YoctoDB: A Partitioned Immutable Embedded Database[*]

Vadim Tsesko
Yandex.Classifieds
mail@incubos.org

Svyatoslav Demidov
Yandex.Classifieds
demidov.svyatoslav@yandex.com

ABSTRACT

YoctoDB is a small embedded engine for extremely fast partitioned immutable-after-construction databases. Several high load services at Yandex.Classifieds implement pipelined partitioned data reindexing. The result of the reindexing process is an immutable index delivered to many search machines, reopened as a part of the composite index and queried when serving user requests. Read performance, memory consumption, fast reopening and reproducible latencies are paramount for the database engine. YoctoDB has successfully provided a solution for all of these services. We describe the role of YoctoDB in the architecture of indexing and search components, it's simple data model, client API, design, implementation and use cases. We conclude the paper with limitations of the approach and directions of future development.

Keywords

Database, embedded, read-only, immutable, parametric search, horizontal partitioning, sharding, high load, low latency, real-time

CCS Concepts

•Information systems → DBMS engine architectures; Information storage technologies; **•Software and its engineering →** *Real-time systems software;*

1. INTRODUCTION

Over the last three years we have designed, prototyped, implemented, tested and successfully deployed an embedded engine for partitioned immutable-after-construction databases called YoctoDB. Initial development started as an ambitious attempt to replace the search engine based on Apache Lucene[1]. Main goals were improving search latencies and shrinking the search cluster of many tens of nodes. YoctoDB is designed to achieve extremely fast throughput and low latency querying at the price of database construction resources invested in building efficient persistent data structures which support low-memory runtime overhead and cheap database reopening to comprise constantly arriving index updates.

YoctoDB is used for a variety of demanding workloads including latency-sensitive data delivery to end users. One notable example is the backend of Yandex.Auto[2] and Auto.ru[3] with multi million monthly audience[4] under load of thousands search requests per second including complex multi-field queries over more then million document index updated every minute.

Once constructed a YoctoDB database logically resembles a simple traditional SQL database containing only one table and supporting `SELECT` queries with a variety of filtering and ordering conditions through SQL-like DSL. YoctoDB doesn't support nested and `GROUP BY` queries yet due to unpredictable temporary memory consumption which is an important factor in real-time use cases. YoctoDB doesn't enforce a static data schema and treats all filterable and sortable fields as comparable arrays of bytes which can be considered flexible or error prone depending on the use case. As a matter of fact, YoctoDB can be regarded as another implementation of the principle stating that the more specialized technology is the more performant it might be.

Section 2 describes the role YoctoDB plays in the architecture of search backends, the set of goals and constraints and motivation for the initial development. Section 3 specifies the data model in detail and Section 4 provides an overview of the API. Section 5 describes the fundamentals of the YoctoDB implementation and Section 6 explains low-level details and particularities of Java implementation. Section 7 provides performance measurements using real queries and lists YoctoDB use cases. Section 8 describes directions for future work. Finally, Section 9 overviews related work and Section 10 presents our conclusions including limitations of the current implementation.

2. MOTIVATION

Yandex.Classifieds indexing and search backends implement horizontal partitioning scheme depicted in Figure 1 and thoroughly described by the authors [13].

[*]The source code of Java implementation is available at https://github.com/yandex/yoctodb
[1]https://lucene.apache.org

[2]https://auto.yandex.ru
[3]https://auto.ru
[4]https://stat.yandex.ru/Russia/Auto

CEE-SECR '16, October 28 - 29, 2016, Moscow, Russian Federation
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4884-3/16/10. . . $15.00
DOI: http://dx.doi.org/10.1145/3022211.3022214

