Around Multi-Model Database

Amirouche Boubekki

https://hyper.dev

"Plans are only good intentions unless they immediately degenerate into hard work"

Peter Drucker

Table of contents

1 Introduction

While there is no shortage of database systems, there is none that meets all my requirements:

1. free software or open source or source available
2. can represent and query relational data
3. can represent and query recursive data
4. can represent and query geospatial data
5. can represent and query time data
6. can represent and query text data
7. support efficient pagination
8. support efficient versioning
9. support horizontal scaling
10. support ACID transactions across “entities”
11. embeddable

We could argue indefinitly that one or more of those requirements are unnecessary, overkill and YAGNI[1]. We could argue that by relaxing a few of the requirements, a particular software or set of softwares can come close. We could argue endlessly that building yet-another-database is NIH[2] syndrom, wheel re-invention that curse the software industry with fragmentation and fatigue. We could invoke UNIX philosophy, entreprise software architectures, experiences, know-how, failed patterns, decades of good services and big communities.

We could agree that a lone inexperienced idealist web developer can not disrupte software industry with an unwiedling project, click-bait titles and wanna-be Sillicon-Valley friendly landing pages even if they have the best intentions in the world.

We could and hopefully we will.

Standing on the shoulders of giants, this document is merely trying to bring the attention of the community to a growing trend in the industry by demonstrating what one can achieve with an *ordered key-value store*. Most of what is described in this document already exists in the wild hidden in closed-source softwares but also hidden in freely available projects written C++, Java, C#, Go, Rust, Python, Clojure and Scheme.

Anyway, the ideas and algorithms described in this document are not only useful. It can help on the path to build your own database abstractions on top *ordered key-value stores* and get the most of what modern data storage systems have to offer.

2 Ordered Key-Value Store

> *An ordered key-value store is to storage systems what Scheme is to programming languages: a programmable programming system.*

In this section, we will see the main principles that allow to build upon higher level abstraction and answer the following questions:

- What is an ordered key-value store?
- How to store data?
- How to query data?
- How to create a schema?

2.1 Getting started

An ordered key-value store (okvs) is an abstraction used in various storage systems that allow on-disk persistence. There is many implementations with various features and limitations. Some of them support somekind transactional guarantees. Some of them are distributed ie. horizontally scalable. I will focus on what is common and what is applicable with most (any?) ordered key-value stores.

Definition 1. *An ordered key-value store is a mapping of bytes-to-bytes where key-value pairs are ordered lexicographically.*

Lexicographical order is natural language dictionary order. That is the ordering where “care” comes before “careful” and after “car”. In the case of Python strings the lexicographical order is the same as the default comparison:

1. You Ain’t Gonna Need It
2. Not Invented Here

```
>>> 'car' < 'care' < 'careful'
   True
>>> 'because' < 'careful'
   True
```

Using numbers as strings leads to the strange situation where it appears like "9" is bigger than "10":

```
>>> '9' > '10'
   False
>>> '10' > '9'
   False
```

The immediate result is that you can not compare the representation of integers as strings to obtain the natural ordering of integers. To be able to preserve natural order of integers when they are represented as bytes, they must be packed otherwise:

```
>>> import struct
>>> pack = lambda integer: struct.pack('>Q', integer)
>>> pack(10) > pack(9)
   True
>>> isinstance(pack(42), bytes)
   True
```

Lemma 2. *There is an algorithm that allows to represent objects and composition of objects of the following types as bytes that preserve their respective natural ordering using a fixed unspecified ordering between objects that are from different types:*

i. boolean

ii. big integer

iii. float

iv. double

v. symbol

vi. string

vii. list

viii. vector

ix. bytevector

Corollary 3. *An ordered key-value store can be seen as a mapping of key-value pairs ordered by key where keys and values are objects and composition of objects of the following types ordered using their natural order and a unspecified fixed order between objects of different types:*

i. boolean

ii. big integer

iii. *float*

iv. *double*

v. *symbol*

vi. *string*

vii. *list*

viii. *vector*

ix. *bytevector*

The okvs can be seen as a $2 \times n$ table where n is the total number of key-value pairs in the database, such as:

key	value
``magic number''	42
``seyfu''	93600
(``hello'' ``world'')	(``from'' ``the'' ``Internet'')
(``subspace-01'' ``key'')	``value''

