Distributed and Parallel Databases (2022) 40:5–25
https://doi.org/10.1007/s10619-021-07334-1

Self-adapting data migration in the context of schema evolution in NoSQL databases

Andrea Hillenbrand[1] · **Uta Störl**[1] · **Shamil Nabiyev**[1] · **Meike Klettke**[2]

Accepted: 15 March 2021 / Published online: 30 April 2021

Abstract

When NoSQL database systems are used in an agile software development setting, data model changes occur frequently and thus, data is routinely stored in different versions. The management of versioned data leads to an overhead potentially impeding the software development. Several data migration strategies exist that handle legacy data differently during data accesses, each of which can be characterized by certain advantages and disadvantages. Depending on the requirements for the software application, we evaluate and compare different migration strategies through metrics like migration costs and latency as well as precision and recall. Ideally, exactly that strategy should be selected whose characteristics fulfill service-level agreements and match the migration scenario, which depends on the query workload and the changes in the data model which imply an evolution of the database schema. In this paper, we present a methodology of self-adapting data migration, which automatically adjusts migration strategies and their parameters with respect to the migration scenario and service-level agreements, thereby contributing to the self-management of database systems and supporting agile development.

Keywords Databases · NoSQL · Data migration · Schema evolution · Self-adapting · Self management

✉ Andrea Hillenbrand
andrea.hillenbrand@h-da.de

Uta Störl
uta.stoerl@h-da.de

Shamil Nabiyev
bdcc.fbi@h-da.de

Meike Klettke
meike.klettke@uni-rostock.de

[1] Computer Science Division, Darmstadt University of Applied Sciences, Darmstadt, Germany

[2] Institute of Computer Science, University of Rostock, Rostock, Germany

1 Introduction

Especially in agile software development requirements change rapidly. Thus, schema-flexible databases have become popular among agile developers as it eases the burden of adapting the database every time the application code changes. However, this flexibility comes at the price of a certain *structural entropy*, i.e., a disarray of the database. If legacy entities are not migrated instantaneously in the event of a data model change, the new application code assumes a different data model than what is being stored in the production database. We refer to this circumstance as *schema evolution*, as the evolution of the data model also implies a schema change in the database.

Figure 1 illustrates the situation with a very simple example. In an online gaming scenario, scores are saved for the players (cf. Fig. 1a). At a certain point in time, the attribute `score` is renamed to `points` in the application code and new entities are then saved accordingly (cf. Fig. 1b). Now, when a query is executed for all players with `points>50`, the legacy entity in Fig. 1a is not retrieved, which is incorrect.

This could be resolved through *query rewriting* [3, 10], which rewrites the queries to reach all legacy entities by keeping track of past schema changes. However, over time the exclusive use of query rewriting leads to high latency for data accesses because of the increasing structural entropy of the database. Therefore, software developers and database administrators are well-advised to take care of a suitable data migration management.

The central issue in data migration management involves a decision how and when to migrate legacy entities according to the latest data model. With the choice of a data migration strategy it can be determined what amount of incurred cost is invested into the structural homogeneity of the database in order to lower latency times. Ideally, just the intended data entities are migrated at an opportune point in time. What that is exactly depends on management requirements, like service-level agreements (SLAs), and characteristics of the migration scenario. The options in this decision with regard to the choice of a migration strategy generally range from an *eager* strategy to a *lazy* strategy in addition to other strategies that lie in between these alternatives. The *eager* strategy is an approach with which data entities, that are affected by a schema change, are migrated to the current schema such that no overhead in latency exists when accessing these data entities. The *lazy* strategy, on the other hand, minimizes the migration costs as only those legacy entities are

```
{
  "id"    : 1234,
  "name"  : "Lisa",
  "score" : 120,
  "ts"    : 13
}
```

(a) New `player` entity at timestamp 13.

```
{
  "id"     : 1235,
  "name"   : "Bart",
  "points" : 60,
  "ts"     : 42
}
```

(b) New `player` entity at timestamp 42.

Fig. 1 Different structure of data entities in the database

migrated that are being accessed, while all other entities remain in their respective schema versions. The characteristics of the migration strategies are discussed in detail in Sect. 2 by means of different metrics that are going to be introduced.

In other words, the decision on the data migration strategy is also a decision on a tradeoff between short latency times, which is crucial for application performance, and between saving costs of unnecessarily migrating legacy entities that are unlikely to be accessed in the future. On a management level, this tradeoff can be utilized to constitute *heuristics* of how legacy entities should be dealt with, and even better, the different heuristics can be made automatic on a data migration level in the form of adaptive migration strategies based on these heuristics. Deciding on the tradeoff between the metrics latency and migration costs is not the only way data migration can be managed. The decision-making can also be based on or complemented by two other metrics that describe that very same tradeoff from a perspective common in information retrieval, i.e., the tradeoff between the classification metrics *precision* and *recall* [15]. Depending on how successful the selection of migrated data entities has been in terms of them actually being accessed, it indicates how well migration costs have been spent in order to improve on latency.

Now, since SLAs and other requirements of application performance or budgetary limits should be known at the time of the software development, at least in the order of magnitude, an approach of a *self*-managing database, which adapts the data migration strategy automatically based on the given constraints, has its advantages. The most important advantages are the reduction of the complexity of the software development and the direct control of the compliance with the SLAs as self-adaptation does not rely on management intervention or the reaction time of the database administrator.

Contributions. We contribute a methodology of self-adapting data migration in NoSQL databases in order to facilitate an automatic compliance of application requirements specified as service-level agreements (SLAs). As database systems evolve frequently in agile development settings, self-adapting data migration can significantly speed up the development process and realize a cost-efficient self-management of database systems. We explore and investigate the possibilities of adaptation by adjusting migration strategies and their parameters with respect to the migration scenario. We take all important characteristics of possible migration scenarios into account that affect application requirements with the goal of eventually realizing a fully automatic database with regard to data migration. We present a methodology of data migration that allows for different patterns and intensity of the query workload that has to be handled, the number and kind of changes in the data model implying schema evolution in the database. Based on this, we devise new migration strategies that self-adapt by utilizing the tradeoffs between the most important metrics in this context, i.e., the metrics of latency and migration costs and the metrics precision and recall, thereby clarifying the operative decisions that have to be made to meet the application requirements. We have implemented one of the proposed adaptive migration strategies into our data migration middleware and can show that it performs better compared to the common strategies.

This present paper is an extended version of a conference paper carrying the title *Towards Self-Adapting Data Migration in the Context of Schema Evolution in*

NoSQL Databases [5]. Beside an elaboration and restructuring of contents in general, this present version discusses another dimension in which data migration can be adapted, that is, how the classification metrics *precision and recall* can be utilized in self-adaptive approaches.

