)\} copy E_A.x to E_B.z cond \{(x \not\in E_A) \land (y \not\in E_A) \land (z \in E_B)\}

rename-move \{(x \in E_A) \land (y \not\in E_A)\} rename E_A.x to y \{(x \not\in E_A) \land (y \in E_A)\}, \{(y \in E_A) \land (z \not\in E_A)\} move E_A.y to E_B.z cond \{(y \not\in E_A) \land (z \in E_B)\}
\{(x \in E_A) \land (y \not\in E_A) \land (z \not\in E_B)\} move E_A.x to E_B.z cond \{(x \not\in E_A) \land (y \not\in E_A) \land (z \in E_B)\}

rename-delete \{(x \in E_A) \land (y \not\in E_A)\} rename E_A.x to y \{(x \not\in E_A) \land (y \in E_A)\}, \{y \in E_A\} delete E_A.y \{y \not\in E_A\}

copy-rename \{(x \in E_A) \land (y \not\in E_B)\} copy E_A.x to E_B.y cond \{(x \in E_A) \land (y \in E_B)\}, \{(y \in E_B) \land (z \not\in E_B)\} rename E_B.y to z \{(y \not\in E_B) \land (z \in E_B)\}
\{(x \in E_A) \land (y \not\in E_B) \land (z \not\in E_B)\} copy E_A.x to E_B.z cond \{(x \not\in E_A) \land (y \not\in E_B) \land (z \in E_B)\}

copy-move \{(x \in E_A) \land (y \not\in E_B)\} copy E_A.x to E_B.y cond \{(x \in E_A) \land (y \in E_B)\}, \{(y \in E_B) \land (z \not\in E_C)\} move E_B.y to E_C.z cond \{(y \not\in E_B) \land (z \in E_C)\}
\{(x \in E_A) \land (y \not\in E_B) \land (z \not\in E_C)\} copy E_A.x to E_C.z cond \{(x \not\in E_A) \land (y \not\in E_B) \land (z \in E_C)\}

move-rename \{(x \in E_A) \land (y \not\in E_B)\} move E_A.x to E_B.y cond \{(x \not\in E_A) \land (y \in E_B)\}, \{(y \in E_B) \land (z \not\in E_B)\} rename E_B.y to z \{(y \not\in E_B) \land (z \in E_B)\}
\{(x \in E_A) \land (y \not\in E_B) \land (z \not\in E_B)\} move E_A.x to E_B.z cond \{(x \not\in E_A) \land (y \not\in E_B) \land (z \in E_B)\}

move-move \{(x \in E_A) \land (y \not\in E_B)\} move E_A.x to E_B.y cond \{(x \not\in E_A) \land (y \in E_B)\}, \{(y \in E_B) \land (z \not\in E_C)\} move E_B.y to E_C.z cond \{(y \not\in E_B) \land (z \in E_C)\}
\{(x \in E_A) \land (y \not\in E_B) \land (z \not\in E_C)\} move E_A.x to E_C.z cond \{(x \not\in E_A) \land (y \not\in E_B) \land (z \in E_C)\}

move-delete \{(x \in E_A) \land (y \not\in E_B)\} move E_A.x to E_B.y cond \{(x \not\in E_A) \land (y \in E_B)\}, \{y \in E_B\} delete E_B.y \{y \not\in E_B\}
\{(x \in E_A) \land (y \not\in E_B)\} delete E_A.x \{(x \not\in E_A) \land (y \not\in E_B)\}

Figure 13. Composition rules for two evolution operations (for simplicity, version subscripts have been omitted from entity types).