Evolving NoSQL Databases Without Downtime

Karla Saur
Intel Labs*
karla.saur@intel.com

Tudor Dumitraş and Michael Hicks
University of Maryland, College Park
{tdumitra,mwh}@umiacs.umd.edu

***Abstract*—NoSQL databases like Redis, Cassandra, and MongoDB are increasingly popular because they are flexible, lightweight, and easy to work with. Applications that use these databases will evolve over time, sometimes necessitating (or preferring) a change to the format or organization of the data. The problem we address in this paper is: How can we support the evolution of high-availability applications and their NoSQL data *online*, without excessive delays or interruptions, even in the presence of backward-incompatible data format changes?**

We present KVolve, an extension to the popular Redis NoSQL database, as a solution to this problem. KVolve permits a developer to submit an upgrade specification that defines how to transform existing data to the newest version. This transformation is applied *lazily* as applications interact with the database, thus avoiding long pause times. We demonstrate that KVolve is expressive enough to support substantial practical updates, including format changes to RedisFS, a Redis-backed file system, while imposing essentially no overhead in general use and minimal pause times during updates.

I. Introduction

NoSQL databases, such Redis [1], Cassandra [2], and MongoDB [3], are increasingly the go-to choice for storing persistent data, dominating traditional SQL-based database management systems [4], [5]. NoSQL databases are often organized as *key-value stores*, in that they provide a simple key-based lookup and update service (i.e., with "no SQL"). While these databases typically lack a formal schema specification, applications attach meaning to the format of the keys and values stored in the database. Keys are typically structured strings, and values store objects represented according to various formats [6], e.g., as Protocol Buffers ("Protobufs") [7], Thrift [8], Avro [9], or JSON [10] objects.

Database schemas change frequently when applications must support new features and business needs. For example, multiple schema changes are applied every week to Google's AdWords database [11]. Applications that use NoSQL databases also evolve data formats over time, and may require modifying objects to add or delete fields, splitting objects so they are mapped to by multiple keys rather than a single key, renaming of keys or value fields [12]. (These changes are similar in concept to relational database schema changes, but the lack of a formal schema allows for a wide variety of less strictly specified changes.) When changes are not compatible with the old version of an application, a straightforward way to deploy them in the field would be to shut down the running applications, migrate each affected object in the database from the old format to the new format, and then start the new version of the application.

High availability applications would prefer to avoid the downtime of shutdown-and-restart upgrades, but evolving a database on-line is challenging. Thrift, Protobufs, and Avro provide some support for format changes by allowing alteration of the data encoding itself or by tracking the version of an object's "schema" [13], [14], but there is still the task of updating each object in the database (e.g., by iterating over all of its keys [15]). For large amounts of data, this can create an unacceptably long pause. As an extreme example, Wikipedia was locked for editing during the upgrade to MediaWiki 1.5, and the schema was converted to the new version in about 22 hours [16]. Developers could avoid shutting down the application by making the new format backward-compatible with the old format, but this could impose a significant constraint on the future evolution of the application. It may also be possible to grant applications read-only access to the old database while the migration takes place, but applications that have even occasional writes will suffer.

A more general approach to evolve the database online is to migrate data *lazily*. When the updated application accesses an object in the old format, the object is converted to the new format on-the-fly. Thus, the long pause due to migrating the data is now amortized over the updated application's execution, causing slower queries immediately after the update but no full stoppage. Currently, the task of implementing lazy data migration falls on the developer: applications are rewritten to expect data in both old and new formats and to migrate from to the new format when the old one is encountered [12], [17], [18], [19]. This approach results in code that mixes application and format-maintenance logic. Since there is no guarantee that all data will ultimately be migrated, the migration code expands with each format change, becoming more confusing and harder to maintain.

To address these problems, this paper presents KVolve,[1] a NoSQL database that provides automatic support for online upgrades using lazy data migration. KVolve presents the *logical view* to applications that *data is at the newest version of the format*. Rather than convert all data at once, keys and values are converted as they are accessed by the application. Pleasantly, to use KVolve requires almost *no changes to application code*—they simply indicate the data version they expect when they connect to the database, and they are

*Work performed while at the University of Maryland.

[1]KVolve stands for *Key-Value store evolution*.

permitted to proceed if their expected version and the logical version match. When a data upgrade is installed, applications with an incompatible version must update themselves. They can do this with *dynamic software updating* (DSU) [20], [21], [22], [23], or by concurrent application switching (as in parallel AppEngine [24]) to avoid lost application state and/or shorten pause times, or by simple stop-and-restart (to the new application version). This is straightforward, in our experience, and need not be disruptive to end users. For example, customer-facing clients in web browsers can maintain session-permanence even as the backend servers, i.e., those connected to a KVolve DB, are upgraded. Such update patterns are common with load-balancing stateless servers [25].

KVolve triggers conversions automatically as data is accessed. To track its progress, KVolve attaches a version identifier to the value of each entry, converting only those keys/values that are out of date. Conversions are written by the developer. KVolve ensures updates are installed atomically in a way that supports fault tolerance. KVolve also automatically ensures that conversions take place atomically with the triggering database action; as such, KVolve avoids races that could clobber concurrent accesses. KVolve requires a conversion function to only access the corresponding old value/key, not *several* old key/values; to allow otherwise could violate logical consistency depending on the order that conversions are triggered. To support laziness, transformations to keys must be reversible and unambiguous. Examination of open-source software histories, and our own experience, suggests that realistic conversions typically satisfy these restrictions.

We describe a proof-of-concept implementation of KVolve as an extension to the popular Redis key-value store. We evaluate this implementation extensively, using both micro-benchmarks (the standard Redis performance benchmark) and macro-benchmarks (two feature-rich applications, redisfs and Amico). Our experiments suggest that KVolve imposes essentially no overhead during normal operation and that complex applications can be upgraded with zero downtime. In particular, when upgrading redisfs we used KVolve to upgrade the filesystem data, and Kitsune [20], a whole-program updating framework for C, to dynamically update the redisfs driver. As a result, we could seamlessly maintain the file system mount point during the upgrade, resulting in zero downtime.

In summary, we make three contributions:

- We identify the challenges for evolving NoSQL databases without downtime (Section II) and, to our knowledge, we propose the first general-purpose, automatic solution to this problem (Section III).
- We describe a proof-of-concept implementation as an extension of the Redis key-value store (Section IV).
- We evaluate this implementation extensively, and we show how to combine KVolve with a dynamic program updating tool for zero-downtime upgrades (Section V).

II. The problem with on-line upgrades

This section details the problem of updating a NoSQL database on-line, and the drawbacks of prior solutions. Our approach, KVolve, is detailed in the next two sections.

A. NoSQL DBs and KV stores

NoSQL databases distinguish themselves from traditional *relational database management systems* (RDBMSs), by supporting a simple, lightweight interface. Our focus is on a NoSQL variant referred to as a *key-value (KV) store* which, as the name implies, focuses on mapping keys to values. There are two core operations: GET k, which returns the value v to which k maps in the database (or "none" if none is present); and SET k v, which adds (or overwrites) the mapping $k \rightarrow v$ in the database. Example KV stores include Redis (the most popular [1], and the target of our proof-of-concept implementation), Project Voldemort [26], Berkeley DB [27], and many others [28].