Table 1. Example okvs with composition of base types

Remark 4. In real world scenario, most of the time the keys are always tuples. And as far as the key is concerned, tuples and non-tuples are not mixed.

Remark 5. The actual okvs manipulates bytes and keeps the keys in dictionary order. It is possible to represent various base types of the host language as bytes that preserve their natural ordering.

Remark 6. We can (almost) freely compose (almost) any base type to possibly achieve a fractal design.

2.2 Forward

2.2.1 Mapping primitives

The okvs, as a mapping, has primitives to get and set a particular key-value pair. The following will retrieve the value associated with a key relying on `(arew data pack)` procedures `(pack . objects)` → `bytevector` and `(unpack bytes)` → `list`:

```
(import (cffi wiredtiger okvs))
(import (arew data pack))


(define database (okvs "/tmp/wt" '((create? . #t))))

(okvs-in-transaction database
                     (lambda (txn)
                       (okvs-ref txn (pack 42)))) ;; => #f

(okvs-in-transaction database
```

```
                      (lambda (txn)
                        (okvs-set! txn (pack 42) (pack "magic"))))

(okvs-in-transaction database
                      (lambda (txn)
                        (okvs-ref txn (pack 42)))) ;; => '("magic")
```

2.2.2 Range procedure

The particularity of the okvs is its primitive procedure that allows to efficently get all key-value pairs that are inside a given range (or slice) of keys, namely `okvs-range`:

```
(import (cffi wiredtiger okvs))
(import (arew data pack))
(import (scheme generator))


(define database (okvs "/tmp/wt" '((create . #t))))

(okvs-in-transaction database
                      (lambda (txn)
                        (okvs-set! txn
                                   (pack 1 'title)
                                   "DIY a multi-model database")
                        (okvs-set! txn
                                   (pack 1 'body)
                                   "That is getting started!")))

(define out
  (okvs-in-transaction database
                       (lambda (txn)
                         (generator->list
                          (lambda (key value) (cons (unpack key)
                                                    (unpack value)))
                          (okvs-range txn (pack 1) #f (pack 2) #f)))))

(equal? out '(((1 body) . "That is getting started!")
              ((1 title) . "DIY a multi-model database"))) ;; => #t
```

In the above example, `'(1 body)` comes before `'(1 title)` because `'body` comes before `'title`.

Another way to look at it is using the table reprensetation:

key	value
(1 body)	‘‘That is getting started’’
(1 title)	‘‘DIY a multi-model database’’

Table 2. Example of range query

Note 7. An astute reader will recognize that in that particular case using a prefix-search allows to retrieve the same set of key-value pairs. We will see later how it can be done.

This works because *keys are always in increasing order*. A more obvious use of the range procedure is to retrieve everything between two points in time. Given the following database:

key	value
#vu8(1)	‘‘baby’’
#vu8(2)	‘‘child’’
#vu8(3)	‘‘adolescent’’
#vu8(4)	‘‘young adult’’
#vu8(5)	‘‘adult’’
#vu8(6)	‘‘old’’
#vu8(7)	‘‘very old’’
#vu8(8)	‘‘dead’’

Table 3. Second example of range query

We can define the experienced persons as everything that is between “adult” and “dead” with “dead” persons excluded:

```
(define out
  (okvs-in-transaction database
                       (lambda (txn)
                         (generator-map->list
                          (lambda (key value) (cons (unpack key)
                                                    (unpack value)))
                          (okvs-range txn #vu8(5) #t #vu8(9) #f)))))
```

Warning 8. The procedure `pack` was not used because keys are just bytes and are not inside a list.