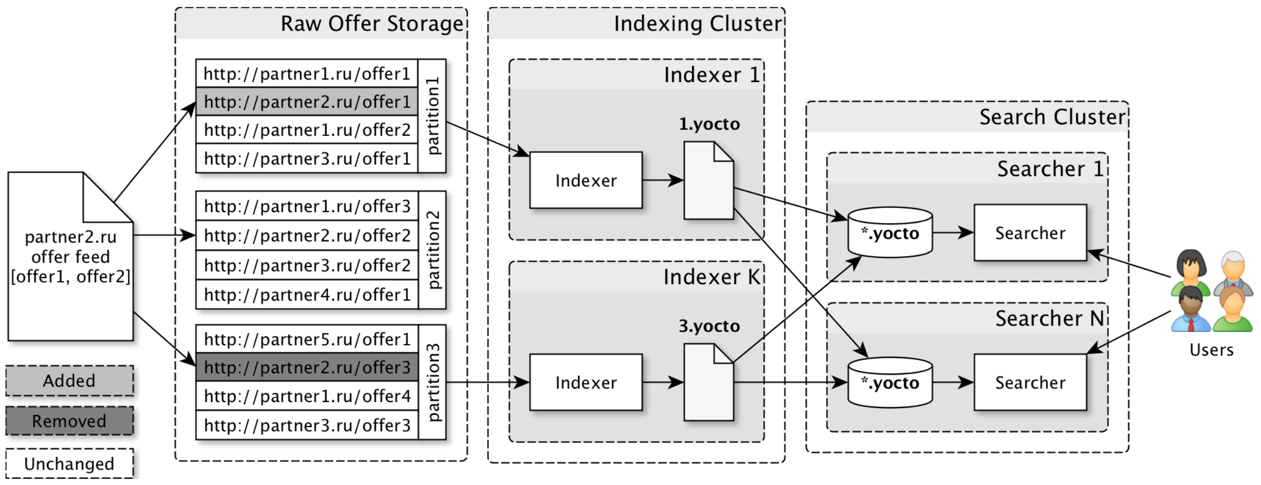
Raw Offer Storage
http://partner1.ru/offer1
http://partner2.ru/offer1
http://partner1.ru/offer2
http://partner3.ru/offer1
partition1
http://partner1.ru/offer3
http://partner2.ru/offer2
http://partner3.ru/offer2
http://partner4.ru/offer1
partition2
http://partner5.ru/offer1
http://partner2.ru/offer3
http://partner1.ru/offer4
http://partner3.ru/offer3
partition3
partner2.ru offer feed [offer1, offer2]
Added
Removed
Unchanged
Indexing Cluster
Indexer 1
1.yocto
Indexer
Indexer K
3.yocto
Indexer
Search Cluster
Searcher 1
*.yocto
Searcher
Searcher N
*.yocto
Searcher
Users

Figure 1: Document processing pipeline
The set of raw offers is partitioned so that possible duplicates are placed into the same partition.
In this case 3 partitions contain offers of 5 partners.
A new version of `partner2.ru` offer feed contains updated `offer1` mapped to `partition1` and unmodified `offer2` mapped to `partition2`. `offer3` is not present in the feed anymore, so it is going to be removed from `partition3`.
The changes applied trigger reindexing of `partition1` and `partition3` by the nodes from the *Indexing Cluster*. As a result, corresponding immutable partition indexes `1.yocto` and `3.yocto` are produced and delivered to local file storage `*.yocto` of each node in the *Search Cluster*.
Nodes in the *Search Cluster* periodically reopen the composite database from local `*.yocto` file storage to comprise the latest indexes of all the partitions and to serve updated data to *Users*' requests.

Indexing Cluster capacity and performance affects the rate of offer reindexing and the resulting time between an offer update accepted by the system and the updated offer made accessible to user requests. The throughput of *Indexing Cluster* can be increased by horizontal scaling i. e. adding extra nodes to the cluster.

Search Cluster serves user requests and demands the highest level of availability, the strictest latency requirements and ability to reopen the composite database quickly and without extra costs to handle the rate of incoming partition updates. At the same time *Indexing Cluster* has softer availability requirements, because in case of being offline it just stops users from getting fresh updates, but the outdated database can still be used to serve requests.

Strictly speaking, *Users* in Figure 1 are not human beings, but other services communicating with *Search Cluster*. The query language provided by *Searcher* supports selecting the subset of documents conforming to multiple (in)equality conditions joined by logical predicates and it is thoroughly described in Section 4.

