Cleager: Eager Schema Evolution in NoSQL Document Stores

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

Meike Klettke
Universität Rostock
meike.klettke@uni-rostock.de

Uta Störl
Hochschule Darmstadt
uta.stoerl@h-da.de

Abstract: Schema-less NoSQL data stores offer great flexibility in application development, particularly in the early stages of software design. Yet over time, software engineers struggle with the heavy burden of dealing with increasingly heterogeneous data. In this demo we present Cleager, a framework for eagerly managing schema evolution in schema-less NoSQL document stores. Cleager executes declarative schema modification operations as MapReduce jobs on the Google Cloud Platform. We present different scenarios that require data migration, such as adding, removing, or renaming properties of persisted objects, as well as copying and moving them between objects. Our audience can declare the required schema migration operations in the Cleager console, and then verify the results in real time.

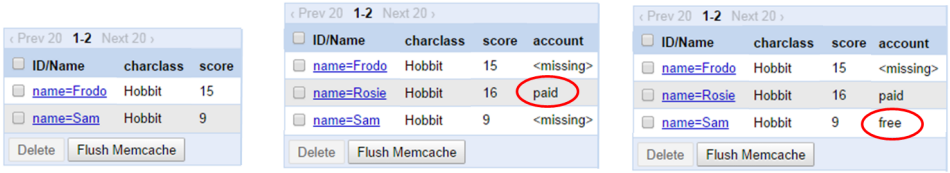
1 Schema Evolution in NoSQL Document Stores

This work concerns the maintainability of feature-rich, interactive web applications from the view-point of schema evolution. In particular, we target web applications backed by schema-less NoSQL document stores such as MongoDB [**?**] or Google Cloud Datastore [**?**]. This is an increasingly popular software stack: The programming APIs are easy to use, there is little setup time required with *database-as-a-service* offerings, and pricing is reasonable. Another sweet spot is that the schema does not have to be specified in advance, so developers may freely adapt the data structures as the application evolves.

Yet over time, dealing with an increasingly heterogeneous collection of persisted objects becomes a nuisance. Since today's schema-less NoSQL data stores still lack the proper tools for managing schema evolution, even the most basic *schema modification operations* (by the terminology of [**?**]), such as adding or removing a field from persisted objects in batch, require programmatic solutions and therefore custom coding.

Example. We consider an online role playing game hosted on Google App Engine and backed by Google Cloud Datastore [**?**]. Figure **??** shows the persisted player objects for Frodo and Sam. Each player has a unique *name*, further a *character class* and a *score*. The next release introduces paid accounts. Figure **??** shows the new object mapper class declaration with an *account* property. The annotation `@Id` declares the identifying property and `@Index` declares that a property can be queried (using Objectify [**?**] syntax).

Figure **??** then shows the data instance after second-generation player Rosie has been



(a) First-generation players. (b) Adding a second-generation player. (c) After lazy migration of player Sam.

Figure 1: Screenshots from the Datastore Viewer illustrating the evolution of persisted objects (termed "entities") in Google Cloud Datastore over time.

```
@Entity
public class Player {
  @Id private String name;
  private String charclass;
  private int score;
  @Index private String account;

  /* Lazy migration after loading. */
  @OnLoad void onLoad() {
    if (account == null)
      account = "free";
  }
  /* Omitting getters and setters. */
}
```

(a) Java class declaration with Objectify annotations.

(b) Query evaluation on the players from Figure **??**.

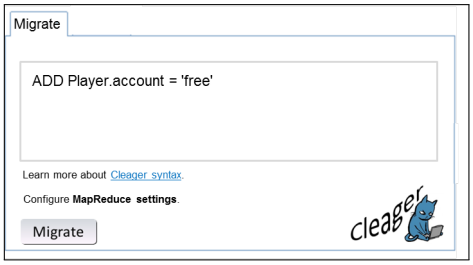
Figure 2: (a) Lazily introducing the new property *account* and (b) querying for free accounts.

persisted. As a schema-less NoSQL data store, Datastore allows for both generations of players to co-exist. Thus, the new code can be released without having to immediately migrate the legacy players Frodo and Sam. While this allows for rapid releases, it adds complexity to the application code. For instance, to search for all players with a free account, the GQL query (Google Query Language) in Figure **??** will return no results on the data instance from Figure **??**, since no persisted object satisfies the query predicate.[1]

One (partial) solution to this problem is *lazy* schema evolution through life-cycle annotations in object mapper class declarations, an increasingly popular approach [**?**]. When legacy player Sam is loaded into the application space with the class declaration from Figure **??**, the `@OnLoad`-method is executed. This sets a default value for the new attribute *account*. After the changes have been persisted, Sam's player object has been successfully migrated (c.f. Figure **??**). Yet unless Frodo's object is loaded into the application space, the missing property will not be added.

