A Datalog-Based Protocol for Lazy Data Migration in Agile NoSQL Application Development

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

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

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

Abstract

We address a practical challenge in agile web development against NoSQL data stores: Upon a new release of the web application, entities already persisted in production no longer match the application code. Rather than migrating all legacy entities *eagerly* (prior to the release) and at the cost of application downtime, *lazy* data migration is a popular alternative: When a legacy entity is loaded by the application, all pending structural changes are applied. Yet correctly migrating legacy data from several releases back, involving more than one entity at-a-time, is not trivial. In this paper, we propose a holistic $Datalog^{\neg}_{non\text{-}rec}$ model for reading, writing, and migrating data. In implementing our model, we may blend established Datalog evaluation algorithms, such as an incremental evaluation with certain rules evaluated bottom-up, and certain rules evaluated top-down with sideways information passing. Our systematic approach guarantees that from the viewpoint of the application, it remains transparent whether data is migrated eagerly or lazily.

Categories and Subject Descriptors H.2.3 [*Database Management*]: Languages

Keywords Schema evolution, NoSQL data stores, Datalog

1. Introduction

In agile web development, NoSQL data stores have become quite popular. Consequently, virtually all major players in the cloud market have a NoSQL data store in their portfolio, e.g., Google (Datastore), IBM (Cloudant), Amazon (SimpleDB), or Microsoft (DocumentDB). Additionally, several open-source products, such as MongoDB or CouchDB, are available. What makes NoSQL data stores attractive is their ability to scale to large volumes of data, and equally, to store data of heterogeneous structure. This allows for frequent releases, whereas a relational database commonly requires an eager migration of all legacy data prior to the next release.

With NoSQL data stores, developers prefer to migrate legacy entities *lazily*, when they are next accessed by the application. Several NoSQL object mapper libraries natively support lazy migration [8]. While a convenient short-term fix, it is rather easy to specify conflicting migration operations in object mapper class declarations, even for very basic changes such as adding, removing, or renaming a property [7]. Moreover, migrations that involve more than one entity at-a-time, or require upgrades from several releases back, increase the challenge. Let us illustrate this point with an example.

EXAMPLE 1. We consider a gaming app. Figure 1 shows the contents of the NoSQL data store over time. Above the time line, we see the actions triggering changes. Below, we see the persisted entities. Entities with continuous borders are persisted eagerly, whereas entities with dashed borders will be derived lazily.

All entities persisted by the first release adhere to the same structure. Typically, this “schema” is imposed by the application, often using dedicated NoSQL object mapper libraries. At times *ts*1 through *ts*5, player and mission entities are persisted. Each player has an identifier and a name. Each mission also has an identifier, a title, and a reference to the player currently pursuing this mission. Internal to the data store, we timestamp all entities.

At time *ts*6, a new release is deployed. From now on, all player entities also carry a property score, initialized to 50. However, we do not migrate the legacy entities eagerly, to avoid downtime. Instead, we exploit the schema flexibility of the NoSQL data store, and store both versions of entities for now.

At timestamp *ts*7, Lisa’s player is updated by a put-call. Several NoSQL data stores follow an append strategy, where an update results in an additional, timestamped entity. At timestamp *ts*8, we yet again release a new version of the application where missions carry their player’s score. Again, we do not immediately migrate the legacy entities. At time *ts*9, the application requests the mission with id 100. Since the mission entity with timestamp *ts*5 is outdated, we need to lazily migrate it. Copying Lisa’s score to her mission entity generates the up-to-date entity.

At *ts*10, the application requests mission 101. In migrating mission 100, we applied the pending copy operation. Yet with mission 101, this is not enough, since Bart’s player does not even carry a score yet. Executing the copy alone yields the incorrect entity derived by the dashed arrows. Instead, we need to proceed as shown by the continuous arrows, and apply *two* pending schema changes: We add the score to player Bart before we copy it to his mission. □

Contributions: In this paper, we propose a Datalog-based model:

1. Our model formalizes all structural variants of persisted entities in non-recursive Datalog with negation. In particular, we capture strongly consistent writes and reads via put- and get-calls. We further model a core set of migration operations for adding, removing, and renaming, as well as copying and moving properties, originally proposed by us in [6].
2. We discuss alternative evaluation strategies for our Datalog programs. In any case, rules capturing put- and get-calls from

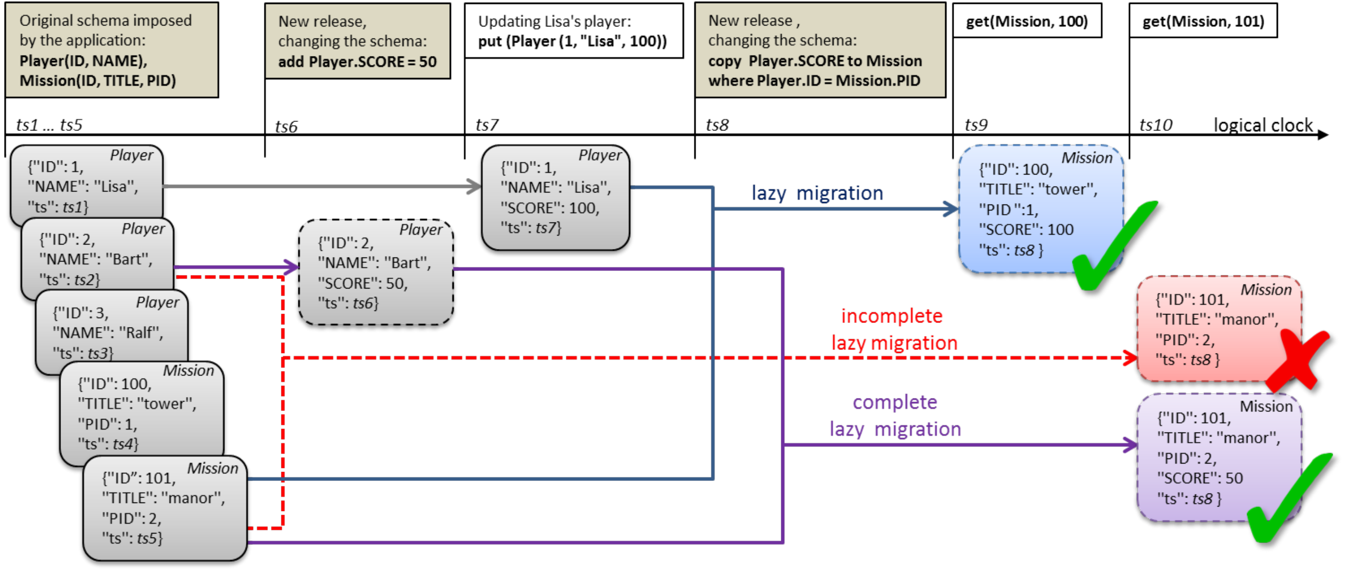

Figure 1. The state of a NoSQL data store over time. At times $ts9$ and $ts10$, the mission entities are loaded and migrated lazily.

the application are carried out eagerly. However, rules capturing migration can be evaluated eagerly or lazily.

3. We leverage the rich body of work on Datalog evaluation algorithms and propose a lazy migration protocol where all get-calls always return up-to-date results. Thus, we may safely employ lazy migration in NoSQL application development, and deploy new releases without worrying about downtimes.

Selected related work: Datalog is a state-of-the-art formalism for data exchange [2]. It is only natural to use it in the context of schema evolution. In this paper, we not only declare schema mappings, but also updates in Datalog. There is considerable work proposing updates to rule-based languages, with little consensus reached [5]. Since we only insert singletons (versus set-oriented updates), updates are less complex in our use case.

There is a large variety of Datalog evaluation algorithms, with top-down, bottom-up, and the magic set algorithm being the most well-known [9]. Incremental evaluation of Datalog queries in the presence of updates has also been a timeless research topic. We refer to [3] for a comprehensive overview over related work.

2. A Holistic Datalog-Based Model

In modelling our use case in Datalog, we disregard certain peculiarities of working with JSON data (such as its hierarchical structure and multi-valued entities[1]), since our main focus is on the semantics and order of data migration. We largely follow the Datalog syntax introduced in [9], capitalizing variables in Datalog rules. The underscore denotes anonymous variables.