While a KV store may place no formatting requirements on values (i.e., treating them as bytearrays), applications typically store values adhering to formats such as JSON [10], Avro [9], or Protobufs [7]. Some KV stores do expect a specific value format; e.g., Cassandra defines typed "rows" in "tables," and MongoDB employs "documents." Likewise, key formats may be unstructured (i.e., just strings) or have some structure added by the system (e.g., a notion of *prefix*, or *namespace*).

B. Example application and update

As an example (adapted from Sadalage and Fowler [12]), consider an on-line store which keeps track of purchase orders. The application stores these orders in a KV store, using keys of the form order:n, where n is a unique invoice number, and values formatted as JSON records describing the purchasing customer and what was ordered. In this key, order is a prefix to assist in key grouping, e.g., as part of the encoding of a table. An example JSON record is shown in Figure 1(a).[2]

Suppose we wish to upgrade the application to support differentiated pricing, which necessitates changing the data format in the KV store. Keys remain the same, but values change: we rename the field price to fullprice, and insert a new field named discountedPrice that is a possible reduction of the original price. The updated orderItems array (the last element of the JSON object) is shown in Figure 1(b).

C. Past approaches to on-line data upgrades

Eager, stop-the-world data upgrades: One approach for implementing the data upgrade described above is to simply halt all client applications and use a script to convert all the data in the KV store that is out of date. Once all the data is updated, the clients can be restarted. For our example, the conversion script would get each purchase order value, modify

[2]JSON defines four primitive types: numbers, strings, booleans, and null. It also defines two container types: arrays, which are an ordered list of values of the same JSON type; and objects, which are an unordered collection of values of any JSON type, with field labels. We use JSON as an example only; other formats are also supported by KVolve.

```
{ "_id": "4BD8AE97C47016442AF4A580",
  "customerid": 99999,
  "name": "Foo Sushi Inc",
  "since": "12/12/2012",
  "order": {
    "orderid": "UXWE−122012",
    "orderdate": "12/12/2001",
    "orderItems": [
       { "product": "Cookies",
         "price": 19.99 }
    ]
} }
```

(a) original format

```
"orderItems": [
  { "product": "Cookies",
    " fullPrice ": 19.99,
    "discountedPrice": 16.99 }
]
```

(b) updated format

Fig. 1: JSON object, and update

it, and store back the updated result. In particular, for each existing purchase order value, the script would replace the existing elements of the orderItems array with new elements whose price field is renamed fullPrice, and which contain with a new discountedPrice field initialized to the old price value.

While simple, the downside of this deployment strategy is the disruption in service while the database is being upgraded. Our experiments show that for even modest-sized databases (hundreds of thousands of keys), this disruption can be on the order of several minutes. On a larger scale, Twitter has deployed 6,000 instances of Redis at each of many data centers, with instances using 40TB heaps [29]. Any update to that data of that size will be very time consuming.

Manual, lazy data upgrades: Rather than pause service while the database is being *eagerly* upgraded, Sadalage and Fowler suggest that the programmer can modify the new version's code to handle old and new formats, and migrate old data *lazily*, when it is encountered. For this example, the application could try to access the fullPrice field of a purchase order's orderItems array. If that field is not present, the application can update the value as described above, and then try again.

This approach works but adds a greater burden on the programmer, who must add the version checking code, and code to upgrade out-of-date values. Such upgrades imply that what once was a *GET may now involve an additional SET* to the updated format. As we explain in the next subsection, this added operation could result in data-corrupting race conditions. Worse, version checking/updating code will grow over time, as mentioned in Section I, and will become more complex as applications expand to deal with a variety of data values. The end result is a significant maintenance headache.

III. KVolve

KVolve aims to solve the on-line upgrade problem in a way that enjoys the best features of lazy and eager data upgrades. In particular, KVolve migrates data lazily, as it is accessed by applications, thus eliminating any long, disruptive pause. But KVolve presents a *logically consistent view* to applications, providing the *appearance* that all the data is instantly upgraded to the new version. As a result, programmers do not need to

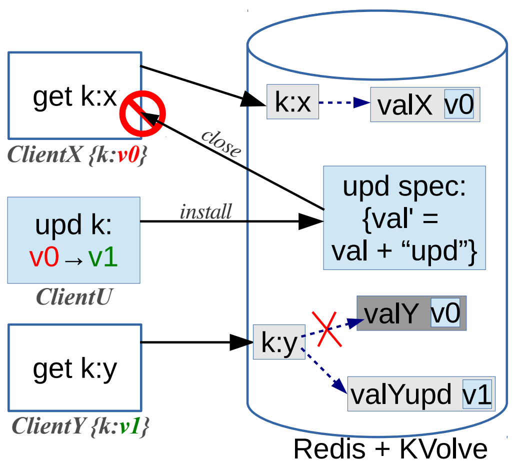


Fig. 2: KVolve architecture.

add any version-management code to their applications; they simply write the application assuming the most recent data version. Because the lazy migration is handled by KVolve, it can ensure there are no errors due to concurrent interactions.

A. KVolve design

Our approach is characterized by three techniques.

Versioned data: We associate logical version identifiers with the database content. Rather than having a global data version, we track separate versions for data associated with different key prefixes. E.g., data mapped from keys p_1:x (for all x) has a separate version ID space than data mapped from keys p_2:x (for all x). A version tag is stored with each data item indicating its actual version, which might be earlier than its logical one (i.e., if the data item has not been migrated yet). Version tags are invisible to applications accessing the database. When the application connects to the database, it indicates the version IDs of the key prefixes it will use, and KVolve compares them to the logical versions of those prefixes. If the two IDs match, KVolve accepts the connection.

Update specifications and state transformer functions: When the database is to be upgraded, the operator installs a specification describing the mechanics. In particular, the specification defines the new logical prefix versions, and provides *state transformer* functions to be used to upgrade particular values. Each transformer function f is associated with a key prefix p. If a key of the form p:x (for some x) maps to a value v, then p:x will be updated to map to $f(v)$. KVolve can handle key format changes, too, as discussed below.

On-demand (lazy) transformation: Once the update specification is installed, applications connected to the database that are out of date must be disconnected. They will reconnect at the new code version (mirroring the situation with eager upgrades). Doing this is not onerous for most applications as discussed in Section IV-B.

Once a new application version starts running, it will submit GETs and SETs to KVolve for handling. If a GET accesses a value that is out of date, KVolve first updates the stale item using the appropriate transformer function. If a data item is several versions out-of-date, transformer functions will be composed and applied automatically.

We illustrate these three techniques in Figure 2. Here, *ClientX* initially connects at version *v0*, and is able to access the value mapped to from k:x safely, since it is also at *v0*. Then, *ClientU* updates the database to version *v1*, and includes a state transformer function for prefix k. This function concatenates the string "upd" to an existing key's value. This update causes *ClientX* to be disconnected because its version *v0* is now inconsistent with the database's logical version *v1*. Finally, *ClientY* connects to the database at version *v1*. It performs a GET on key k:y. This key maps to a stale value, having version *v0*. Therefore, KVolve remaps the key to a the value produced by running the transformer function on the old value. Then it returns the updated value to *ClientY*.