Requirements

As a result of the architecture analysis we have declared the following high-level nonfunctional requirements and constraints for the YoctoDB design:

- Query throughput and latency metrics are paramount
- Query throughput and latency must be stable and deterministic
- A composite database must be able to be reopened without extra cost in additional memory or preprocessing required
- We can afford to spend more computing resources in the indexing phase to improve throughput and/or latency in the search phase
- Usually database files reside in disk cache, because they have been received recently

We will try our best to link the design decisions described in the following sections to the requirements listed above.

3. DATA MODEL

YoctoDB *database* is a collection of documents. A *document* has an optional *payload* represented by an opaque array of bytes and a set of named *field* values stored as comparable arrays of bytes. There are several *field* types supported:

- **Filterable** fields for extracting corresponding documents using possibly multiple (in)equality or range conditions at querying stage
- **Sortable** fields for ascending/descending and possibly composite ordering of filtered documents
- **Full** fields for both filtering and sorting

Filterable fields may have 0, 1 or more values for each document that enables storing and indexing multi-valued fields, for instance tags. Sortable fields require exactly one value for each document.

YoctoDB provides conversions from primitive values to comparable byte arrays which are stored persistently and used during querying phase.

Composite YoctoDB database is a logical entity treating a set of basic YoctoDB databases as a whole. It applies a user query to each of the underlying databases and merges the results in the streaming way according to the specified ordering directives.

Example

As an example consider a car selling offer:

- The payload is a serialized rich offer object containing description, image URLs, etc.
- Filterable `region_id` field as a byte array representation of the offer geographic region identifier
- Filterable `mark` field as a byte array representation of the ASCII string value: `FORD`, `BMW`, etc.
- Filterable `model` field as a byte array representation of the ASCII string value: `FOCUS`, `X6`, etc.
- Sortable `date` field as a big-endian byte array representation of the offer creation date converted to `int` UNIX time with seconds precision
- Full `price` field as a big-endian byte array representation of the offer `long` price in cents

This document schema allows for the following example queries[5]:

- Count all the offers having the specified `region_id` value
- Show top 10 offers having the specified value of `region_id`, `mark` equal to `BMW`, `price` in range `[10000$, 20000$]` and order the offers by `date` descending

It is straightforward to suggest an isomorphic SQL model (see Listing 1) and SQL queries (see Listing 2).

Listing 1: Example SQL model

```
CREATE TABLE offers(
  id INT AUTO_INCREMENT,
  payload BLOB,
  region_id INT,
  mark TEXT,
  model TEXT,
  date TIMESTAMP,
  price BIGINT);

CREATE INDEX region_id_idx
  ON offers(region_id);
CREATE INDEX mark_idx ON offers(mark);
CREATE INDEX model_idx ON offers(model);
CREATE INDEX date_idx ON offers(date);
CREATE INDEX price_idx ON offers(price);
```

Listing 2: Example SQL queries

```
SELECT COUNT(*) FROM offers
  WHERE region_id = :region_id;

SELECT payload FROM offers
WHERE
  region_id = :region_id AND
  mark = "BMW" AND
  price BETWEEN (1000000, 2000000)
ORDER BY date DESC
LIMIT 10;
```

[5]All the example field values need to be converted to comparable arrays of bytes corresponding to the mapping used during the database construction phase.

If `region_id` field is always specified as a first condition clause in real queries (that quite often is the case) an example composite database could be partitioned by this field to narrow the search space.

4. API

YoctoDB is an embedded database engine supporting two distinct APIs:

- **Mutable** database building API including database serialization
- **Immutable** database querying API including opening the serialized database

Persistent YoctoDB database serialization format is language agnostic, but only Java API is implemented at the moment, so the examples below use Java.

The common sequence of API calls looks the following way:

1. Create a database builder
2. Create zero or more document builders, for each one:
   (a) Optionally set payload
   (b) Set zero or more field values
   (c) Add the document builder to the database
3. Build and serialize the immutable database
4. Open the immutable database
5. Optionally construct a composite database from immutable ones
6. Run zero or more queries against the database
7. Close the database

The calls before database serializing use mutable API, the calls after opening the serialized database use immutable API. Listing 3 shows an example of a YoctoDB session using both APIs[6].

4.1 Mutable

Mutable API works with in-memory database representation optimized for convenient traversal, analysis and post-processing during the subsequent serialization phase. The details are described in Section 5.