2.1 Puts and Gets against NoSQL Data Stores

We model the application accessing entities in a straightforward manner. Let $kind(ID, P_1, \ldots, P_n)$ be the current schema for a given kind, as declared by an object mapper class declaration of the current application release: Each entity of this kind has a unique ID, and properties named $P_1, \ldots, P_n$. We persist an entity via a method call $put(kind(id, p_1, \ldots, p_n))$, and we load an entity given its identifier via a call *get(kind, id)*. To keep our model lean, we do not look into deletion or queries other than calls to get, which returns a singleton. We assume strict consistency for puts and gets, as implemented by some (but not all) NoSQL data stores (e.g., [4]).

Internally, we timestamp all entities. Our timestamp simulates a logical clock, and tracks the last modification under the assumption that all actions are carried out eagerly.

EXAMPLE 2. We assume the current application release declares the schema $Player(ID, NAME)$. We further consider the database state entered by executing the action sequence

```
a1: put(Player(1, "Lisa"));
a2: put(Player(1, "Lisa S."));
a3: get("Player", 1);
```

i.e., persisting Lisa's player, updating it, and retrieving it back.

We model the put-calls by EDB predicates added to the database. Below, `ts1` and `ts2` denote fresh timestamps.

```
r1: Player(1, "Lisa", ts1).
r2: Player(1, "Lisa S.", ts2).
```

We also model the get-call by Datalog rules, fetching back the latest instance of the entity (i.e., that with the latest timestamp). Self evidently, in a performant implementation, get-calls would be supported by an index to efficiently retrieve the latest instance.

```
r3: legacyPlayer(ID, TS) :-
      Player(ID, _, TS), Player(ID, _, NTS), TS < NTS.
r4: latestPlayer(ID, TS) :-
      Player(ID, _, TS), not legacyPlayer(ID, TS).
r5: getPlayer(ID, NAME)  :-
      Player(ID, NAME, TS), latestPlayer(ID, TS).
```

Evaluating query predicate `getPlayer(1, NAME)` yields the up-to-date property values of Lisa's player. □

In maintaining the state of the data store incrementally, we distinguish two kinds of Datalog rules: Facts derived from *residual rules* model lasting changes, such as writing an entity. A fact derived from a residual rule will hold true even in the presence of future actions. Since residual rules declare a monotonously growing set of facts, we may re-use these facts in future computations. *Transient rules* compute intermittent facts only required for evaluating the

[1] Mapping such data formats to a relational model has been studied extensively for XML. For our purposes, we simply "flatten" JSON documents.

Let $kind[r](ID, P_1, ..., P_n)$ be the schema imposed by the current application release. *ts* denotes a fresh timestamp associated with release r.

i) **add** $kind.P_{n+1} = v$, where P_{n+1} is a new property name and v is a default value (in the new version of the entity, P_{n+1} has value v):

```
kind[r+1](ID, P1,...,Pn, v, ts)    :- kind[r](ID, P1,...,Pn, OTS), latestkind[r](ID, OTS).
```

ii) **delete** $kind.P_i$

```
kind[r+1](ID, P1,...,P(i-1),P(i+1),...,Pn, ts) :- kind[r](ID, P1,...,Pn, OTS), latestkind[r](ID, OTS).
```

Let $kindS[r](ID, S_1, ..., S_n)$ and $kindT[r](ID, T_1, ..., T_m)$ be the current source and target schema imposed by the application.

iii) **copy** $kindS.S_i$ **to** *kindT* where $kindS.ID = kindT.T_j$

```
kindT[r+1](ID_T, T1,...,Tm, Si, ts) :- kindT[r](ID_T, T1,...,Tm, TS_T), latestkindT[r](ID_T, TS_T),
                                       kindS[r](ID_S, S1,...,Sn, TS_S), latestkindS[r](ID_S, TS_S), ID_S = Tj.
kindT[r+1](ID_T, T1,...,Tm, null, ts) :- kindT[r](ID_T, T1,...,Tm, TS_T), latestkindT[r](ID_T, TS_T),
                                       not kindS[r](ID_S, S1,...,Sn, TS_S), ID_S = Tj.
```

iv) **move** $kindS.S_i$ **to** *kindT* where $kindS.ID = kindT.T_j$, with the same rules as for copy, as well as the following rule:

```
kindS[r+1](ID, S1,...,S(i-1),S(i+1),...,Sn, ts) :- kindS[r](ID, S1,...,Sn, OTS), latestkind[r](ID, OTS).
```

Figure 2. Residual rules for migrations declared in the schema evolution language from [6].

query predicate. Facts derived from transient rules may not hold true in the presence of future changes, and can be timely discarded.