Structure. This paper is structured as follows: In Sect. 2, we discuss the characteristics of particular data migration strategies. In Sect. 3, we introduce the architecture of our schema management middleware *Darwin* featuring various migration strategies, as well as the tool-based migration advisor *MigCast*, including exemplary charts of the metrics, with which we can assess the performance of the migration strategies. In Sect. 4, we discuss four approaches how migration strategies can be advanced to exhibit self-adaptive capabilities. In Sect. 5, we discuss the realization and simulation results of one such self-adaptation approach, an adaptive migration strategy, which we have already implemented in our middleware *Darwin*. In Sect. 6, we address related work, before we conclude with an outlook in Sect. 7.

2 Data migration strategies

In Sect. 2.1, we define the used terms of this paper. In Sect. 2.2, we get into the details of the data migration strategies and discuss their advantages and disadvantages depending on the migration scenario. In Sect. 2.3, we sum these characteristics up in the form of a compact table.

2.1 Notions concerning data migration strategies

By the term *data access latency* we refer to the time that it takes to retrieve a requested data entity, i.e., a read access. Latency's antagonist in the competition for optimal application performance are the *data migration costs*, by which we refer to the charges occasioned by migrating the data. We differentiate between *on-release* and *on-read* migration costs, which together add up to the *cumulated migration costs*. *On-release* migration costs are caused when entities are migrated in the event of a schema change and depend on how many entities are affected and how these legacy entities are handled. In contrast to on-release migration costs, *on-read* migration costs are caused when entities are being accessed that exist in older versions than the current schema indicates. Furthermore, we refer to the charges, that would have to be invested in order to migrate all legacy entities to a structurally homogeneous database instance, as *migration debt*. We abbreviate these terms by *latency*, *migration costs* when referring to the cumulated costs, and *migration debt*. Figure 3 depicts the metrics for all migration strategies in a particular migration scenario, which we discuss in Sect. 3. As regards the used classification metrics in this paper, the notion of *precision* is defined as the fraction of correctly predicted entities among the total predicted entities, and *recall* as the fraction of correctly predicted entities among the accessed entities.

2.2 Data migration strategies

2.2.1 The eager migration strategy

With the *eager* strategy, all legacy entities are migrated instantaneously at the release of data model changes. This being the case, the application code can directly access a structurally homogeneous database instance at any point in time. Then data access latency is minimal as there is no data structure entropy involved, which would need to be accounted for, and no migration debt can be accumulated. However, this optimal condition of the database is paid for accordingly in terms of migration costs, as all data needs to be kept up-to-date, even that data which is likely to not ever be accessed again in the future.

2.2.2 The lazy migration strategy

Whereas *eager* migration resolves the above-mentioned tradeoff between migration costs and latency in favor of optimal latency no matter the price, *lazy* migration can be found at the opposite end of this tradeoff. As the name already implies, with *lazy* migration the legacy data remains completely unchanged in the event of a release of data model changes. However, in case that legacy entities are being accessed, they are migrated on-the-fly in order to adhere to the new data model, which causes a runtime overhead [4, 12]. This situation occurs rather often and may or may not come at an opportune time with regard to application performance. With the *lazy* migration strategy software developers can react flexibly to agile requirement changes at no immediate migration costs, however, this entails carrying the full debt of structural entropy.

A compromise between the two competing goals can be reached by migrating data *proactively*, i.e., by acting in advance of situations when migrating legacy entities could cause latency overhead. Ideally, exactly those legacy entities should be migrated that will be accessed in the near future. Although future data accesses cannot be predicted reliably, data migration strategies can be thought of that proactively migrate data reasonably. Here, we present *incremental* and *predictive* migration, which we have implemented in our middleware tool *Darwin* (cf. Sect. 3). In Sect. 5, we discuss another proactive strategy, the *adaptive* strategy, which we have implemented as exemplary realization of a *self-adaptive* approach serving as a proof of concept of Sect. 4.3.

2.2.3 The incremental migration strategy

With *incremental* migration, releases of data model changes are usually treated like with *lazy* migration, i.e., legacy data remains unchanged and the migration debt increases accordingly. However, lazy periods of time are interrupted by regular bouts of tidying up the database. Then, the structurally heterogeneous database instance migrates all legacy entities and thus, gets rid of the runtime overhead that is caused by updating legacy entities on-the-fly when being accessed. *Incremental* migration usually does this tidying up in certain *increments* of time coinciding with particular

releases of data model changes at a regular cycle. Ideally, this regular database update can be matched with known periods when data accesses are less frequent, for example at night or weekends. This being the case for *incremental* migration strategy, latency and migration costs vary between the opposite approaches of *eager* and *lazy* migration. Thus, a compromise on the above-mentioned tradeoff is being reached on a regularly alternating basis, with periods of no and periods of moderate migration debt.

Similarly, it also appears beneficial to migrate all legacy entities incrementally prior to periods of frequent and extensive data accesses in order to take particular advantage of low latency times. In Sect. 4, we initiate the *incremental* migration thereby choosing these points in time, based on certain events, in order to settle all of the migration debt, so that latency is optimal again when it is required.

2.2.4 The predictive migration strategy

Although correctly forecasting all future data entity accesses hardly seems possible, a *predictive* approach can be justified by the assumption that the oftener entities were accessed in the past, the more likely it is that they be accessed again in the future. The so-called *hot* data, i.e., frequently accessed data, should be kept up-to-date, as otherwise the runtime overhead and on-read migration costs would have to be taken into account just as often as their schema changes.

As a consequence, *predictive* migration is implemented in our middleware by keeping track of past data accesses while ordering the accessed entities accordingly via *exponential smoothing*. This established technique in time series data weighs the entities by their actuality and access frequency: The more recent the entity accesses, the higher the weight of the entity. However, past accesses are not weighed equally, but weights decrease exponentially over time simulating an aging process of the entities by accounting for actuality as well as for access frequency [8]. In *Darwin*, the ordered, to-be-migrated entities are kept in the so-called *prediction set*. By this approach, a subtler compromise is being reached on the tradeoff between latency and migration costs in comparison with *incremental* migration. The basis for this is the Pareto distribution of cold and hot data, which is very common in OLTP database applications [8]. This phenomenon is utilized in the *predictive* approach to keep a reasonable balance of spending on migration costs in order to improve latency. The prediction set has a certain, fixed size, which is defined prior to the application of the strategy. Data, that is not contained in the prediction set, is migrated when it is accessed lazily. In Sect. 4, we adapt the *predictive* strategy by means of controlling the prediction set size according to SLA requirements and the migration scenario.

2.3 Summary

Table 1 summarizes and evaluates the characteristics of the presented data migration strategies including two more references of comparison: the possibility of an *adaptive* migration strategy, which is motivated and defined in Sect. 4.3 and evaluated in Sect. 5, and the *no-migration* scenario. In the latter scenario, it is not possible to

Table 1 Characteristics of the discussed data migration strategies with regard to certain metrics, including the reference of the no-migration strategy

Migration strategy	Migration costs	Latency overhead	Migration debt	Effort for query rewriting
Eager	−−−	+++	+++	+++
Incremental	−−	o	++	o
Predictive	+	−	o	o
Lazy	++	−−	−−	o
Adaptive	o	+	o	o
None (versioning)	+++	−−−	−−−	−−−

migrate any data at all, for example, for regulatory or other technical reasons. Then all datasets are kept in these particular versions. In order to access datasets in previous or subsequent versions, backward and forward query rewriting has to be used, respectively, which *rewrites* each query by distributing them onto all present schema versions [3, 10]. This exclusive use of query rewriting, though, leads to a very high latency overhead and migration debt.