B. Ensuring logical consistency

KVolve's goal is to provide a logically consistent view to applications. That is, any sequence of commands issued by up-to-date clients should produce the same results whether interacting with a fully (i.e., eagerly) updated database or with one whose data is being migrated lazily, as it is accessed. This goal imposes three requirements on KVolve's implementation. First, state transformations must occur atomically with the operation that induced them. Second, transformer functions may only reference the old version of the to-be-updated key/value, and key changes are restricted. Third, the update specification must persist once it is installed, so that logical consistency is maintained following recovery from a fault.

Atomicity: Upgrading data atomically ensures that clients accessing the data concurrently with the lazy transformations will not cause anomalies that cannot occur with eager upgrades or during normal operation. To see how an anomaly could be introduced, consider a trace with a GET k:x by client A and a SET k:x w by client B. Suppose that k:x maps to v in the old database, and the update's transformer function f operates on a key's old value to produce the new one.

In an eager update, k:x's value v is updated to be $f(v)$. Then there are two possible execution schedules: client A could retrieve $f(v)$ and client B could set k:x to w, or client B could do the set, in which case A returns w (which is already up to date). In both cases, the final database maps k:x to w.

In a lazy update, a transformer must be invoked before returning the value to A. One way to implement this would be to convert client A's GET into two commands when dealing with an out-of-date value: SET k:x $f(v)$ (i.e., set it to the updated value) and then GET k:x (i.e., return that transformed value $f(v)$ to the client). But in this case, client B's SET could be scheduled in between client A's SET and GET. This would result in A's GET returning the w SET by B, but then A's SET overwriting w with $f(v)$. This final state would never be possible in the eager case. Effectively, improperly implemented lazy updates could cause client B's operations to fail silently, without notifying B of the failure. This anomaly violates logical consistency. In contrast, this scheduling is not possible if client A's read and update must always be atomic. One of KVolve's benefits, over by-hand modification of code to support lazy migration, is that it can ensure atomicity automatically.

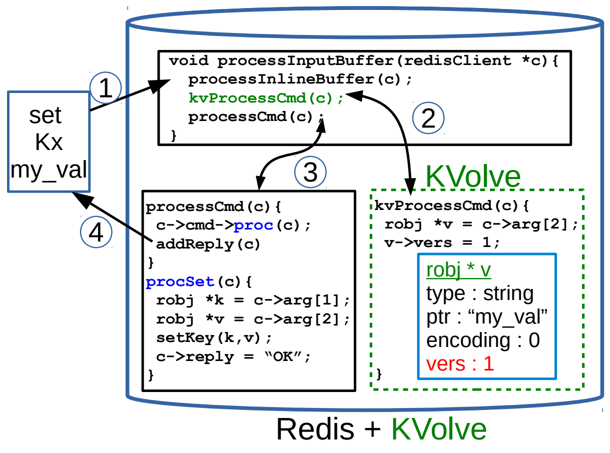
Limited domain of transformer function: KVolve restricts transformer functions f in two ways. First, the transformer may only operate over the old version of the key/value it is updating, and not any other items in the database. Second, the transformer may not change keys arbitrarily; instead it only supports unambiguous bijections on a key's prefix.

The first restriction ensures that when f runs, it will operate on the same keys/values it would have if run when the update was installed. This is because only the first GET of a key could possibly see a stale value, and it will immediately update it. As such, it ensures a logically consistent view. On the other hand, if we allowed f to access other data items, it is easy to see how logical consistency is broken. For example, suppose the function f to update a key k:x's value also examined m:x's value v. If the new-version code executed a SET m:x w prior to a GET k:x, then f would read w, not v.

The second restriction ensures that lazy key updates can be implemented safely and efficiently. After an update is initiated, the new application version will issue commands using the *new* keys. For example, suppose an update changes the prefix from k to m:j, so that keys k:n would become m:j:n (for all n). After the update, applications will submit commands like GET m:j:n. If the key is present, we need to be sure that it is a new-version key, not an old one that has yet to be transformed; as such transformations may not map to key prefixes that are also present in the old database version. On the other hand, if the key m:j:n is not present, KVolve should look for the *old* version of the key, in case it is there and thus needs to be updated. To do this, KVolve will have to run the transformation backwards, i.e., on m:j:n to produce k:n. Limiting transformations to key prefixes helps make backward transformation efficient, since KVolve can match keys against (new-version) prefixes directly.

Restricting transformation functions in this way is conceptually limiting, but not practically so, we believe. We analyzed 18 of the most active projects on GitHub that used Redis to store program data, and none of the programs contained value changes that were dependent on other value changes, and key changes were limited to prefix changes.

Fault tolerance: Many KV stores provide fault tolerance guarantees; i.e., there is a way to checkpoint the database so that it can be recovered after a crash. As such, if the database crashes during a lazy upgrade, KVolve's implementation should ensure the logical view is retained following recovery. KVolve ensures this by (a) storing per-data version tags in the database, so they are made persistent; and (b) storing the update specification (and logical version) in the database, atomically, when the update is installed. This way, if the database crashes before the update is fully installed, then on recovery the database will still appear (correctly) to be at the old version. But once the update is fully installed, then the database identifies as being at the next logical version, and lazy migration can pick up where it left off after recovering from a failure.



```
void processInputBuffer(redisClient *c){
  processInlineBuffer(c);
  kvProcessCmd(c);
  processCmd(c);
}
```

```
processCmd(c){
 c->cmd->proc(c);
 addReply(c)
}
procSet(c){
 robj *k = c->arg[1];
 robj *v = c->arg[2];
 setKey(k,v);
 c->reply = "OK";
}
```

```
kvProcessCmd(c){
 robj *v = c->arg[2];
 v->vers = 1;
}
```

Fig. 3: Workflow for Redis and KVolve

IV. Implementation

This section describes our implementation of KVolve as a modular extension to the popular Redis key-value store.

A. *KVolve implementation overview*

KVolve is implemented as a separate library compiled into Redis. It works by preprocessing commands coming in from the client before passing them along to Redis, as depicted in Figure 3. In Step 1, the client issues the command, e.g., SET Kx my_val. In Step 2, kvProcessCmd, KVolve's hook, is called to preprocess the command (the dashed green box is the KVolve library). Once the KVolve preprocessing is complete (which might involve changes to data's contents and version field), control returns to normal Redis. In Step 3, Redis's processCmd function calls the function pointer shown in blue (which depends on the choice of command—here it is procSet because the client requested SET), and this adds the affected object to the database, including any changes to the version field set during KVolve's processing. Finally, in Step 4 Redis responds to the client's request, acknowledging to the client that it successfully executed the SET command.

All of this is sure to be atomic because Redis is single-threaded: it processes each command it receives in its entirety before moving to the next. Redis provides commands, such as multi, that can be used to execute a group of commands atomically; KVolve's design works in concert with such commands. We also believe that KVolve's basic "interceptor" architecture would work in multi-threaded KV store implementations by employing appropriate synchronization.

B. *Describing and installing updates*

An update consists of *transformer functions* that will convert the old version of a key and/or value to the new version. The programmer compiles the transformer functions into a shared object file that she can direct KVolve to install (using a repurposed Redis command). After installation, the shared object and metadata about it are stored persistently in Redis, meaning that the specification is restored in case of a crash.