Let’s do another example, let’s store some news items by date and try to query everything from 2011/11 (included) to 2012/02 (included). News items are represented as UUID bytevectors and dates are lists that follow the following format `'(year month)`:

key	value
(2000 1)	#vu8(219 88 74 238 124 239 75 18 191 215 105 190 101 222 125 76)
(2001 3)	#vu8(231 255 144 226 106 191 65 242 181 190 106 253 16 64 92 182)
(2010 9)	#vu8(51 158 56 139 142 120 66 227 191 95 48 137 189 81 39 255)
(2011 7)	#vu8(167 242 232 92 115 73 71 241 186 92 247 41 169 101 242 44)
(2011 10)	#vu8(187 64 202 92 211 190 73 42 137 51 199 16 22 52 73 249)
(2011 11)	#vu8(187 64 202 92 211 190 73 42 137 51 199 16 22 52 73 249)
(2011 12)	#vu8(29 82 65 228 182 136 65 5 187 160 123 192 130 132 8 74)
(2012 2)	#vu8(180 241 185 247 24 206 66 212 181 228 49 224 217 24 205 212)

Table 4. Third example of range query

2.2.3 Prefix range procedure

We previsouly mentioned that given prefix range queries “naturally” appear in the context of okvs databases. Consider the previous table again:

key	value
(2000 1)	#vu8(219 88 74 238 124 239 75 18 191 215 105 190 101 222 125 76)
(2001 3)	#vu8(231 255 144 226 106 191 65 242 181 190 106 253 16 64 92 182)
(2010 9)	#vu8(51 158 56 139 142 120 66 227 191 95 48 137 189 81 39 255)
(2011 7)	#vu8(167 242 232 92 115 73 71 241 186 92 247 41 169 101 242 44)
(2011 10)	#vu8(187 64 202 92 211 190 73 42 137 51 199 16 22 52 73 249)
(2011 11)	#vu8(187 64 202 92 211 190 73 42 137 51 199 16 22 52 73 249)
(2011 12)	#vu8(29 82 65 228 182 136 65 5 187 160 123 192 130 132 8 74)
(2012 2)	#vu8(180 241 185 247 24 206 66 212 181 228 49 224 217 24 205 212)

Table 5. Example of prefix range query

The corresponding code will look like the following:

```
(define out
  (okvs-in-transaction database
                       (lambda (txn)
                         (generator-map->list
                          (lambda (key value) (cons (unpack key)
                                                    (unpack value)))
                          (okvs-prefix txn (pack 2011))))))
```

It rely on `(okvs-prefix transaction prefix)` to query every key-value pairs that have a given `PREFIX`. As you can see, the packing procedure behaves as we would expect.

Remark 9. The procedure `okvs-prefix` is tiny wrapper around `okvs-range`.

2.3 Beyond

Key composition is the pratice of storing list of objects as key that take advantage of the ordered nature of the key space with the procedure `okvs-range`. It allows to ground schemas and build higher level abstractions.

2.3.1 Unique keys

The unique keys pattern rely on storing what is a "value" from the application perspective in the key and possibly leave the value column empty.

Let's re-imagine the previous news items example and make it more realistic. In a real world scenario, multiple news can be published the same year. To be able to store and query by date multiple news items from the same year, *one must make the key unique*.