The characteristics of the compared strategies are the metrics of (cumulated) migration costs, latency overhead compared to eager migration, that is, compared to reading data that is already migrated, migration debt, and in the last column, effort for query rewriting. Regarding the semantics of the used symbols, the ratings do not imply a high or low value, but describe whether the characteristic behavior is favorable or unfavorable. For example, high migration costs are usually avoided and are thus represented by as many signs of "–", as opposed to potentially saved costs of migration debt expressed through as many signs of "+". A neutral "o" implies a balanced compromise on a particular metric. Compare the table with Figs. 3 and 6, which show *MigCast*-generated charts illustrating the evaluation of the metrics in terms of each of the migration strategies.

3 Architecture

In this section, we describe our tool-based advisor *MigCast* which is based on our schema management middleware *Darwin*, both visualized in Fig. 2. Following a discussion on the architecture, exemplary *MigCast*-generated charts are shown in Fig. 3, by which we discuss the above defined metrics as an example of a migration scenario.

Darwin supports the entire schema management life cycle [14]. Schema management for NoSQL databases consists of two main tasks: The *schema evolution management* as such, and the *data migration*, which has to be embedded in a precise schema evolution management in order to be a safe and sound process. For the task of schema evolution management, we use *Darwin* to initially declare or extract schemas, define schema evolution operations, or extract schema versions and schema

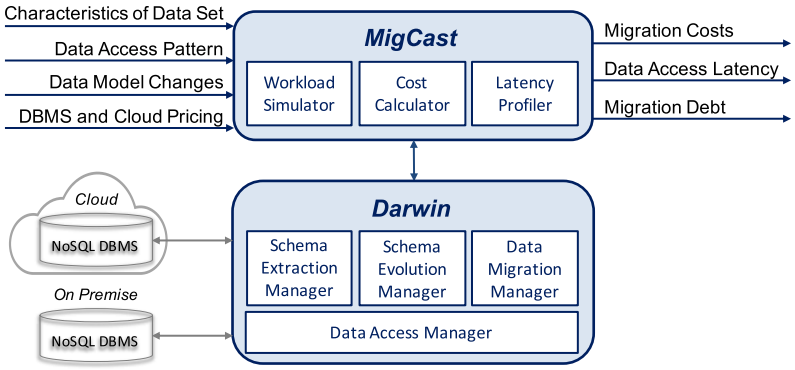


Fig. 2 Architecture of the migration advisor *MigCast* based on the middleware *Darwin*, their submodules as well as input parameters and calculated metrics

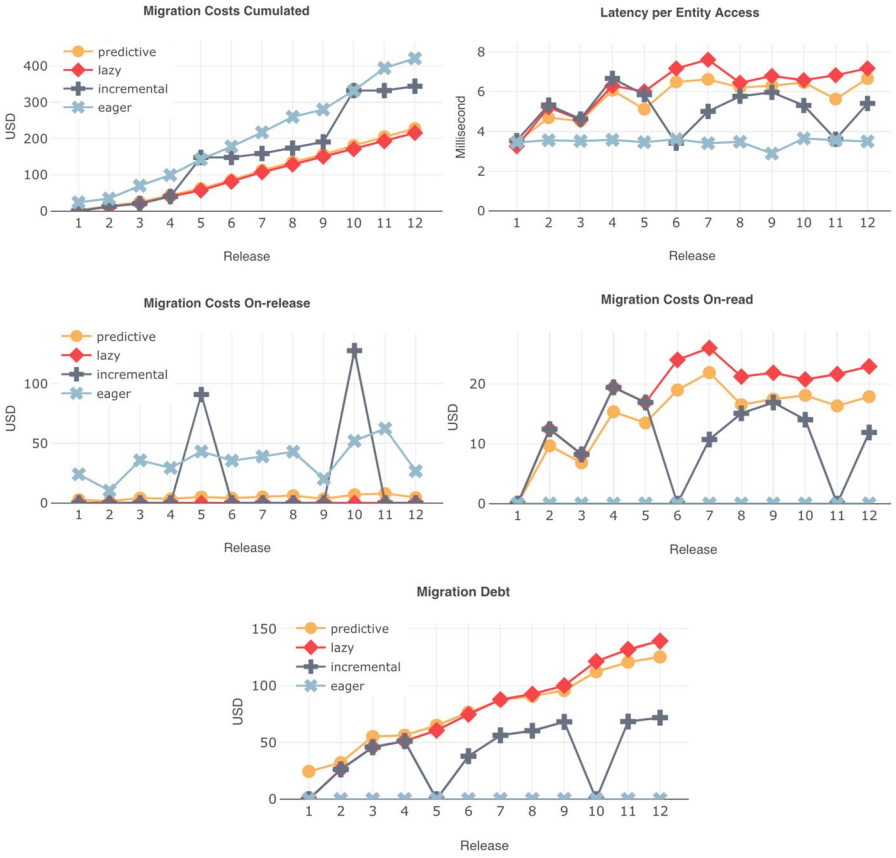


Fig. 3 Impact on the metrics for different migration strategies in an exemplary migration scenario: High workload of data entity accesses in a Pareto pattern, medium multi-type complexity, low cardinality of 1:1-relationships of the underlying data model, growth rate of entities 10% beginning with 1,000 simulated entities and scaled up to 10M entities, and a cloud price USD 0.2 per 1M I/O requests

evolution operations from legacy data. We have presented these functionalities in Klettke et al. [7] and Störl et al. [14]. In this present paper, we present a methodology of self-adapting data migration which builds on our demo paper about *MigCast* [4], which focuses on data migration itself.

All data migration strategies presented in Sects. 2 and 5 are supported by *Darwin*. The data migration is performed in *Darwin* by the *Data Migration Manager* based on the schema evolution operations managed by the *Schema Evolution Manager*. The *Schema Evolution Manager* implements different kinds of schema evolution operations that we have defined in Scherzinger et al. [13]: *Single-type* schema evolution operations affect just one class, table, or entity type, respectively, of which we implemented `add`, `delete`, and `rename` into our middleware, whereas *multi-type* operations affect several entity types at once, of which we implemented `copy` and `move`. All these schema evolution operations could theoretically also be performed using the `add` and `delete` operations. Thus, all schema changes could be executed using these schema evolution operations.