There are two kinds of updates: key/value updates and key updates (only). As an example of the former, Figure 4 shows a transformer for the example from Figure 1. The old key (a string) and value (binary data) are passed in by reference, and the function will update them to the new versions via these references. In this case, the body of the function uses the Jansson library [30] to implement the change to the purchase order example from Figure 1 described in Section II-B; the last two lines update the value (the key is not changed). Writing this code is a bit tedious. As done in our prior work [20], [31], [32], we could easily implement a domain-specific language to simplify the process.

```
void test_fun_updval(char ** key, void ** value, size_t * val_len){
  json_t *root = json_loads(*value);
  json_t *arr = json_get(json_get(root, "order"), "orderItems");
  // iterate over the {product: ... , price: ... } array entries
  for(i = 0; i < json_array_size(arr); i++){
     // Set discountedPrice to (price − 3)
     // Set fullPrice to price, then delete price
  }
  *value = json_dumps(root); // Set the updated value
  *val_len = strlen(*value); // Set the updated val length
}
```

Fig. 4: Example pseudocode transformer function for JSON

Along with the transformer functions, an update specification contains a function that is invoked when the shared object is loaded as part of an update. This function consists of a series of calls to install transformer functions. Our example above is installed by the following call:

```
kvolve_upd_spec("order","order", 0, 1, 1,test_fun_updval);
```

This call indicates that the order prefix doesn't change, from version 0 to version 1, while the test_fun_updval should be called for each key with the prefix order.

Key prefixes can be changed without requiring a transformer function. For example, in the Amico program described in Section V-C, the keys are renamed from the prefix amico:followers to the prefix amico:followers:default. To describe this update, the initialization function would include the call:

```
kvolve_upd_spec("amico:followers", "amico:followers:default",1, 2, 0);
```

where the version numbers are 1 and 2, and the final 0 indicates that there are no functions to manipulate the value.

KVolve will close the connection to all clients using the old version of updated prefix(es). A disconnected client will not be allowed to reconnect until it is upgraded to the new version. Clients not using updated prefix(es) will not be affected. To use KVolve, therefore, processes connecting to KVolve must be coded to support disconnection, upgrade, and restart.

KVolve stores update specifications indefinitely. We find that the transformer functions take up a small amount of space relative to the rest of the data. However, if program updates are very large or very frequent, one could employ a background client or similar thread to force updates to outdated data by GETting them all; once done, all update information could be freed for that version. (This would essentially be a hybrid of the lazy and eager approach.)

C. Key lookup

After an update is installed, the database's logical version is advanced. Because the new version might transform the format of keys, KVolve may need to look up the *old* version of a key specified in an application's GET or SET commands, so that it can update that key (as per Section III-B). To support this, we use a *update information hash table* (UIHT). This table maps a key prefix to a record which contains both that same prefix and pointers to records that describe the next and/or previous versions of the prefix. For example, after a key update from foo: to foo:bar:, the table would map prefix foo: to a record q whose next pointer would be to a record r for foo:bar:, which points back to q; the table would also map prefix foo:bar: to r. As such KVolve can trace through all current and former versions of a prefix for applying updates. The table's records also contain transformation functions for moving forward between versions, and track the IDs of client connections that are using a particular prefix version, so they can be disconnected on an update to it.

After a key update, client queries will use the new key format; e.g., after updating prefix foo: foo:bar:, client commands will refer to keys foo:bar:n. KVolve will first look to see if a key exists under the issued name. If it is not there, an old, non-updated version of the key may be present. As such, KVolve looks up foo:bar: in the UIHT to see if a record is present that maps to an old prefix. In this case, KVolve will see that prefix foo:bar: points back to prefix foo:, so it reissues the client command with key foo:n instead. If it finds it, it updates the key name to foo:bar:n and returns the value to the client. If no match is found, it will continue to follow backpointers in the UIHT if prior versions should be considered. If none are found, meaning that no key is present, KVolve returns control to Redis without further action.

Looking for keys under previous prefixes adds additional lookups only once (during the update) when the key is present under an old version. However, the case where there is no key present under any prefix version will add unnecessary additional lookups each time the non-present key is queried. An application that frequently queries keys that are not present where there has been a prefix change could negatively impact performance. In previous work [32], we experimented with adding a *sentinel value* to mark the key as absent, skipping the step of checking for the key under previous prefixes, thus saving on lookup time but adding a bit extra storage.

D. Getting and setting values

In the simplest case, keys map to string values. We consider this case first. Our implementation currently supports 36 Redis commands and all of the main Redis data structures (string, set, list, hash, sorted set); we discuss containers in the next subsection. We focused on implementing commands that modify data. The majority of Redis' commands that we did not implement do not impact updating the data (e.g. commands related to networking such as the pub/sub functionality or connectivity).

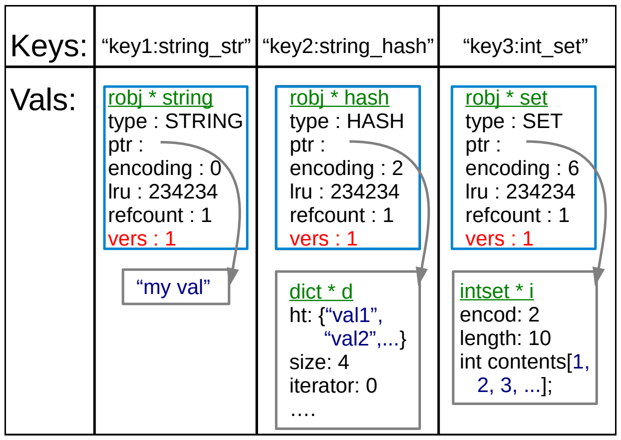


Fig. 5: Storing different data types

GET: If the client request involves getting a string, KVolve must first prelookup the existing robj value structure in the database to get the version information. An example of the key-value pair for string types is shown for key1:string_str in the first column of Figure 5. This action retrieves a pointer to the actual object structure that is stored in the database, so any modifications that KVolve makes to this object will be automatically stored in Redis. Note that this requires an additional database lookup by KVolve, on top of the one that Redis will do later when it does its own lookup to handle the client request. However, this is an O(1) operation and does not incur excessive overhead relative to the other operations that KVolve must already perform.

If the version field of the robj (in this example the version is 1 shown in red for key1:string_str in Figure 5) is current for the prefix of the key, or if the key is not present under the current or any former prefix (and therefore no robj exists for the key), then KVolve returns control to Redis and does no further processing. If the version is not current, either in the current prefix or a former prefix, KVolve will update the key and value, as specified. All of the necessary information to perform the update (the transformer functions themselves, and the meta-data about which prefixes and versions the updates apply to) is stored in the update information hash table, and KVolve uses that information to apply the update as follows:

- In the case of a key prefix change, KVolve uses the update information from the hash table to perform the key rename, leaving the value untouched.
- In the case of a value change, KVolve calls any applicable user-supplied functions and applies them to the value, starting from the oldest needed update and working forward to the current version. After all of the transformer functions have been applied, KVolve stores the updated value in the robj (which is a pointer to the actual structure stored in the database) and updates the version string to match the current version by setting the field in the robj.

If both actions (key prefix and value change) are necessary, KVolve will perform both. KVolve then returns control flow to Redis, and when Redis performs its own GET, it will retrieve and return the newly updated key to the client.