Here is an *application* view of the news:

identifier	year
#vu8(219 88 74 238 124 239 75 18 191 215 105 190 101 222 125 76)	2015
#vu8(231 255 144 226 106 191 65 242 181 190 106 253 16 64 92 182)	2015
#vu8(51 158 56 139 142 120 66 227 191 95 48 137 189 81 39 255)	2016
#vu8(167 242 232 92 115 73 71 241 186 92 247 41 169 101 242 44)	2017
#vu8(187 64 202 92 211 190 73 42 137 51 199 16 22 52 73 249)	2018
#vu8(187 64 202 92 211 190 73 42 137 51 199 16 22 52 73 249)	2018
#vu8(29 82 65 228 182 136 65 5 187 160 123 192 130 132 8 74)	2019
#vu8(180 241 185 247 24 206 66 212 181 228 49 224 217 24 205 212)	2019

Table 6. View of the application of some news items related to their year of publication

If we want to query the news items by year, the year must be part the key since okvs can only query using the key. Since the okvs is a mapping they can not be multiple values associated with the same key except if you use a list to group every identifiers in the same key-value pair.

There is occasions, where grouping multiple values for a given key is a good thing, example in the case categorical data and more generaly in situations where the list will not grow too much. Another inconvenient of grouping values, is that you can not easily do pagination. We will explain how pagination happens using key-composition, it will be obvious that grouping values is not helpful in this case.

The recommended way to represent the news items in a way that makes it easy to query and paginate the identifiers is to append the identifier to the key and leave the value empty:

key
`(2015 #vu8(219 88 74 238 124 239 75 18 191 215 105 190 101 222 125 76))`
`(2015 #vu8(231 255 144 226 106 191 65 242 181 190 106 253 16 64 92 182))`
`(2016 #vu8(51 158 56 139 142 120 66 227 191 95 48 137 189 81 39 255))`
`(2017 #vu8(167 242 232 92 115 73 71 241 186 92 247 41 169 101 242 44))`
`(2018 #vu8(187 64 202 92 211 190 73 42 137 51 199 16 22 52 73 249))`
`(2018 #vu8(187 64 202 92 211 190 73 42 137 51 199 16 22 52 73 249))`
`(2019 #vu8(29 82 65 228 182 136 65 5 187 160 123 192 130 132 8 74))`
`(2019 #vu8(180 241 185 247 24 206 66 212 181 228 49 224 217 24 205 212))`

Table 7. year-identifier key space

That way every key is unique, and one can use `okvs-range` to query by date. To do pagination, one must keep around the last seen key, and do another `okvs-range` starting from it.

2.3.2 Subspace

Subspaces are another key-composition pattern that allows to split the key space into multiple subspaces, that when used with care, allow to reproduce a similar concept to collections and tables.

Again, let's consider news items and more generally somekind of news application where news items are associated with authors. That is there is at least two entities in that application. In other databases, it would be translated into two collections or tables.

In okvs, we can use subspaces. A subspace is a collocated set of keys that share a common prefix.

Warning 10. A subspace must not share a prefix with another subspace.

The following table demonstrate the use of prefixes to separate different entities:

key	value
`(author 1 firstname)`	`"Amirouche"`
`(author 1 lastname)`	`"Boubekki"`
`(news 1 author)`	`1`
`(news 1 tags)`	`(okvs apple foundationdb)`
`(news 1 title)`	`"FoundationDB was opensourced by Apple"`
`(news 2 author)`	`1`
`(news 2 tags)`	`(okvs mongodb wiredtiger)`
`(news 2 title)`	`"WiredTiger was bought by MongoDB"`

Table 8. Subspaces

2.4 Summary

1. The okvs is a mapping of bytes-to-bytes where key-value pairs are ordered lexicographically by key.
2. The `pack` procedure allows to represent most base data types as bytes while preserving their natural order.
3. Therefore in most cases, we can consider that the okvs stores composition of base data types.
4. The okvs support classic mapping procedures `okvs-set!` and `okvs-ref`.
5. The most important primitive is `okvs-range` which behavior stems from the property that keys are stored in increasing order.
6. Key composition allows to build schemas.