In Scherzinger et al. [13], we also discussed that the create-or-replace semantics inherent in our NoSQL database programming language make for a *well-defined* behavior of schema evolution operations. This facilitates safe and efficient migration processes. Without the use of such a schema management middleware, the data migration would have to be performed within the application, or even less elegantly in the context of NoSQL schema evolution, by migration scripts, both of which would likely turn out quite costly and error-prone [12]. Furthermore, the migration strategies and metrics discussed in Sect. 2 have been developed and tested for document, wide-column, and multi-model data stores. *Darwin* interfaces with popular NoSQL database management systems, among them *MongoDB*, *Couchbase*, *Cassandra*, and the multi-model database *ArangoDB*.

Based on *Darwin*, we have implemented the tool-based advisor *MigCast* for exploring data migration strategies [4] in order to examine the effects of the data migration strategies with regard to the metrics of migration costs, latency, and migration debt. As illustrated in Fig. 2, *MigCast* calculates these metrics based on the characteristics of the database instance, the characteristics of the data access pattern, the number and complexity of the data model changes, and the options of several database management systems and of different cloud pricing models. Then, *MigCast* visualizes the metrics for each of the migration strategies in different graphs showing their development through 12 releases of schema changes.

Based on the above characteristics the metrics are calculated as follows: On a sample set of data, the *Workload Simulator* of *MigCast* executes the workload of served data entity accesses between releases of new software, which is parameterizable to simulate different distributions of data entity accesses. The accesses can cause on-read migration costs with migration strategies that delay migrating legacy entities. Then, *Darwin* performs the schema evolution operations implied by the data model changes corresponding to the consecutive releases. After the schema evolution operations are applied, affected legacy entities are migrated depending on the migration strategy, which can cause on-release migration costs. The metrics are determined by help of a *Cost Calculator* and a *Latency Profiler* of *MigCast* (cf. Fig. 2). These modules deliver results based on a cloud provider pricing model per

served 1M of I/O-requests, i.e., read or write requests, and the time that it takes to serve the requests.

In order to illustrate this, Fig. 3 depicts the on-read and on-release migration costs during the releases of schema changes and their cumulation, as well as entity access latency and migration debt of the discussed data migration strategies for a specific and typical migration scenario. The charts are generated using a Pareto-distributed data access pattern, 50% of multi-type schema operations, 1:1-relationships of the cardinality of the underlying data model (an online gaming scenario more detailed as in Fig. 1), MongoDB and a cloud provider pricing model of USD 0.2 per served 1M of I/O-requests. Other migration scenarios can be simulated by *MigCast* as all characteristics are parameterizable. The x-axes represent consecutive software releases, while the y-axes represent each of the discussed metrics.

As can be observed consistent with the discussion of the migration strategies in Sect. 2, and in particular, consistent with the summary of the characteristics of the data migration strategies in Table 1, the graph of the *eager* strategy represents an upper bound of the cumulated migration costs and on-read costs, as well as a lower bound of latency and migration debt. Analogously, the *lazy* strategy can be observed to be the lower bound in terms of cumulated migration costs and on-release costs and an upper bound with regard to latency, migration debt and on-read migration costs. In this sense, the *eager* and *lazy* strategy can be viewed as baselines, by which the other migration strategies can be assessed in terms of their performance with regard to the different metrics. As we can see for instance, the overhead on latency is oftentimes twice as high with the *lazy* strategy than with the *eager* strategy, yet half of migration costs are accumulated illustrating the discussed tradeoff between migration costs and latency. Consistently in between the *eager* and *lazy* strategies, the other discussed strategies can be found, with the exception of the *incremental* strategy in terms of on-release migration costs at releases 5 and 10 when *eager* migration is initiated at preset increments. The *predictive* migration has advantages in terms of lower latency compared to *lazy*, despite only marginally higher migration costs, though being caused by on-release costs beside on-read costs only in case of *lazy*.

In different migration scenarios, the migration strategies show different characteristics. For instance, at lower workload of data entity accesses those migration strategies that utilize the Pareto distribution pattern, e.g., the *lazy* strategy, are measured at lower on-read and cumulated migration costs. If the workload, however, is evenly distributed among the data entities, then the *lazy* approach produces more migration costs. In case that the growth rate of the number of data entities is higher, the slopes of the graphs are tend to become higher. These and other correlations between migration scenario characteristics and metrics of migration strategies have been the focus of our research, but go beyond the scope of this paper (rf. to Sect. 7 for an outlook).

Now, the question how and when data migration strategies can be adapted, or self-adaptive, is going to be presented in the following section. The results calculated by *MigCast* are crucial in order to monitor, predict, and verify the effects of the data migration strategies in terms of the metrics. In this sense *MigCast* serves

as a migration advisor at the same time facilitating an automatic adjustment of the migration strategies.

4 Self-adapting data migration

In the following, we determine *what* options we have for data migration to be self-adapting and *when* self-adaptation is advisable. Depending on these determined options and criteria, we can describe *how* data migration can be self-adapting. Addressing all of these dimensions of self-adaptation of data migration in unison constitutes the most important task towards realizing a self-management of database systems.

4.1 The "what" of self-adapting data migration

If adaptation is advisable, then there are two options of self-adaptation that can be distinguished:

- Step 1: *Initiation* of a migration strategy that satisfies all constraints.
- Step 2: *Adjusting the parameters* of the selected migration strategy.

Ideally, the choice of a migration strategy and its parameters are adapted to suit the migration scenario and fulfill all required constraints.

4.2 The "when" of self-adapting data migration

Let us gather criteria for adaptation in order to be able to determine how data migration can be self-adapting. Criteria that make data migration advisable in order to approximate an optimal decision on the tradeoff between latency and migration costs, and/or the tradeoff between precision and recall, respectively, can be taken from stipulated service-level agreements (SLAs). They can also be inquired of software project stakeholders or deduced from non-functional requirements such as usability, performance, and availability.

We investigate the following criteria:

- Compliance with an average or maximum *latency* when accessing data, or a change in latency within a certain period, as a quantitative descriptor.
- Compliance with a maximum limit for *migration costs* within a certain period as a quantitative descriptor accounting for budgetary liabilities.
- Compliance with a maximum *migration debt* at a certain point in time as a quantitative descriptor accounting for necessary budgetary provisions.
- Compliance with an average or minimum *precision* when accessing data, or a change in the precision within a certain period, as a qualitative descriptor for the relevance of past migrations.

– Compliance with an average or minimum *recall* within a certain period as a qualitative descriptor for the relevance of past migrations.
– Compliance with a maximum *database entropy* as a qualitative descriptor accounting for the fact that data migration for *multi-type* operations is considerably more expensive than for single-type operations.

In a self-adapting database, the question when to adapt data migration can be answered in respect of the above: If a threshold value of stipulated requirements of the SLAs is exceeded, then actions are taken accordingly and automatically, so that it does not require a repeated inquiry of or intervention by stakeholders. Notwithstanding the above, it is conceivable that adaptations can also be implemented by software developers proactively without formally stipulated SLAs, yet the discussed options and criteria should be assessed carefully in order to comply with the best practices of database management.