SET: If the client request involves setting a string, KVolve first checks to see if the request has any flags that would prevent the value from getting set. These flags, XX or NX, respectively specify to only set the key if it already exists, or only set the key if it does not already exist. If necessary, KVolve does a lookup in the database to determine if the key exists, indicating if the value will be set for the requested key. (As described in Section IV-A, if the prefix changes, KVolve will search for the key under the old prefix to see if it exists.) If the value will not be set due to the flags, KVolve does nothing and returns control to Redis. In this SET command, or any such command where Redis will be adding the robj to the database, Redis deletes the old robj and replaces it with the new one from the client's request. Therefore, all that KVolve must do is set the most current version string in the robj for the prefix of the key. (Remember that there is no need to attempt to update the value in the key, because the client's provided value is guaranteed to be at the up-to-date version.)

If this set occurs after a key prefix change, KVolve must delete the old value for the key to ensure that deprecated key versions are not unnecessarily bloating Redis. For example, in a change to redisfs (presented in Section V-B), an old key prefix was named skx:/ but after an update, the new name postfixes DIR such that the key is now named skx:DIR:/. If the user were to set the key skx:DIR:/root before getting (and updating) it, this would leave the old key skx:/root still in the database. Therefore, KVolve must check to see if the existing version under the old key prefix exists, and if it does, delete it. It does this by first checking if the prefix had any previous changes. If not, it does nothing. If so, it checks and deletes the old key if necessary. At this point, KVolve returns control to Redis, and Redis adds the robj structure to the database, which also contains the updated version string to be retrieved later if necessary.

E. Sets, hashes, lists, and sorted sets

The other Redis value data structures are containers of sub-values. The base of Redis containers are all robj structures, and they store the actual data. Figure 5 shows examples in columns two and three of robjs that contain a hash of strings and a set of integers, respectively. KVolve stores version information in the container, not in the contained values (to avoid more pervasive changes to Redis), so updates to containers happen all at once.

The process for doing a GET or SET on one of the container elements is the same as for the string type described in Section IV-D, except that if an update is necessary then all sub-elements are updated using a Redis-provided iterator.

V. Experimental results

This section considers the performance impact of KVolve, during normal operation and during an update. Our experimental results are summarized as follows:

- Using the standard benchmark that is included with Redis, we found that KVolve adds essentially no overhead during normal operation, and we determined that storing the version and update information in Redis adds only about a 15% overhead in space.
- We updated the redisfs file system which included renaming some keys and compressing some data stored in keys, and found the operating overhead to be in noise, and the pause time to be close to zero as opposed to 12 seconds for an offline data migration.
- We updated the Amico social network system and found no added overhead, with a pause time of close to zero as opposed to 87 seconds for an offline data migration.

In our experiments, we worked with read loads because they are the worst case, as is this the case where the lazy update takes place. In the write case, the old data is simply replaced by the new data, which is guaranteed to be already up-to-date due to version checking.

All experiments were performed on a computer with 24 processors (Intel(R) Xeon(R) CPU E5-2430 0 @ 2.20GHz) and 32 GB RAM with GCC 4.4.7 on Red Hat Enterprise Linux Server release 6.5. All tests report the median of 11 trials, and communication was via localhost with ~.03 ms latency.

A. Steady state overhead

First we report the steady state overhead for KVolve reported by Redis's included benchmark, *Redis-bench*. Redis-bench acts as a client that repeatedly issues commands to Redis. The default settings for Redis-bench are with 50 clients, with 10,000 repetitions of a single operation at a time (only 1 request per round trip), and with a single key (getting or setting a single key multiple times). However, Redis-bench allows many different configurations. For a longer benchmark, we increased the number of operations to 5 million operations and for a more realistic benchmark we performed these operations over 1 million keys, leaving the rest of the default settings alone. We ran this experiment over localhost which had a latency of ~.03ms. We chose three types of GET operations (string gets, set pops, and list pops) and three types of SET operations (string sets, set adds, and list pushes), as these were part of the default benchmark operations test suite.

Table I shows the steady state overhead of this experiment. We show unmodified Redis in column 3 for comparison and broke the overhead into separate categories: KVolve with no prefixes to update declared (causing KVolve to return immediately for each key) in column 4, KVolve with a single prefix declared (causing a hash lookup and a version check for each key) in column 5, and KVolve with a previous prefix declared but no previous keys with the old prefix (causing a hash lookup, a version check, and a string concatenation to look for a non-existent previous key) in column 6. Each sub-column of Table I shows the total time for the test, the siqr (Semi-Interquartile Range) to show the variance, and the overhead as a comparison against unmodified Redis. We ran this benchmark many times with various configurations (multiple key prefixes to track, less or fewer keys, less or fewer clients, etc) and found that the overhead varied generally around $\pm 3\%$, with no consistent pattern between any of the tests, even repeated tests with the exact same setup. The

TABLE I: Redis-bench for Redis vs KVolve (times in seconds, median of 11 trials)

		Redis		No NS, KVolve			With NS, KVolve			&Prev NS, KVolve		
		time	siqr	time	siqr	OH	time	siqr	OH	time	siqr	OH
With single instructions	String Get	58.83	(0.25)	58.08	(0.11)	**-0.77%**	58.46	(0.26)	**-0.12%**	59.51	(0.85)	**1.67%**
	String Set	63.52	(1.04)	63.66	(0.70)	**0.22%**	65.39	(1.49)	**2.49%**	64.82	(0.35)	**2.05%**
	List Pop	58.47	(0.41)	58.93	(0.68)	**0.79%**	58.71	(0.15)	**0.41%**	59.34	(0.47)	**1.49%**
	List Push	59.49	(0.66)	59.55	(0.56)	**0.10%**	60.02	(0.74)	**0.89%**	60.87	(0.94)	**2.32%**
With 10 *pipelined* instructions	String Get	9.73	(0.30)	9.75	(0.27)	**0.21%**	9.96	(0.25)	**2.36%**	9.93	(0.20)	**2.06%**
	String Set	13.77	(0.26)	14.56	(0.18)	**5.74%**	14.56	(0.32)	**5.74%**	14.48	(0.33)	**5.16%**
	List Pop	9.60	(0.32)	9.71	(0.25)	**1.15%**	9.55	(0.43)	**-0.52%**	9.63	(0.27)	**0.31%**
	List Push	14.22	(0.39)	14.38	(0.25)	**1.13%**	14.40	(0.41)	**1.27%**	14.48	(0.36)	**1.83%**

TABLE II: Max resident set size (RSS)

Program	**Max RSS**
Redis, empty	7.7MB
Redis, 1M 10-byte values	112.1MB
KVolve, empty	7.7MB
KVolve, 5 prefixes, 1M 10-byte values	128.6MB

numbers presented in the table show some negative and some positive overhead, reflecting this variation. Notice that the siqr numbers show that the variance is relatively high, as high as 1.49s for setting strings with KVolve and a prefix, shown in the fourth row of the fifth column.

The bottom half of Table I shows a modification of the original overhead experiment, using a pipeline to feed 10 instructions into each round trip to Redis-bench over localhost. This reduced the I/O overhead, putting more emphasis on KVolve operations. We found that these numbers showed a bit more overhead, and allowed us to bound the overhead at 5.74% for 10 subsequent pipelined instructions. This test demonstrated that although there is some overhead added by KVolve, for the non-pipelined version and most commonly-used scenario (Table I), the overhead is mostly buried in I/O and very low overall. (In our test programs, described next, Amico pipelined at most 3 instructions per round trip, and redisfs did not use pipelineing.)