To reliably evaluate this query, *all* legacy objects need to be migrated. Today, developers actually write custom code to *eagerly* migrate legacy objects. Writing custom code is time consuming and requires careful testing, since the code is executed directly on production data. The Cleager framework offers a new and declarative alternative for eager migration.

[1] In GQL, it is actually *not possible* to formulate a query that returns players Frodo and Sam by a SQL-style query predicate such as "`where account is not null`". GQL queries are evaluated over indexes, and missing values are not indexed. We refer to [**?**] for a brief overview over *null* values in NoSQL data stores.



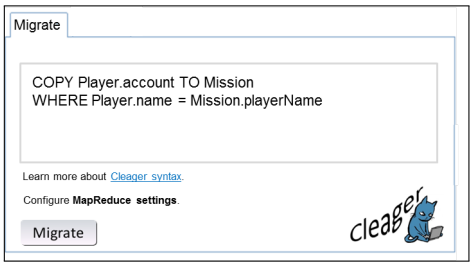
(a) Adding a property in the Cleager console.

```
def map(String key, JSON value) {
 if value is a Player object
 then value.addProperty("account", "free");

 put(key, value); // Persisting the change.
}

/* A map job is sufficient. */
```

(b) The MapReduce job for adding a property.



(c) Copying a property in the Cleager console.

```
def map(String key, JSON value) {
 if value is a Player object
 then emit(value.name, value);

 if value is a Mission object
 then emit(value.playerName, value);
}

def reduce(String key, List<JSON> valueList) {
 let p be the Player object in valueList;
 for each Mission m in valueList do
  m.setProperty("account", p.account);
  put(m.key, m); // Persisting the change.
 od
}
```

(d) The MapReduce job for copying a property.

Figure 3: The Cleager console with schema modification operations ((a) and (c)) for migrating first-generation players. The declarative operations are then executed as MapReduce jobs ((b) and (d)).

2 MapReduce-powered Eager Data Migration with Cleager

In [**?**] we have proposed a declarative *NoSQL schema evolution language* by which developers can specify common schema modification operations, such as adding, deleting, or renaming properties in batch. This covers the bulk of operations frequently required in web development, according to a study in [**?**]. In addition, our language allows that properties are moved or copied between objects, since data duplication and denormalization are fundamental principles in NoSQL data stores. This goes beyond the schema modification operations commonly supported by relational databases. For more complex and infrequent modifications (e.g., splitting or merging property values), developers can always resort to custom MapReduce jobs. We refer to [**?**] for the full specification of the Cleager language.

Figure **??** shows the Cleager console with a declarative statement that adds an *account* property to all players. Cleager executes this migration as a MapReduce job. For adding, removing, and deleting properties from persisted objects, a simple Map jobs suffices. Figure **??** shows the MapReduce job for adding an *account*-property in pseudo-code. Here, the reducer implements the identity function. This eager migration is to be executed on the legacy player objects prior to the new release, which introduces paid accounts.

Figure **??** shows the declaration for copying property *account* from the player object to all missions belonging to this player. Since Google Cloud Datastore, like most NoSQL data

stores, does not support joins in its query language, data denormalization is a common practice to avoid having to implement joins at runtime within the application code. In Figure **??**, we show the pseudo-code of the corresponding MapReduce job.

Cleager thus translates declarative schema migration operations into MapReduce jobs, making use of massive parallelization on the Google Cloud Platform infrastructure [**?**].

3 Demo Outline

1. We present a small gaming application running as software-as-a-service on Google AppEngine backed by Google Cloud Datastore. Our audience can inspect the player data in the Datastore Viewer. We then show how changes to the application code cause heterogeneous player objects to co-exist in the data store, as in Figure **??**.
2. We show that lazy schema evolution via object mapper libraries only affects those objects that are loaded into the application space at application runtime.
3. We show how queries over heterogeneous data can lead to unexpected results, since legacy objects may not be considered in query evaluation, as depicted in Figure **??**.
4. Our audience may write eager schema migration operations using our language from [**?**], within the web-based Cleager console (e.g., as seen in Figure **??**).
5. We monitor the execution of the schema migration operations as MapReduce jobs in the Google Cloud Platform on large data sets. Together, we then inspect the results.

Acknowledgements: We thank Kok Wing Tang from OTH Regensburg for implementing a first prototype of Cleager as part of his Bachelor thesis.