There are many overloaded implementations of `withField()` method converting different primitive values to comparable byte arrays. For instance, integer types are converted to fixed-length big-endian representation. In case of a missing conversion a user may construct an instance of `UnsignedByteArray` herself and pass it to `withField()` method.

[6]Java imports are omitted to save space.

Header	Segment	Segment	...	Checksum

Figure 2: YoctoDB file format

4.2 Immutable

YoctoDB query DSL supports the following features:

- Value filtering operators: `eq`, `gt`, `gte`, `lt`, `lte`, `in`, `between`
- Boolean combinators: `and`, `or`, `not`
- `orderBy` multiple fields in ascending or descending order
- `skip` and `limit`
- `count` results
- Apply a custom callback to the filtered documents IDs in order
- Extract document payload by document ID
- Extract sortable field value by document ID

When supplying field values during query construction a user must convert primitive values to comparable byte arrays using the same conversion facilities as used in mutable API.

Document payload is returned as an efficient `Buffer` view without allocating extra memory.

5. IMPLEMENTATION

We are going to start with persistent YoctoDB database format, describe its segments and persistent collections they rely upon. Finally, query execution is considered, partitioning, memory and thread safety aspects are explained.

5.1 Format

YoctoDB file format is depicted in Figure 2. Basically YoctoDB is serialized to a sequence of bytes encoding *Header*, zero or more *Segments* and *Checksum*. Header contains magic byte sequence, database format version and checksum length. Checksum is calculated in a streaming way during YoctoDB database serializing and is optionally checked when opening the database.

5.2 Segments

There are several types of Segments:

- **Filterable** segment for each named filterable field
- **Sortable** segment for each named sortable field
- **Full** segment for each named field supporting filtering and sorting
- **Payload** segment for document payloads

Each segment starts with a header containing segment size and type[7]. Segments are assembled from abstract persistent collections[8].

[7]Segment headers are omitted in figures.

[8]When depicting segment examples square brackets denote lists or sets and curly brackets denote mappings.

Field Name	Set of Values	Value to Docs

Figure 3: Filterable segment structure

"mark"	["BMW", "FORD"]	{0: [0, 2], 1: [1, 3, 4]}

Figure 4: Filterable segment example
The segment contains two sorted unique field values: `FORD` and `BMW`. Field `mark` value `FORD` appears in documents numbered `0` and `2`, value `BMW` appears in documents `1`, `3` and `4`.

Filterable

Filterable segment (see Figure 3) stores a document field name, a persistent collection representing the sorted set of unique values and a persistent multi-map from a value index to indexes of documents having the value which serves as a reverse index. This layout makes it possible to choose the subset of unique values conforming to the query condition using the set of values and extracting corresponding document indexes for the values' indexes using the multi-map. Figure 4 shows an example of filterable field `mark` segment.

Sortable

Sortable segment (see Figure 5) in addition to all the data *Filterable* segment contains is extended with a persistent collection directly mapping a document index to the value index to support extracting document field value by the end user. Value index to document index multi-map introduced in *Filterable* segments supports traversing document indexes in the value increasing or decreasing order which is used when executing sorting queries. Figure 6 shows an example of sortable field `date` segment.

Full

Full segment representation currently doesn't differ from *Sortable* segment, because *Sortable* segment is an extension of *Filterable* segment and its data structures support all the necessary filtering and sorting operations.

Payload

Payload segment contains only one persistent collection storing the indexed list of byte arrays representing documents' payloads. Figure 7 shows a payload segment example.

5.3 Persistent Collections

YoctoDB segments implementing filtering, sorting and payload extraction[9] rely on generic persistent collections:

- `ByteArraySortedSet` for representing sorted sets of values used for filtering and sorting
- `IndexToIndexMultiMap` for storing inverse mapping from a value index to document indexes

[9]We are going to use Java class names as collection names to be concise.