In addition to time overhead, KVolve incurs some additional memory overhead due to tracking the version information. Table II shows the maximum resident set size as reported by *ps*. Empty, Redis and KVolve take up about the same amount of size in memory. With 1 million keys each mapping to 10-byte values and with 5 separate prefixes declared, KVolve takes up about 16.5MB (~15%) more memory than unmodified Redis. This includes the extra version field (4 bytes) on each value structure, the amount of space it takes to store the version lookup information and hash table, and any extra padding that may be automatically added to the additional structures.

B. Redisfs

Redisfs [33] uses Redis as the backend to the FUSE [34] file system. The inode information, directory information, and all file system data are stored in Redis. On startup, FUSE mounts a directory with Redis as the backend, and a user can perform all of the normal operations of a file system, with the data silently being stored in Redis. Redisfs has 8 releases, ~2.2K lines of C code each. In redisfs.5, released March 4th, 2011, file data is stored in a Redis as a binary string with no compression, and the directory keys have the format skx:/path/todir. In redisfs.7, released March 11th, 2011, file data is compressed using zlib, and directory keys have the format skx:DIR:/path/todir. (Note that redisfs.6 contained an error and was retracted, so we use versions .5 and .7.) This change makes it impossible to view the directories or any of the files using redisfs.7 for any files created using redisfs.5.

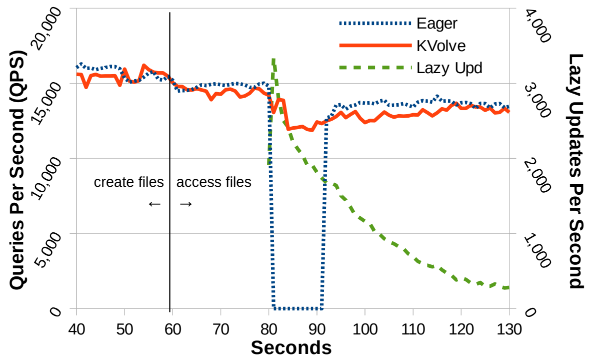


Fig. 6: Lazy vs. eager updates for RedisFS

In all versions, the inode data is stored across 12 Redis keys including meta information such as modification time and file size. All file system information is represented in redisfs with four prefixes: the skx:/ prefix for directories (which is updated to skx:DIR/ in redisfs.7), the skx:NODE prefix for inodes (some of which is updated to add compression in redisfs.7), skx:PATH for paths to directories, and skx:GLOBAL to track internal structure; the last two are not updated. To make redisfs compatible with KVolve, we added only 6 lines of code in both versions which consisted of an additional call to Redis on start-up to declare that we would be using those 4 prefixes at either version .5 or .7, along with a few additional lines of error handling.

We performed an update from redisfs.5 to redisfs.7, both by migrating the keys offline (referred to as the Eager version), and with KVolve to automatically rename the directory keys as they are accessed and to add compression to the files as they are accessed. In addition to updating redisfs with KVolve, we also used Kitsune [20], whole-program update software for C, to allow us to also dynamically update redisfs along with the data so that the users experience no downtime; the switchover from .5 to .7 is completely seamless. Normally, killing redisfs.5 and restarting at redisfs.7 also causes the mount point to be unmounted then remounted (causing the user to have to switch back into the mounted directory after remount), but with Kitsune, the mount point is not disrupted during the switchover. We used the file system benchmark PostMark [35] to generate a workload for redisfs, creating an initial 10,000 files ranging from 4-1024 bytes in 250 subdirectories plus the root directory, for a total of 251 directories. We ran PostMark outside the root directory mount point, accessing the files via full path name to avoid having to change directories due to the restart for the Eager (non-KVolve/Kitsune) version.

Figure 6 shows the results of the redisfs experiment. After about 60 seconds, PostMark switched from creating the new files to reading from or appending to existing files. As shown on the *left* y-axis, both KVolve and the Eager version had a very similar average Queries Per Seconds (QPS), displayed by the solid and finely dashed lines. At 80 seconds, we killed redisfs.5. For KVolve, we used Kitsune to dynamically update to redisfs.7 without pause, maintaining the mount point so that the benchmark never lost access to the files or the directory structure, and KVolve continued to process queries throughout the update. For the Eager version, we halted all traffic to Redis and migrated the data, performing the renames and compression as necessary. In this update, not all of the keys needed to be updated, only the 251 directory keys that needed to be renamed and the 10,000 data keys that needed to be compressed. However, the database contained 123,002 total keys, and the to-be-updated keys were searched for in the database, adding to the pause time. This offline update process took about 12 seconds, as shown in Table III.

In addition to showing the QPS lines, the green widely-dashed line in Figure 6 shows the number of lazy updates per second for KVolve, corresponding to the *right* y-axis. Immediately after the update, this number burst to ∼3K keys per second, and quickly trailed off as keys were lazily updated. KVolve renamed the 251 directory keys, updated the version on all 112,752 keys in the skx:INODE prefix, and compressed the data for the 10K keys in that prefix that contained file data.

Overall the impact on the update experienced by redisfs was minimal, as the QPS dipped only slightly right after the update before it quickly returned to full speed around the 120 second mark of the experiment. After the update, the overall QPS was slower for both KVolve and redisfs because the files must be compressed and decompressed as they were accessed.

C. *Amico*

Amico [36] maps relationships in the style of a social network, defining a set of users and the relationships between them. Amico provides an API that allows queries over a data set of users: a user may be following or be followed by any number of other users. Amico is backed by Redis, has 10 versions created between 2012-2013, and is written in ~200 lines of Ruby code. Amico version 1.2.0, released Feb 22, 2012, stores these relationships in 5 different types of Redis keys with the following prefixes: amico:followers, amico:following, amico:blocked, amico:reciprocated, and amico:pending. In version 2.0.0, released Feb 28, 2012 (the next consecutive version after 1.2.0), the developers added the concept of a

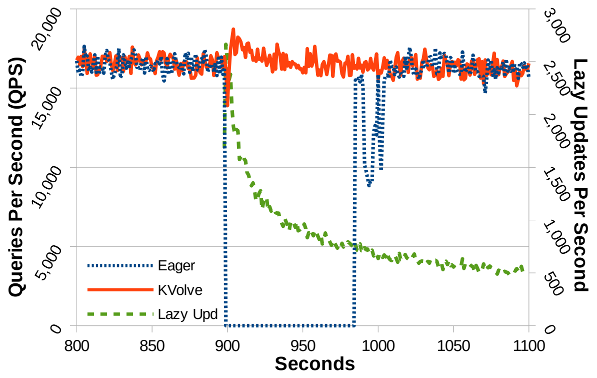


Fig. 7: Lazy vs. eager updates for Amico

"scope" so that there could be different graphs stored in Redis with prefixes to keep them separate, such as "school" network and a "home" network. The default name for the scope is "default", such that all of the keys are prefixed with amico:followers:default for example. This change makes databases created with Amico 1.2.0 incompatible with Amico 2.0.0. To make Amico work with KVolve, we only changed the same 4 lines of code in each version to declare the prefixes right after Amico connects to Redis.