2.5 Exercices

Exercise 1. Consider the following database:

key	value
`(author 1 firstname)`	`"Amirouche"`
`(author 1 lastname)`	`"Boubekki"`
`(news 1 author)`	`1`
`(news 1 tags)`	`(okvs apple foundationdb)`
`(news 1 title)`	`"FoundationDB was opensourced by Apple"`
`(news 2 author)`	`1`
`(news 2 tags)`	`(okvs mongodb wiredtiger)`
`(news 2 title)`	`"WiredTiger was bought by MongoDB"`

Table 9. News items with authors

It is not possible to query by "tag" without doing a full scan. Rework the database schema to support querying news by tag.

3 Row Store

The row store is inspired from Relational DataBase Management Systems (RDBMS) in sense that it rely on fixed-length lists to represent rows in primary source of truth. So called indices, a secondary source of truth, are updated in the same transaction as the row and allow to efficently query and do pagination. More advanced indexing scheme are considered in other sections of this document.

In this, section we will:

- Represent tabular data
- Introduce the idea of "metadata" subspace to store information that allow among other things to do introspection
- Explain how to create secondary representation and how they are useful aka. indices.

3.1 What is the data?

Continuing with the idea of a news application we will represent news items, authors and tags. Here is the high-level RDBMS representation of the data we are interested in:

pk	firstname	lastname
1	Amirouche	Boubekki

Table 10. Author table

pk	title	year	author
1	“MongoDB acquired WiredTiger”	2014	1
2	“Apple open-sourced FoundationDB”	2018	1

Table 11. News items table with a foreign-key on author column

pk	name	description
1	“MongoDB”	“Company behind WiredTiger and MongoDB”
2	“FoundationDB”	“ACID, distributed, fault-tolerant okvs”
3	“WiredTiger”	“ACID, embeddable, powerful okvs”
4	“Apple”	“Steve Jobs legacy”

Table 12. Tag table

news	tag
1	1
1	3
2	2
2	4

Table 13. Many-to-many relations between news items and tags

3.2 How we want to query the data?

- We want to be able to retrieve news items with its author information and the related tags
- We want to be able to retrieve news items by year

3.3 Implementation

3.3.1 Primary representation

key	value
`(author row 1)`	`("Amirouche" "Boubekki")`
`(news row 1)`	`("MongoDB acquired WiredTiger" 2014 1)`
`(news row 2)`	`("Apple open-sources FoundationDB" 2018 1)`
`(news-tag row 1)`	`(1 1)`
`(news-tag row 2)`	`(1 3)`
`(news-tag row 3)`	`(2 2)`
`(news-tag row 4)`	`(2 4)`
`(tag row 1)`	`("MongoDB" "Company behind WiredTiger and MongoDB")`
`(tag row 2)`	`("FoundationDB" "ACID, distributed, fault-tolerant okvs")`
`(tag row 3)`	`("WiredTiger" "ACID, embeddable, powerful okvs")`
`(tag row 4)`	`("Apple" "Steve Jobs legacy")`

Table 14. Primary representation of the news data in a row store

Every entities is in a subspace that takes the name of the entity table.

Given the current schema, we can query rows by dubbed primary keys (pk) and make pagination but only seemingly random ordereding using it.

3.3.2 Secondary representation

4 Document Store

5 Tuple Stores

5.1 Triple Store

5.2 Generic Tuple Store

5.3 Versioned Generic Tuple Store

6 Full-Text Search

7 Space and Time

8 Conclusion

As of June of 2019, it is not clear whether seyfu will be a standalone database server or incorporated in one application or remain a fantasy. The most advanced implementation is coded in Scheme programming language and can be found on the Internet in arew repository[3].

It seems to me that *ordered key-value stores* are useful, useable and used.

A question remains: Will it remain a tool for the experts? Maybe it will become more widely accepted in the toolbelt of backend software developers? One way to find out the response to that question is to answer another question: Are most of any domain needs covered by the abstractions provided in this document or there is still some specific needs that arise in some domain that require chirurgical intervention. In the latter case, *ordered key-value stores* are what will be the more useful to know.

3. https://git.sr.ht/~amz3/chez-scheme-arew