4.3 The "how" of self-adapting data migration

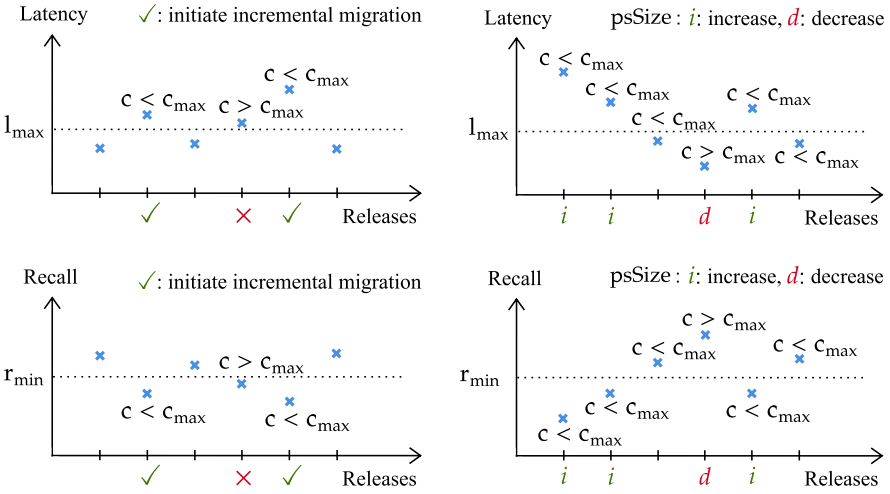
In the following, we describe how adjustments to migration strategies can be made automatic based on the previously determined options and criteria. Depending on the distribution of data entity accesses that have to be served, what we call the *workload pattern*, several use cases can be distinguished. The remainder of this section is thus structured as follows: We distinguish by the kind of query workload pattern, i.e., whether the data accesses of the workload can be assumed to be localized or whether this information is not available. We discuss for both cases how adjustments can be put into effect automatically based on the previously determined options and criteria in order to approximate an optimal decision on the tradeoffs between latency and migration costs and/or between precision and recall.

4.3.1 Random distribution of data accesses

The first and basic use case assumes that the distribution of data accesses does not contain adaptation-relevant information, i.e., the distribution is either highly variable, random, unpredictable, or unknown at the time. In this case, where we cannot utilize any information to our advantage, we suggest an *self-adaptively incremental* approach in which incremental migration is initiated at a varying frequency depending on the above discussed requirements, as depicted in Fig. 4a. Precisely, during new software releases implying schema changes (horizontal axes), the incremental migration is initiated at a frequency so that the *average latency in a certain period of time* $\mathbf{l}$ complies with the requirement of a maximal latency $\mathbf{l_{max}}$, while the *average migration costs in a given period of time* $\mathbf{c}$ do not exceed the maximum migration costs $\mathbf{c_{max}}$:

$(\mathrm{I}_{\mathrm{l/c}})$ **if** $\mathbf{l} > \mathbf{l_{max}} \wedge \mathbf{c} < \mathbf{c_{max}}$ **then** execute an incremental migration;

We use $\mathbf{l_{max}}$ as threshold to illustrate the principle. Of course, $\mathbf{l_{max}}$ should be specified slightly below the value agreed in the SLA, so that this threshold is certain to not be exceeded (safety margins for thresholds can be applied analogously with



(a) Applying a *self-adaptively incremental* approach by using the tradeoff between migration costs and either latency or recall.

(b) Applying a *requirement-adaptive* approach by using the tradeoff between migration costs and either latency or recall.

Fig. 4 Self-adaptation of data migration strategies with random (left) or localized (right) read accesses controlled through different metrics

all other metrics, but are left out for brevity). In case that an initiated incremental migration would exceed the migration costs $\mathbf{c_{max}}$, the requirements can be prioritized by stakeholders in advance. If $\mathbf{l_{max}}$ must be complied with at any cost, then $\mathbf{c}$ is disregarded:

(I_l) **if** $\mathbf{l > l_{max}}$ **then** execute an incremental migration;

In addition or as an alternative to the above, the *average recall in a given period of time* $\mathbf{r}$ is required to comply with the minimal recall $\mathbf{r_{min}}$, while the *average migration costs in a given period of time* $\mathbf{c}$ do not exceed the maximum migration costs $\mathbf{c_{max}}$. We formulate the conditional expression by substituting recall for latency, since the fraction of correctly predicted entities among the accessed entities is correlated (inversely proportional) to latency as with the investment of migration costs recall becomes higher and this translates to a lower latency, and vice versa.

($I_{r/c}$) **if** $\mathbf{r < r_{min} \wedge c < c_{max}}$ **then** execute an incremental migration;

Accordingly, imagine that latency would have to be minimal, i.e., there must not exist any legacy entities that are accessed causing latency overhead no matter the migration costs, and this would be better controlled by a measure like the recall which technically guarantees an optimal latency. In this admittedly technical case, an eager migration strategy is emulated by means of the incremental strategy, where $\mathbf{r}$ frequently reaches $\mathbf{r_{min}} = 1$:

(I_r) **if** $\mathbf{r < r_{min}}$ **then** execute an incremental migration;

If we assumed that there are no parallel migration routines, as is ordinarily the case, then $r < 1$ would be tantamount to the situation that schema evolution operations have taken place and incremental migration is about to be initiated.

4.3.2 Localized distribution of data accesses

In many database applications a certain bias can be detected where accesses concentrate more on some data entities than on others [8]. In case that data accesses are localized, we can utilize this additional information to our advantage for the purpose of self-adapting data migration. We have explained in Sect. 2 that the *predictive* approach is superior compared to approaches that do not take the distribution of data accesses into account. Here, latency can be kept lower despite a growing migration debt that is caused through different versions of data. We discuss for this use case how adjustments to the *predictive* approach can be put into effect automatically based on the previously determined criteria in order to approximate an optimal decision on the tradeoff between latency and migration costs. We distinguish:

- a *complexity-adaptive* approach avoiding a backlog of complex schema evolution;
- a *requirement-adaptive* approach as a feedback control system based on requirements with respect to latency and migration costs or recall and migration costs;
- a *relevance-adaptive* approach as a feedback control system based on requirements with respect to the ratio of precision and recall;
- an *efficiency-adaptive* approach, measured by the ratio of latency improvement by invested migration costs.

4.3.2.1 The complexity-adaptive approach In *predictive* migration, the prediction set has a certain, fixed size. Data that is not contained in the prediction set is migrated when it is accessed lazily. If *expensive* operations have occurred in the course of the schema evolution history, then a latency peak can occur if data entities are accessed that are affected by such schema operations. For example, this usually happens during major schema changes in agile development settings. In detail, these expensive schema operations are `copy`, `move`, `split`, or `merge`, i.e., operations that affect several entity types at once [1, 13].

Now, this situation of a backlog of complex data model changes can be prevented by adjusting the prediction set size at certain points in time. In *Darwin*, we have implemented this increase of the prediction set size in case that a certain number of multi-type operations have accrued. This number, which represents a measure for the backlog of multi-type operations with respect to an entity type, can be specified in *Darwin* as a parameter, thereby allowing the possibility to control the degree of structural entropy in a database as it pertains to dependencies between different entity types.