EXAMPLE 3. In the previous example, `r3` is a residual rule: Once the initial player entity has been overwritten, it remains a legacy entity. The transient rules `r4` and `r5` merely assist in evaluating the query predicate. Yet if Lisa's player is updated later on, the facts derived from these rules are no longer valid. □

Initially, the data store is empty. Let $a_1, \dots, a_n$ be an action sequence. For each action a_i we compile an *action tuple*

$$AT(a_i) = (\Delta_i, R_i, T_i, p_i)$$

where Δ_i is a set of EDB facts (i.e., entities put to the data store), R_i and T_i are residual and transient Datalog rules, and p_i is a query predicate (empty for put-calls).

To capture the effect of action a_n, we evaluate a Datalog query $Q_n = (\Pi_n, p_n)$ where the Datalog program Π_n consists of the rules $R_1 \cup \dots \cup R_n \cup T_n$, and the data instance D_n consists of all entities put to the data store, i.e., $D_n = \Delta_1 \cup \dots \cup \Delta_n$. Then the result $\Pi_n(D_n)$ of applying Datalog program Π_n to D_n is the set of IDB facts that are logical consequences of $\Pi_n \cup D_n$.

Since we evaluate non-recursive Datalog rules with negation, we may compute the IDB relations in the order of their dependencies [9]. This is our basis for maintaining the database state incrementally. The function $ResDB(a)$ declares the computation of the residual entities after executing an action sequence $a_1, \dots, a_n$:

$$ResDB(a_1) = R_1(\Delta_1)$$
$$ResDB(a_n) = R_n(\ ResDB(a_{n-1}) \cup \Delta_n\ )$$

Instead of evaluating all residual rules from scratch, we incrementally build upon the facts already derived from residual rules:

$$\Pi_n(D_n) = T_n(ResDB(a_n)).$$

We next specify how to compile action tuples for puts and gets.

Put-calls: Let $kind(ID, P_1, \dots, P_n)$ be the schema currently imposed by the application. We consider a put-call

$$put(kind(id, p_1, \dots, p_n));$$

Let *ts* be a fresh timestamp, then we generate an action tuple where the residual rule tracks which entities have become legacy entities:

```
Δ = { kind(id, p1,...,pn, ts) },
R = { legacykind(ID, TS) :- kind(ID, P1,...,Pn, TS),
        kind(ID, S1,...,Sn, NTS), TS < NTS. },
```

and further, $T = \emptyset$, and p being the empty query.

Get-calls: Next, we consider a get-call requesting an entity,

$$kind(id, p_1, \dots, p_n) := get(kind, id);$$

Let *ts* be a fresh timestamp, then we generate an action tuple with $\Delta = \emptyset$, and further

```
R = { legacykind(ID, TS) :- kind(ID, P1,...,Pn, TS),
        kind(ID, S1,...,Sn, NTS), TS < NTS.}
T = { latestkind(ID, TS) :-
        kind(ID, P1,..., Pn, TS), not legacykind(ID, TS).,
      getkind(ID, P1,...,Pn) :-
        kind(ID, P1,..., Pn, TS), latestkind(ID, TS). }
p = getkind(id, P1,...,Pn)
```

EXAMPLE 4. Example 2 shows the rules derived from the action sequence `a1` through `a3`. □

In compiling action tuples, certain rules may be generated multiple times for different actions. Since these rules will be identical, it is merely a syntactical artifact of our approach.