Field Name	Set of Values	Value to Docs	Doc to Value

Figure 5: Sortable segment structure

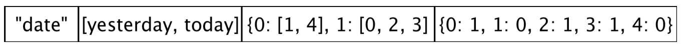

Figure 6: Sortable segment example
The segment contains two sorted unique field `date` values: `yesterday` and `today`. `yesterday` value appears in documents numbered `1` and `4`. `today` value appears in documents `0`, `2` and `3`. In addition, the segment contains a mapping from document index to value index.

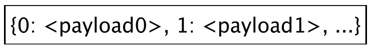

Figure 7: Payload segment example
The segment contains documents' payloads represented as arrays of bytes and indexed by document number.

- `IndexToIndexMap` for storing direct mapping from a document index to the value index
- `ByteArrayIndexedList` for representing document payloads accessible by document index

Each collection has mutable and immutable implementation. Mutable implementation is used during in-memory YoctoDB database construction and populating it with documents. When serializing YoctoDB an efficient persistent representation of the collection is built. Afterwards the immutable implementation is used for querying.

Persistent collections substitute the core of the YoctoDB engine and heavily contribute to its efficiency and performance. Each of them is described in details below.

`ByteArraySortedSet`

The supported operations are:

- Get the element (byte array) by index
- Find the index of the element
- Find the index of the smallest element greater than the provided element
- Find the index of the greatest element smaller than the provided element

There are two different implementations of the collection interface for two distinct use cases. **`FixedLengthByteArraySortedSet`** (see Figure 8) is used when all the byte arrays have the same length, for instance, primitive integer values. **`VariableLengthByteArraySortedSet`** (see Figure 9) is less compact, but it supports storing elements with different length, for instance, strings.

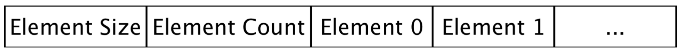

Figure 8: `FixedLengthByteArraySortedSet` format
When finding the element based on its index the element offset is calculated by multiplying element index and element size. Binary search is used to find the provided element.

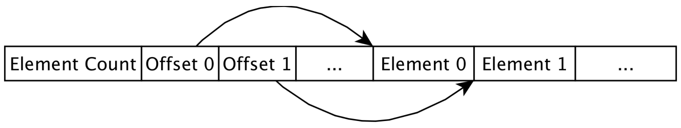

Figure 9: `VariableLengthByteArraySortedSet` format
When finding the element based on its index the element offset is extracted from the offset table.
Binary search is used to find the provided element. Each comparison includes two steps: extract the element offset and compare byte arrays starting from the offset.

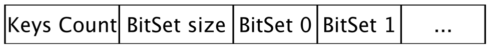

Figure 10: `BitSetIndexToIndexMultiMap` format

`IndexToIndexMultiMap`

The supported operations are:

- Fill the user provided `BitSet` with value indexes of keys in the user provided range
- Get ascending/descending iterator for the entry set

There are two different implementations: `BitSet`-based (see Figure 10) and `List`-based (see Figure 11). The most compact representation is chosen during YoctoDB serializing.

`IndexToIndexMap`

The only supported operation is getting the integer value by an integer key. Figure 12 shows the simple format.

`ByteArrayIndexedList`

The only supported operation is getting byte array by index. There are two implementations for fixed size and variable size byte arrays. Storage formats are equal to `ByteArraySortedSet` formats.

5.4 Query Execution

Query execution consists of three phases:

1. Build a `BitSet` containing documents' IDs that satisfy query filtering predicates using *Filterable* and/or *Full* segments
2. Optionally reorder documents' IDs according to `order by` clauses using *Sortable* and/or *Full* segments
3. Optionally[10] apply a user supplied callback to each document in order[11] until the callback signals to stop iteration or the iterator exhausts

[10]If it is not a `count` request case.
[11]Document callback may extract document payload and/or sorting field values from the database.

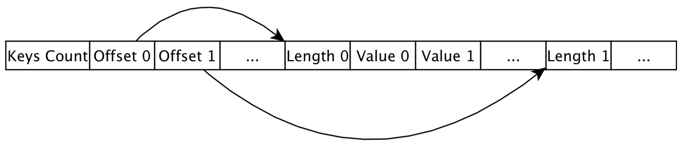

Figure 11: `IntIndexToIndexMultiMap` format

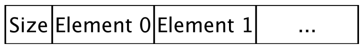

Figure 12: `IndexToIndexMap` format
Element size is implicit, because elements represent fixed-sized integer indexes.