For this experiment, we used the LiveJournal data set from the SNAP [37] library. The LiveJournal data set has 4,847,571 nodes and 68,993,773 directed edges defined by ordered node id numbers *A follows B* such as 186032 2345471, which we shuffled into two separate files for reading in a random order. To create a workload, we started two programs with calls to Amico 1.2.0: one program to read from the first random file and add nodes to the Amico network, and one program to read from the second random file and perform queries over nodes in the network such as querying if USER A followed USER B or querying the number of followers of USER A. After letting the programs run for 900 seconds (15 minutes), the Redis database was filed with 792,711 keys containing nodes and edge data.

At the 900 second mark, as shown in Figure 7, we stopped both of the Amico 1.2.0 programs. For the Eager case (finely dashed line), we then updated all 792,711 keys, renaming them to have the default scope prefix in all of the key names. This migration took ∼87 seconds as shown in Table III. In addition to the pause, the Eager case shows a continued disruption until around the 1,000 second mark. After the migration was complete, we started the same writer/reader programs, this time using Amico 2.0.0. For the KVolve case (solid line), we immediately started the two Amico 2.0.0 programs after the update so that the keys could be lazily migrated. Right at the update point, there is a ~2K drop in the QPS (*left* y-axis), before a brief spike and a return to the original rate. The widely-dashed green line corresponds to the *right* y-axis and shows the number of lazy updates that take place each second. Because this is a very large data set, many of the keys are not accessed immediately, taking full advantage of laziness. Although the lazy updates continue at a rate of about 500 per second at the 1,100 second mark, this does not significantly

TABLE III: Offline update pause times

	Pause (s)	Update Events
Amico	87s	792,711 : rename
redisfs	12s	10,000 : compress, 251 : rename (123,002 total keys in database)

impact overall queries per second, as shown by the solid line maintaining a similar QPS before and after the update.

VI. Related Work

In the realm of relational databases, the evolution of an application's schema is characterized by the changes to the `CREATE TABLE` statements used to instantiate the schema in subsequent versions of the application. In practice, complex schema changes often require taking the application offline or locking the database tables, such as the update to Wikipedia that held a write lock for 22 hours [16]. Prior research has proposed supporting non-blocking schema changes by accepting out of date copies of database objects [38] or by implementing changes on-the-fly using triggers [39] or log redo [40]. Additionally, several professional tools can perform `ALTER TABLE` operations in a non-blocking manner [41], [42], [43], [44], [45]. Because these tools focus only on the database, the changes implemented must be backward compatible to avoid breaking the application logic. To avoid this limitation, the Imago system [46] proposed installing the new version in a parallel universe, with dedicated application servers and databases, which allowed it to perform an end-to-end upgrade atomically. This can be achieved in practice by deploying parallel AppEngine [24] applications, at multiple versions. However, this approach duplicates resources and exposes the new version to the live workload only after the data migration was completed.

In contrast, the F1 database from Google implemented an asynchronous protocol [11] for adding and removing tables, columns and indexes, which allows the servers in a distributed database system to access and update all the data during a schema change and to transition to the new schema at different times. This is achieved by having stateless database servers with temporal schema leases, by identifying which schema-change operations may cause inconsistencies, and by breaking these into a sequence of schema changes that preserve database consistency as long as servers are no more than one schema version behind. Google's Spanner distributed key-value store [47] (which provides F1's backend) supports changes to key formats and values by registering schema-change transactions at a specific time in the future and by utilizing globally synchronized clocks to coordinate reads and writes with these transactions. These systems do not address changes to the format of Protobufs stored in the F1 columns or Spanner values [12] or inconsistencies that may be caused by interactions with (stateful) clients using different schemas [48].

Schema evolution in NoSQL databases is less well understood, as these databases do not provide a data definition language for specifying the schema. However, many applications attach meaning to the format of the keys and values stored in the database, and these formats may evolve over time. In particular, the values often correspond to data structures serialized using JSON [10] or a binary format like Thrift [8], Protobufs [7], or Avro [9]. The latter formats have schema-aware parsers, which include some support for schema changes, e.g. by skipping unknown fields or by attempting to translate data from the writer schema into the reader schema [13]. However, orchestrating the actual changes to the data and the application logic is entirely up to the programmer.

One approach to defining schema changes defines a declarative schema evolution language for NoSQL databases [49]. This language allows specifying more comprehensive schema changes and enables the automatic generation of database queries for migrating eagerly to the new schema. (While the paper also mentions the possibility of performing the migration in a lazy manner, which is needed for avoiding downtime, design and implementation details are not provided.) Other approaches use a domain-specific language (DSL) for describing data schema migrations for Python [31] and for Haskell datatypes [50]. Many other approaches [51], [52], [53], [54] have focused on the problem of synthesizing the transformation code to migrate from one schema version to the next, and the transformation is then typically applied offline, rather than incrementally online. In this paper, we focus on how to apply a transformation without halting service rather than synthesizing the transformation code.

In practice, developers are often advised to handle all the necessary schema changes in custom code, added to the application logic that may modify the data in the database [12], [17], [18], [19]. This approach burdens programmers with complex code that mixes application and schema-maintenance logic and does not provide a mechanism for reasoning about the correctness of schema changes performed concurrently with the live workload.

Our work is also related to the body of research on dynamic software updates [20], [23], [21], [22], which aim to modify a running program on-the-fly, without causing downtime. However, with the exception of a position paper [55], these approaches focus on changes to code and data structures loaded in memory, rather than changes to the formats of persistent data stored in a database.

VII. Conclusions and Future Work

This paper has presented KVolve, a general approach to evolving a NoSQL database without downtime. KVolve adapts Redis to migrate data as it is accessed, reducing downtime that would otherwise result during a data upgrade, and minimizing required changes to applications. We find that KVolve imposes essentially no overhead when not performing an update, and minimal overhead when performing an update.

In the future, we would like to expand KVolve to work with Redis Cluster, a distributed implementation of Redis. We also would like to add direct support for programmer-specified, backward-compatible updates, which would support continued operation without restarting clients. Finally, we would like to streamline writing the transformation function with a DSL, simplifying the update planning process.

We plan to release our code and make it freely available.