2.2 Migrating Legacy Entities

We consider a small, general-purpose set of schema evolution operations, first proposed in [6], containing add, rename, remove, copy, and move operations. We believe the given operations cover the bulk of practical use cases. In particular, copying properties (thus denormalizing entities), is essential in NoSQL data stores, where joins are often not supported. We can easily support additional operations, as long as they can be expressed in our Datalog fragment.

Schema evolution operations also modify the database state. We denote the current schema of application release r for a given kind by $kind[r](ID, P_1, \dots, P_n)$. The next release $r+1$ introduces a new schema version. With each new schema version $kind[r+1](ID, P_1, \dots, P_m)$, we always compile the residual rules shown below. We do so for each kind occurring in the data store. These rules ensure that only the values from the latest instance of an entity are migrated. Note that these rules are monotonous: Once the new schema version has been introduced, any entities persisted by future actions will match a new schema. Therefore, facts derived from the rules below remain true even in the presence of future puts.

```
legacykind[r](ID, TS) :-
    kind[r](ID, P1,...,Pn, TS),
    kind[r](ID, S1,...,Sn, NTS), TS < NTS.
latestkind[r](ID, TS) :-
    kind[r](ID, P1,..., Pn, TS), not legacykind[r](ID, TS).
```

For each new release, we also declare a new schema version for all other kinds. So for each $kind'[r](ID, A_1, \dots, A_k)$ with entities

```
// add Player.SCORE = 50;
Player2(ID, NAME, 50, ts6) :-
  Player1(ID, NAME, OTS), latestPlayer1(ID, OTS).
Mission2(ID, TITLE, PID, ts6) :-
  Mission1(ID, TITLE, PID, OTS), latestMission1(ID, OTS).

// copy Player.SCORE to Mission where Player.ID = Mission.PID
Mission3(ID, TITLE, PID, SCORE, ts8) :-
  Mission2(ID_T, TITLE, PID, TS_T), latestMission2(ID_T, TS_T),
  Player2(ID_S, NAME, SCORE, TS_S), latestPlayer2(ID_S, TS_S),
  ID_S = PID.
Player3(ID, NAME, SCORE, ts8) :-
  Player2(ID, NAME, SCORE, OTS), latestPlayer2(ID, OTS).
```

Figure 3. (Selected) residual rules for our running example.

persisted in production that are *not* affected by a schema change, we declare the following residual rule. Again, *ts* is a fresh timestamp, associated with release r.

kind'$[r+1]$`(ID, A1, ..., Ak,` *ts*`) :-`
kind'$[r]$`(ID, A1, ..., Ak, OTS), latest`*kind'*$[r]$`(ID, OTS).`

This greatly simplifies our Datalog rules. Naturally, in a practical implementation, we optimize away such ineffective rules.

Figure 2 shows the residual rules for our schema evolution operations. The rules for adding and removing properties are straightforward. We omit the rule for renaming, since it is trivial. In copying and moving properties, we always assume a 1:N relationship between the source and the target kind (c.f. our discussion of safe migrations in [6]). Moreover, the residual rules declare an outer-join semantics for target entities without a matching source entity.[2]

EXAMPLE 5. For our scenario from Figure 1, Figure 3 shows (a subset of) the rules for the migrations between releases. □

3. Data Migration Protocols

There is a variety of Datalog evaluation algorithms, with the most prominent being standard textbook material [9]. We are interested in execution protocols where put- and get-calls are evaluated eagerly and schema migrations are either eager or lazy. We consider lazy migration correct if upon accessing an entity via a get-call, we obtain the same entity as with eager migration. Then then the migration strategy remains transparent to the application.

We assume that put- and get-calls are always evaluated eagerly, and therefore in an incremental bottom-up approach. Since we only generate non-recursive Datalog with negation, $Datalog^{\neg}_{non\text{-}rec}$, we may simply evaluate rules bottom-up, in the order of their dependencies, or even simpler, the timestamps assigned to them. Due to the monotonicity of residual rules we may maintain derived facts incrementally, in the spirit of semi-naive Datalog evaluation.