Two aspects are worth mentioning with regard to document filtering. First, all the filtering predicates joined by boolean operators are applied according to the query syntax tree order to fulfill the initial requirement about latency stability and determinism (see Section 2). In other words, no query optimizer exists.

Second, short circuiting is implemented: as soon as it is obvious that the result is empty the query completes. This feature is extensively used with composite databases described below.

5.5 Partitioning

Composite database is a collection of simple databases. Composite database query execution engine runs the query through each of the underlying databases and merges the results taking into account optional sorting clauses.

In practice, it is advantageous to partition data based on real user queries and to rely on short circuiting implemented. For instance, if all the real user queries start with a filter on geographical region ID, then geo partitioning will substantially reduce the search space, because composite database query execution engine will quickly drop all the irrelevant simple databases due to short circuiting.

5.6 Memory

Current implementation of YoctoDB relies on read-only `mmap`ped files through Java `MappedByteBuffer` facilities. No data is loaded into Java heap apart from small metadata including field names, sizes and elements count. It means that almost no resources are spent on database reopening, so it is fast and cheap. Opening recently received database files is especially fast, because usually they reside in the write-through disk cache if the memory subsystem allows for it.

Query engine allocates a minimal sufficient number of `BitSet`s necessary for calculating and storing filtered document IDs when interpreting the query syntax tree. Each `BitSet` requires $1/8 * documents$ bytes. Sorting after filtering is implemented in a streaming way with respect to optional `skip` and `limit` constraints.

Typically, a request querying a database containing several million documents leads to allocating only about a megabyte of additional memory in Java heap which is quickly garbage collected because of the small amount of relatively large arrays. Contemporary garbage collectors with throughput of many GB/s make possible handling thousands of complex queries per second.

5.7 Thread Safety

Due to being immutable YoctoDB currently doesn't use any synchronization primitives which leads to perfect CPU scaling. `BitSet`s are allocated and used per thread per request.

The only issue that might arise in practice is the necessity for coordination when freeing database resources after switching to a freshly reopened database. This can be implemented using Java `Phaser` synchronization primitive. Nevertheless, the issue affects only the system maintenance not the query performance.

6. REFINEMENTS

This section explains low-level details of YoctoDB Java implementation currently being used in production and the lessons learned. We hope it will help contributors or anybody willing to implement anything similar.

6.1 64-bit

Due to Java `ByteBuffer` supporting only 32-bit addressing we had to implement own `Buffer` abstraction supporting 64-bit operations and delegating to a sequence of non-overlapping Java `ByteBuffer`s.

We hope that future Java versions will provide 64-bit addressing facilities, which will improve the performance and will make YoctoDB `Buffer` implementation redundant.

6.2 `BitSet` pools

It looked quite natural to implement `BitSet` pooling, possibly thread-local, to reuse the instances and to decrease GC load. Practical experiments showed no latency or CPU utilization profit gained.

This result might be caused by low cost of garbage collecting a small amount of short-lived arrays that underlie `BitSet` objects.

6.3 `unmap`

Current Java `MappedByteBuffer` implementation postpones real `unmap`ping until the buffer finalization stage which happens during garbage collecting it. YoctoDB object graphs are small, but relatively long-lived and are usually promoted to the old generation. At the same time, one of the goals of GC tuning is to make full GC pauses as infrequent as possible. It is obvious, that these aspects contradict each other.

We have observed accumulation of millions of file descriptors pointing to not yet `unmap`ped, but already non-existent files. When GC actually happened, we noticed huge JVM pauses due to millions of system `unmap`ping calls.

As a workaround we have to force invocation of the corresponding private `java.nio.DirectByteBuffer` `cleaner` calling `unmap` in application code as soon as the database stops being used[12]. This solution was borrowed from Apache Lucene `MMapDirectory`[13]. Unfortunately it will stop working since Java 9[14].

6.4 Roaring `BitSet`s

We have experimented with replacing persistent `BitSetIndexToIndexMultiMap` and `IntIndexToIndexMultiMap` (see Section 5.3) by Roaring bitmaps[15] described in [1, 10].