This self-adaptation approach can be implemented by software developers as a proactive measure in order to avoid exceptionally long latency times. This proactive decision can be inquired of the stakeholders or lies solely in the hands of developers that are experienced with that particular kind of database-backed application. With the *complexity-adaptive* approach, the prediction set size **psSize** of the *predictive* migration is increased if a certain number of multi-type operations $\mathbf{m_{max}}$ since the

last increase of the prediction set size is exceeded and causes substantial database entropy.

$$(P_m) \quad \textbf{if } \#(\text{multi-type operations}) > \mathbf{m_{max}} \textbf{ then } \text{increase } \textbf{psSize}; \; ...//\text{subsequent migration } \textit{decrease } \textbf{psSize};$$

We have implemented this first approach in *MigCast* as an *adaptive* strategy and compare it in Sect. 5 with the other migration strategies outlined in Sect. 2. An initial value of **psSize** (equivalent to the prediction set size of the *predictive* strategy) should be chosen commensurate with the expected proportion of multi-type operations in the set of evolution operations. Without loss of generality, we specified the initial value in *MigCast* at 10% of the number of entities per type. An example of the advantage of this complexity-adaptive migration is demonstrated in Fig. 6 of Sect. 5.

4.3.2.2 The Requirement-Adaptive Approach In this approach, we suggest a feedback control system based on stipulated requirements regarding latency and migration costs. Let us assume that since that last release of data model changes the current latency exceeds the stipulated maximal latency $\mathbf{l_{max}}$. Then, the prediction set size is stepwise increased with each release, and data is migrated in the order of the prediction set until the stipulated maximal migration costs $\mathbf{c_{max}}$ are exhausted:

$$(P_{l/c}) \quad \textbf{if } \mathbf{l} > \mathbf{l_{max}} \wedge \mathbf{c} < \mathbf{c_{max}} \textbf{ then } \text{increase } \textbf{psSize}; \; \textbf{else if } \mathbf{l} \leq \mathbf{l_{max}} \wedge \mathbf{c} \geq \mathbf{c_{max}} \textbf{ then } \text{decrease } \textbf{psSize};$$

It follows that if $\mathbf{l} \leq \mathbf{l_{max}} \wedge \mathbf{c} < \mathbf{c_{max}}$ holds, then the prediction set is not changed. In this case, the system fulfills all requirements of the application and guarantees the SLA concerning latency and migration costs. If $\mathbf{l} > \mathbf{l_{max}} \wedge \mathbf{c} \geq \mathbf{c_{max}}$ holds, then the SLA targets cannot be fulfilled simultaneously at this specific time. In this case, different strategies can be applied, e.g., a notification of stakeholders to include a prioritization of either not exceeding migration costs or latency into the SLA targets.

Figure 4b shows an example of the behavior of the system. By using this algorithm, a sufficient latency is reached, maybe keeping some reserves for migration expenditures in future releases. Note that with a prediction set size of 100% the requirement-adaptive approach coincides with eager migration, and a prediction set size of 0% is synonymous with lazy migration.

Certain constellations can lead to repeated increases and decreases of the prediction set of alternating releases. If continuous switching is not desired, a delay can be built in which prohibits an increase if a decrease was executed in the previous release, and vice versa. In this particular case, the system behaves like a *hysteresis* control well-established in dynamic systems [9].

Note that, as with the self-adaptively incremental approach, the migration cost-latency tradeoff can be complemented by requirements for minimal recall thresholds. For instance, the requirement-adaptive adaptation ($P_{l/c}$) can be reformulated as:

$$(\mathrm{P}_{\mathrm{r/c}}) \quad \textbf{if } \mathbf{r} < \mathbf{r_{min}} \wedge \mathbf{c} < \mathbf{c_{max}} \textbf{ then } \text{increase } \textbf{psSize}; \\ \textbf{else if } \mathbf{r} \geq \mathbf{r_{min}} \wedge \mathbf{c} \geq \mathbf{c_{max}} \textbf{ then } \text{decrease } \textbf{psSize};$$

4.3.2.3 The relevance-adaptive approach Furthermore, we suggest a feedback control system based on requirements regarding the ratio of the classification metrics precision and recall, which could be stipulated in the SLA as well (cf. Sect. 2.1 for a definition of precision and recall). The measures of precision and recall can be considered as qualitative descriptors for the relevance of past migrations: If all predicted entities have been accessed, then precision is 100%, and if all accessed entities have been predicted, then the recall is 100%. In this case, all predictions were relevant. Analogously, if one entity was not accessed, yet predicted, it would not have been relevant to predict it to be accessed—the precision is $< 100\%$. If one entity was not predicted but accessed, thus being a legacy entity, then it would have been relevant to predict it to be accessed—recall is $< 100\%$. Since the accesses cannot generally be predicted exactly, a tradeoff stipulated in the SLA is tantamount to a certain ratio between precision and recall of α, i.e., the ratio of the accessed entities and the predicted entities.

$$(\mathrm{P}_{\mathrm{p/r}}) \quad \textbf{if } \mathbf{p/r} > \alpha \textbf{ then } \text{increase } \textbf{psSize}; \\ \textbf{else if } \mathbf{p/r} < \alpha \textbf{ then } \text{decrease } \textbf{psSize};$$

E.g., if $\alpha = 1$, then precision and recall are equal, which means that there are just as many falsely predicted as falsely not predicted entities. By means of α, the relative importance of the metrics can be controlled, for instance, when migration costs are relatively cheap and high latency is should be avoided as high opportunity costs, then α should be set lower in order to emphasize the importance of a higher recall. Depending on the scenario, a classification metric could be used that also considers correctly not predicted entities among the not accessed entities, like *selectivity*, or combinations of metrics, like the traditional F-measure [15].

As discussed above with requirement-adaptive adaptation, if continuous switching is not desired, a delay can be built in which prohibits an increase if a decrease was executed in the previous release, and vice versa. Alternatively, the ratio of precision and recall can be compared against an interval around α, $x < \alpha < y$, such that the prediction set size is increased if $\mathbf{p/r} > x$ and decreased if $\mathbf{p/r} < y$.