Eager migration: Since the residual rules for data migration are within $Datalog^{\neg}_{non\text{-}rec}$, they may also be evaluated bottom-up and incrementally. As a straightforward optimization, we can timely discard legacy entities: Given two versions of an entity $kind[i](id, p_1, \ldots, p_n, ts_i)$ and $kind[j](id, s_1, \ldots, s_m, ts_j)$, we may discard the former if $ts_i < ts_j$. In fact, many prominent NoSQL data stores follow an append-approach with put-calls and timestamp all entities. A data store-internal garbage collection mechanism then discards legacy entities no longer needed (e.g. [1]).

Lazy migration: Lazy migration is triggered by get-calls, and evaluated top-down, deriving only the necessary intermittent facts from residual rules. We intend to blend incremental bottom-up evaluation for put-calls with top-down evaluation for get-calls, employing sideways information passing.

EXAMPLE 6. We again consider our showcase. Given the residual rules from Figure 3, the entities timestamped *ts*6 and *ts*8 in Figure 1 (as well as some additional, purely auxiliary facts) are derived lazily upon the get-call at time *ts*10. □

In implementing lazy migration, we need to hold on to legacy entities in several versions until they are no longer needed. For instance, in Figure 1, the mission entity with timestamp *ts*5 may only be discarded once lazy migration has produced its most up-to-date version (timestamped *ts*8). Thus, a special-purpose "garbage collection" mechanism is required. This will need to hold on to residual rules until they, too, cannot fire anymore, since all matching entities have meanwhile been garbage collected.

4. Outlook on Future Work

This article introduces a formal model for migrating legacy data between software releases against NoSQL data stores. To our knowledge, this is the first systematic approach to a relevant practical problem where only shirt-sleeve solutions exist today.

Our Datalog rules not only define clean semantics for data migration, but also come with a variety of correct evaluation algorithms: Whether we evaluate rules eagerly (bottom up), lazily (top-down), or blend both approaches (with eager puts and gets, and lazy migrations), we are guaranteed to obtain a correct result. Our immediate next step is the detailed specification of our migration protocol and a formal proof of its correctness.

We are currently integrating our protocol into a real-life NoSQL data store as part of our schema management component. This requires extensions to more complex NoSQL data models. In particular, a practical implementation will need to bridge the impedance-mismatch between hierarchical data format persisted in NoSQL data stores and the Datalog rules that identify properties by index position. Further, we will extend to access operations beyond put- and get-calls, and consider simple queries. Another task is to devise an efficient garbage collection mechanism for pruning legacy entities from the NoSQL data store.

References

[1] J. Baker, C. Bondç, J. C. Corbett, J. J. Furman, et al. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proc. CIDR'11*, 2011.

[2] A. Doan, A. Halevy, and Z. Ives. *Principles of Data Integration*. Morgan Kaufmann Publishers Inc., 1st edition, 2012.

[3] G. Dong and J. Su. First-Order Incremental Evaluation of Datalog Queries. In *DBPL'93*, 1993.

[4] Google Inc. Google Cloud Datastore, Aug. 2015. `https://developers.google.com/appengine/docs/java/datastore/`.

[5] M. Liu. Extending Datalog with Declarative Updates. In M. Ibrahim, J. Köng, and N. Revell, editors, *Database and Expert Systems Applications*, volume 1873 of *LNCS*. Springer, 2000.

[6] S. Scherzinger, M. Klettke, and U. Störl. Managing Schema Evolution in NoSQL Data Stores. In *Proc. DBPL'13*, 2013.

[7] S. Scherzinger, T. Cerqueus, and E. Cunha de Almeida. ControVol: A Framework for Controlled Schema Evolution in NoSQL Application Development. In *Proc. ICDE'15*, 2015. Demo Paper.

[8] U. Störl, T. Hauf, M. Klettke, and S. Scherzinger. Schemaless NoSQL Data Stores – Object-NoSQL Mappers the Rescue? In *BTW'15*, 2015.

[9] J. D. Ullman. *Principles of Database and Knowledge-base Systems, Vol. I and II.* Computer Science Press, Inc., New York, USA, 1988.

[2] This differs from the semantics of copy- and move operations in our earlier work [6]. The new approach is consistent with the assumption in this paper, namely that the latest application release implicitly declares a schema, so *null* values are introduced for missing values.