REFERENCES

[1] "Redis," http://redis.io/.
[2] Apache, "The apache cassandra project," http://cassandra.apache.org.
[3] MongoDB, "Mongodb," http://www.mongodb.com/.
[4] G. R. Guide, "Market guide for in-memory dbms," http://www1.memsql.com/gartner.html, 2014.
[5] M. Asay, "Nosql databases eat into the relational database market," http://www.techrepublic.com/article/nosql-databases-eat-into-the-relational-database-market/, 2015.
[6] J. Medina, *Nosql 133 Success Secrets - 133 Most Asked Questions on Nosql - What You Need to Know*. Emereo Pty Limited, 2014.
[7] Google, "Protocol buffers," https://developers.google.com/protocol-buffers/.
[8] Apache, "Apache thrift," http://thrift.apache.org/.
[9] ——, "Apache avro," http://avro.apache.org/.
[10] E. International, "The json data interchange format," http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf.
[11] I. Rae, E. Rollins, J. Shute, S. Sodhi, and R. Vingralek, "Online, asynchronous schema change in F1," in *VLDB*, 2013.
[12] P. Sadalage and M. Fowler, *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Pearson Education, 2012.
[13] M. Kleppmann, "Schema evolution in avro, protocol buffers and thrift," http://martin.kleppmann.com/2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html, 2013.
[14] Project Voldemort, "Voldemort support for avro schema evolution," http://www.project-voldemort.com/voldemort/avro-schema-evolution.html.
[15] S. Sanfilippo, "Finally redis collections are iterable," http://antirez.com/news/63, 2013.
[16] WikiMedia, "Mediawiki 1.5 upgrade," http://meta.wikimedia.org/wiki/MediaWiki_1.5_upgrade, 2005.
[17] D. Bellot, "Painless data migrations with schema-less nosql data stores and redis," https://github.com/ServiceStack/ServiceStack.Redis/wiki/MigrationsUsingSchemalessNoSql, 2011.
[18] D. Rethans, "Managing schema changes with mongodb," http://derickrethans.nl/managing-schema-changes.html, 2013.
[19] stackoverflow.com, "Are there any tools for schema migration for nosql databases?" http://stackoverflow.com/questions/1961013/are-there-any-tools-for-schema-migration-for-nosql-databases.
[20] C. M. Hayden, K. Saur, E. K. Smith, M. W. Hicks, and J. S. Foster, "Kitsune: Efficient, general-purpose dynamic software updating for C," *ACM Trans. Program. Lang. Syst.*, vol. 36, no. 4, p. 13, 2014.
[21] C. Giuffrida, C. Iorgulescu, and A. S. Tanenbaum, "Mutable checkpoint-restart: automating live update for generic server programs," in *Proceedings of the 15th International Middleware Conference, Bordeaux, France, December 8-12, 2014*, L. Réveillère, Ed. ACM, 2014, pp. 133–144.
[22] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew, "Polus: A powerful live updating system," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 271–281.
[23] L. Pina, L. Veiga, and M. W. Hicks, "Rubah: DSU for java on a stock JVM," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, A. P. Black and T. D. Millstein, Eds. ACM, 2014, pp. 103–119.
[24] Google, "Google app engine: Platform as a service," https://cloud.google.com/appengine/docs.
[25] Oracle, "Using sessions and session persistence," https://docs.oracle.com/cd/E11035_01/wls100/webapp/sessions.html.
[26] Project Voldemort, http://www.project-voldemort.com/.
[27] Berkeley DB, http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html.
[28] DB-Engines, http://db-engines.com/en/ranking/key-value+store.
[29] Y. Yu, "Scaling redis at twitter," https://www.youtube.com/watch?v=rP9EKvWt0zo, Aug 31, 2014, San Francisco Redis Meetup.
[30] P. Lehtinen, "Jansson," http://www.digip.org/jansson/.
[31] K. Saur, J. Collard, N. Foster, A. Guha, L. Vanbever, and M. Hicks, "Safe and flexible controller upgrades for SDNs," in *Proceedings of the Symposium on SDN Research (SOSR)*, Mar. 2016.
[32] K. Saur, T. Dumitras, and M. W. Hicks, "Evolving nosql databases without downtime (version 1)," *CoRR*, vol. abs/1506.08800v1, 2015. [Online]. Available: http://arxiv.org/abs/1506.08800v1
[33] S. Kemp, "Redisfs," http://www.steve.org.uk/Software/redisfs/.
[34] M. Szeredi, "Fuse," http://fuse.sourceforge.net/.
[35] J. Katcher, "Postmark," Network Appliance, Tech. Rep. TR 3022, 1997.
[36] Agora Games, "Relationships (e.g. friendships) backed by redis," https://github.com/agoragames/amico.
[37] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," https://snap.stanford.edu/data/soc-LiveJournal1.html, Jun. 2014.
[38] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu, "Mariposa: A wide-area distributed database system," *VLDB J.*, vol. 5, no. 1, pp. 48–63, 1996.
[39] M. Ronström, "On-line schema update for a telecom database," in *ICDE*, 2000, pp. 329–338. [Online]. Available: http://dx.doi.org/10.1109/ICDE.2000.839432
[40] J. Løland and S. Hvasshovd, "Online, non-blocking relational schema changes," in *Advances in Database Technology - EDBT 2006, 10th International Conference on Extending Database Technology, Munich, Germany, March 26-31, 2006, Proceedings*, ser. Lecture Notes in Computer Science, Y. E. Ioannidis, M. H. Scholl, J. W. Schmidt, F. Matthes, M. Hatzopoulos, K. Böhm, A. Kemper, T. Grust, and C. Böhm, Eds., vol. 3896. Springer, 2006, pp. 405–422. [Online]. Available: http://dx.doi.org/10.1007/11687238_26
[41] S. Noach, "openark kit," https://code.google.com/p/openarkkit/.
[42] M. Callaghan, "Online schema change for mysql," https://www.facebook.com/notes/mysql-at-facebook/online-schema-change-for-mysql/430801045932, 2010.
[43] Percona, "pt-online-schema-change," http://www.percona.com/doc/percona-toolkit/2.1/pt-online-schema-change.html, 2013.
[44] SoundCloud, "Large hadron migrator," https://github.com/soundcloud/lhm, 2011.
[45] Serious Business, M. Freels, R. Ravi, and R. Olson, "Tablemigrator," https://github.com/freels/table_migrator, 2009.
[46] T. Dumitraş and P. Narasimhan, "Why do upgrades fail and what can we do about it?: Toward dependable, online upgrades in enterprise system," in *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, ser. Middleware '09. New York, NY, USA: Springer-Verlag New York, Inc., 2009, pp. 18:1–18:20.
[47] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, p. 8, 2013.
[48] T. Dumitras, P. Narasimhan, and E. Tilevich, "To upgrade or not to upgrade: impact of online upgrades across multiple administrative domains," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, Reno/Tahoe, NV, Oct 2010, pp. 865–876.
[49] S. Scherzinger, M. Klettke, and U. Störl, "Managing schema evolution in nosql data stores," in *Proceedings of the 14th International Symposium on Database Programming Languages (DBPL 2013)*, 2013.
[50] A. Gundry, "Coping with change: data schema migration in haskell," http://cufp.org/2015/coping-with-change-data-schema-migration-in-haskell.html, 2015.
[51] E. Rahm, "Towards large-scale schema and ontology matching," in *Schema Matching and Mapping*, Z. Bellahsene, A. Bonifati, and E. Rahm, Eds. Heidelberg: Springer, 2011, ch. 1, pp. 3–27.
[52] C. Curino, H. J. Moon, A. Deutsch, and C. Zaniolo, "Automating the database schema evolution process," *The VLDB Journal*, vol. 22, no. 1, pp. 73–98, 2013.
[53] Y. Velegrakis, R. J. Miller, and L. Popa, "Mapping adaptation under evolving schemas," in *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, ser. VLDB '03. VLDB Endowment, 2003, pp. 584–595.
[54] Y. An and T. Topaloglou, "Semantic web, ontologies and databases," in *VLDB Workshop, SWDB-ODBIS 2007, Vienna, Austria, September 24, 2007, Revised Selected Papers*, V. Christophides, M. Collard, and C. Gutierrez, Eds. Berlin, Heidelberg: Springer-Verlag, 2008, ch. Maintaining Semantic Mappings Between Database Schemas and Ontologies, pp. 138–152.
[55] A. Deshpande and M. Hicks, "Toward on-line schema evolution for non-stop systems," Presented at the 11th High Performance Transaction Systems Workshop, September 2005.