4.3.2.4 The efficiency-adaptive approach Last but not least, we suggest an adaptation with respect to the differentials of latency or recall. In order to illustrate this, we draw up correlations between latency and migration costs, and between recall and migration costs, respectively, in a coordinate system of Fig. 5. The correlations may be assumed negatively logarithmic and logarithmic, respectively, as data accesses are usually localized, i.e., Pareto distributed. As dependent variables, latency $\mathcal{L}$ and recall $\mathcal{R}$ are now plotted against the migration costs as discrete-valued steps of the prediction set size as the independent variable **psSize**. The derivatives of the latency and recall functions $\mathcal{L}'$ and $\mathcal{R}'$ indicate how they change with respect to changes of the prediction set size. In other words, the derivatives $\frac{d\mathcal{L}}{d\mathbf{psSize}}$ and $\frac{d\mathcal{R}}{d\mathbf{psSize}}$ can be viewed as

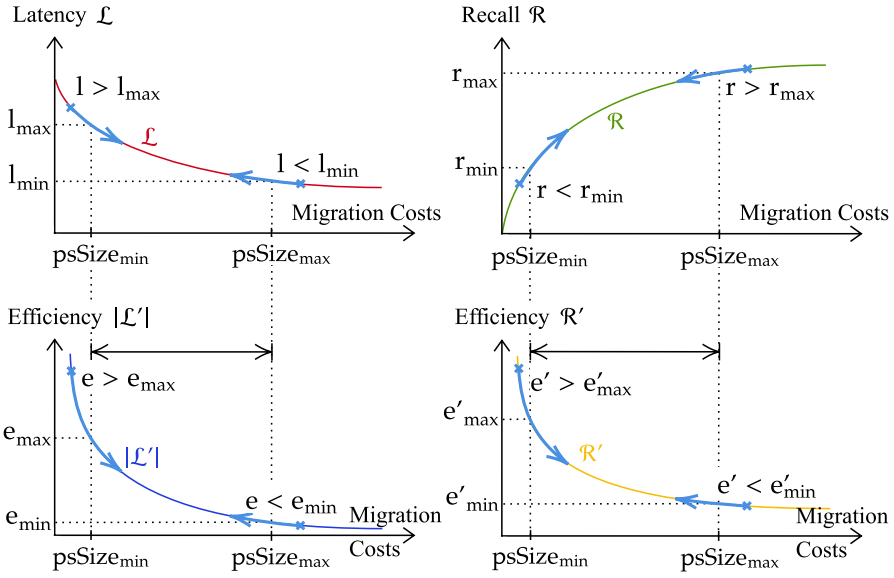


Fig. 5 Applying a efficiency-adaptive approach by means of derivatives of latency or recall as efficiency measures per invested migration cost

the efficiency of latency/recall change per migration investment. In order to remain consistent with the intuitive notion of efficiency, we draw up the negative derivative of latency.

As regards latency, we can use this notion in order to formulate a requirement by which migration, while improving on latency until $\mathbf{l_{min}}$, is intended to remain cost-efficient up to a stipulated minimal efficiency $\mathbf{e_{min}}$:

$$(\mathrm{P}_{\mathrm{l/e}}) \quad \begin{array}{l} \textbf{if } \mathcal{L}(\textbf{psSize}) > \mathbf{l_{max}} \textbf{ then} \text{ increase } \textbf{psSize}; \\ \textbf{else if } |\mathcal{L}'(\textbf{psSize})| < |\mathbf{e_{min}}| \textbf{ then} \text{ decrease } \textbf{psSize}; \end{array}$$

Here, $\mathbf{e_{min}}$ can be thought of as specified as an SLA in order to put a requirement into effect regarding efficiency of migration costs, which correlates to a certain latency $\mathcal{L}(\mathbf{x_{min}}) := \mathbf{l_{min}}$ where $|\mathcal{L}'(\mathbf{x_{min}})| = \mathbf{e_{min}}$ and $\mathcal{L}(\mathbf{x_{max}}) := \mathbf{l_{max}}$ where $|\mathcal{L}'(\mathbf{x_{max}})| = \mathbf{e_{max}}$. This way stakeholders can exercise their influence to avoid spending on unnecessary migration costs for improving a latency that is already acceptable for the intended application.

As regards recall, the efficiency-adaptive adaptation can be reformulated analogously: While improving on recall until $\mathbf{r_{max}}$, migration is intended to remain cost-efficient up to a stipulated minimal efficiency $\mathbf{e'_{min}}$:

$$(\mathrm{P}_{\mathrm{r/e}}) \quad \begin{array}{l} \textbf{if } \mathcal{R}(\textbf{psSize}) < \mathbf{r_{min}} \quad \textbf{then} \text{ increase } \textbf{psSize}; \\ \textbf{else if } |\mathcal{R}'(\textbf{psSize})| < |\mathbf{e'_{min}}| \quad \textbf{then} \text{ decrease } \textbf{psSize}; \end{array}$$

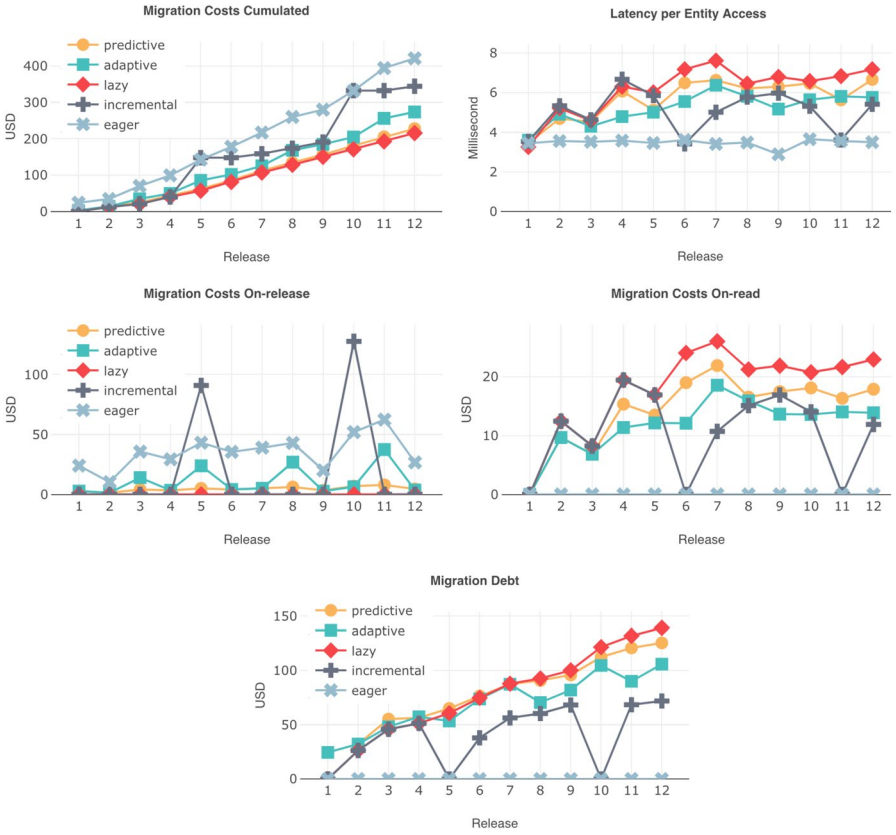


Fig. 6 Impact on the metrics for different migration strategies including the complexity-adaptive strategy (same migration scenario as in Fig. 3)

Here, a requirement can be stipulated in terms of efficiency with respect to recall $\mathcal{R}(\mathbf{x_{min}}) := \mathbf{r_{min}}$ where $\mathcal{R}'(\mathbf{x_{min}}) = \mathbf{e'_{max}}$ and $\mathcal{R}(\mathbf{x_{max}}) := \mathbf{r_{max}}$ where $\mathcal{R}'(\mathbf{x_{max}}) = \mathbf{e'_{min}}$.

We described in detail how adjustments to data migration strategies can be automatically adapted in order to fulfill all required constraints while suiting the migration scenario.

5 Evaluation of a self-adaptive strategy

In the preceding section, we discussed four approaches how migration strategies can be advanced to have self-adaptive capabilities. As a result, the complexity of software development can be reduced, because requirements are complied with automatically and do not rely on human intervention or reaction time. As a first step towards realizing all presented approaches of self-adaptation into our middleware *Darwin*,

we have implemented the *complexity-adaptive* approach. We can show already that this self-adaptive migration strategy has advantages and is favorable in respect of the characteristics summarized in Table 1, because it is a good compromise.

In Fig. 6, the *MigCast* simulation now also includes an *adaptive* migration strategy in the sense of the *complexity-adaptive* approach of Sect. 4.3. The investment of proactive migration costs through the increase in the prediction set size reduces the migration debt compared to the *predictive* approach. This can be observed, for instance, at release 11 of the charts, when on-release costs are invested with the *adaptive* strategy and the latency subsequently remains constant (release 12) avoiding a peak in latency happening with the *predictive* strategy, which is the a result of a backlog of complex schema evolution. In general, it can be observed, certainly from release 4 onwards, that the *adaptive* strategy is advantageous in this migration scenario compared to *predictive* migration with regard to on-read performance and latency. Peaks of long latency times are avoided systematically contributing to a more stable application performance.

This advantage of a mostly lower and relatively constant latency comes at a very fair price: Comparing the migration costs of all migration strategies, it can be concluded that the *adaptive* strategy distributes the incurring costs better between the releases but also between on-release and on-read costs at slightly more but reasonable cumulated costs (cp. with Table 1).

6 Related work

While there is a certain amount of related work that focuses on offline eager migration, for instance, Curino et al. [2] and Velegrakis et al. [16], there is very limited literature on other data migration strategies. Some approaches suggest lazy data migration for very large NoSQL databases, for instance, in Scherzinger et al. [13], and in Saur et al. [12], the overhead of lazy migration is discussed in terms of NoSQL databases. The foundations of different data migration strategies like incremental and predictive migration have been introduced in [6] and investigated in Klettke et al. [4]. Although workload monitoring can be applied for different tasks like automated database tuning [11], we are not aware of any approaches for automated selection and adaptation of these data migration approaches as has been presented in this paper.

7 Conclusion and outlook

In this paper, we have presented a methodology of self-adapting data migration, which automatically adjusts migration strategies and their parameters with respect to the migration scenario and SLAs. Our methodology takes various characteristics into account like the query workload to be handled, the number and kind of changes in the data model caused by schema evolution, and the requirements for the application specified in service-level agreements in terms of a reasonable compromise on

the tradeoffs between the cost metrics latency and migration costs and between the classification metrics precision and recall.

The resulting advantages of the concrete implementation of the *complexity-adaptive* strategy are so promising that we are currently working on the implementation of all other approaches into our middleware. Then, the methodology of self-adaptation can be compared in form of different configurations of adaptive migration strategies in order to eventually support a fully automatic database in respect of all possible migration scenarios. Complementing this, we have also been working on the investigation of the correlation between migration costs and latency and the variation of migration scenario characteristics in order to shed light on each of their impacts. Based on these quantitative results, the methodology of self-adaptation can be evaluated in terms of concrete performance effects in different migration scenarios. Based on these investigations, a heuristics can be distilled in order to support migration decisions by software project stakeholders, ultimately validating the heuristics and making the transition to a fully automatic database.

Funding Open Access funding enabled and organized by Projekt DEAL. This work has been funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 385808805.

References

1. Curino, C., Moon, H.J., Tanca, L., Zaniolo, C.: Schema Evolution in Wikipedia—toward a web information system benchmark. In: Proc. ICEIS'08, pp 323–332 (2008)
2. Curino, C., Moon, H.J., Deutsch, A., Zaniolo, C.: Automating the database schema evolution process. VLDB J. **22**(1), 73–98 (2013)
3. Herrmann, K., Voigt, H., Behrend, A., Rausch, J., Lehner, W.: Living in parallel realities: co-existing schema versions with a bidirectional database evolution language. In: Proc. SIGMOD'17, ACM, pp 1101–1116 (2017)
4. Hillenbrand, A., Levchenko, M., Störl, U., Scherzinger, S., Klettke, M.: MigCast: Putting a price tag on data model evolution in NoSQL data stores. In: Proc. SIGMOD'19, ACM, pp 1925–1928 (2019)
5. Hillenbrand, A., Störl, U., Levchenko, M., Nabiyev, S., Klettke, M.: Towards self-adapting data migration in the context of schema evolution in NoSQL databases. In: 2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW), pp 133–138 (2020)
6. Klettke, M., Störl, U., Shenavai, M., Scherzinger, S.: NoSQL schema evolution and big data migration at scale. In: Proc. SCDM'16, IEEE, pp 2764–2774 (2016)
7. Klettke, M., Awolin, H., Störl, U., Müller, D., Scherzinger, S.: Uncovering the evolution history of data lakes. In: Proc. SCDM'17, IEEE, pp 2462–2471 (2017)
8. Levandoski, J.J., Larson, P., Stoica, R.: Identifying hot and cold data in main-memory databases. In: Proc. ICDE'13, IEEE, pp 26–37 (2013)
9. Mellodge, P.: A Practical Approach to Dynamical Systems for Engineers. Elsevier, Amsterdam (2016)

10. Möller, M.L., Klettke, M., Hillenbrand, A., Störl, U.: Query rewriting for continuously evolving NoSQL databases. In: Proc. ER'19, Springer, LNCS, vol 11788, pp 213–221 (2019)
11. Mozaffari, M., Nazemi, E., Eftekhari-Moghadam, A.: Feedback control loop design for workload change detection in self-tuning NoSQL wide column stores. Expert Syst. Appl. 142 (2020)
12. Saur, K., Dumitras, T., Hicks, M.W.: Evolving NoSQL databases without downtime. In: Proc. ICSME'16, IEEE, pp 166–176 (2016)
13. Scherzinger, S., Klettke, M., Störl, U.: Managing schema evolution in NoSQL data stores. In: Proc. DBPL'13 (2013)
14. Störl, U., Müller, D., Tekleab, A., Tolale, S., Stenzel, J., Klettke, M., Scherzinger, S.: Curating variational data in application development. In: Proc. ICDE'18, pp 1605–1608 (2018)
15. Ting, K.M.: Precision and Recall. In: Sammut, C., Webb, G.I. (eds.) Encyclopedia of Machine Learning, pp. 781–781. Springer, Boston (2010). https://doi.org/10.1007/978-0-387-30164-8_652
16. Velegrakis, Y., Miller, R.J., Popa, L.: Mapping adaptation under evolving schemas. In: Proc. VLDB'03, Morgan Kaufmann, pp 584–595 (2003)