Unfortunately, Roaring Bit Set Java implementation we used[16] is not optimized for read-only databases. The overhead of reopening a huge number of persistent collections

[12]This feature has not been included into the open-sourced YoctoDB code base yet.
[13]https://github.com/apache/lucene-solr/blob /master/lucene/core/src/java/org/apache/lucene/store/MMapDirectory.java
[14]https://issues.apache.org/jira/browse/LUCENE-6989
[15]http://roaringbitmap.org
[16]https://github.com/RoaringBitmap/RoaringBitmap

based on `ImmutableRoaringBitmap` was memory prohibitive due to noticeable heap pollution. We might consider reimplementing the approach with read-only optimization in mind.

6.5 Versioning

We would like to highlight the importance of proper versioning of persistent formats. Currently YoctoDB format version stored in the header is incremented as soon as any non-compatible persistence change occurs, for example, a new segment format is added or the existing persistent collection layout is changed. We plan to support backward compatibility in future YoctoDB releases, but at the moment changes are reflected in the `CHANGELOG`[17] and the format version is checked at runtime when opening a YoctoDB database for reading.

Real applications store YoctoDB partitions in directory structures prefixed with integer database schema version. This allows easy downgrading after having switched to a new version and discovered any runtime problems, but it requires implementing rotation of legacy version data files.

6.6 Document Building

To make the process of YoctoDB document construction less tedious and error-prone we developed an automatic converter of annotated Java classes to YoctoDB documents using Java reflection facilities. This feature substantially improved code maintainability because application model classes and mapping to YoctoDB data model are co-located.

We hope to release `yoctodb-converter` as a separate open-source project.

6.7 Index Transport

Figure 1 shows the role of YoctoDB in the document processing pipeline. Each YoctoDB database built by an *Indexer* process needs to be delivered to each *Searcher* node. In our deployment there are multiple *Searcher* nodes in many geographically distributed datacenters.

If an *Indexer* process sends databases to each one of *Searcher* node itself, it can easily overflow the local network interface making it a bottleneck. Besides, inter-datacenter links have limited network throughput (compared to intra-datacenter fabric) and higher latencies.

We developed a reliable transport mechanism[18] optimizing inter-datacenter traffic and increasing scalability of the system that works the following way:

1. Each *Searcher* instance registers in Discovery Service[19] in the group corresponding to local datacenter
2. After building a YoctoDB database *Indexer* process sends it to a random *Searcher* node in each datacenter
3. Each *Searcher* unit having received from *Indexer* and persisted a YoctoDB database retransmits it to other *Searcher* nodes in local datacenter

Therefore a YoctoDB database is sent by *Indexer* only to a small number of *Searcher* nodes limited by the number of datacenters. Each *Searcher* instance uses fast intra-datacenter links to deliver the database of other local nodes.

[17]https://github.com/yandex/yoctodb/blob/master /CHANGELOG
[18]As an Akka Extension: http://doc.akka.io/docs/
[19]Currently we use Apache Zookeeper for service discovery: http://zookeeper.apache.org

Table 1: Synthetic key-value performance (μs)

Database	Mean	Max	95%
YoctoDB	30	162	47
Lucene	156	505	174
H2	1858	2342	2001
SQLite	1930	2597	2186

If the number of nodes in each datacenter is too high, the same approach can be used to implement a rack aware transport.

7. REAL APPLICATIONS

The first application to integrate YoctoDB was Yandex.Auto which originally had been using Apache Lucene as a search backend. Before initiating a full-blown YoctoDB project we had built a prototype to evaluate several alternatives including H2[20] and SQLite[21].

7.1 Synthetic Performance

The slowest Yandex.Auto search queries filter a large portion of the documents and then aggregate the results, for instance, group the documents and calculate field value ranges. At the same time, these queries happen quite often.

We have compared the database engines using a simple synthetic benchmark to test the performance in read-only mode with dataset size similar to real use cases. The data model is a table with two columns:

- `id` — primary key storing a row number
- `f` — indexed field containing `md5(id % 10)` value

`SELECT COUNT(id) FROM test WHERE f = ?` is used as a query.

The test machine has the following configuration:

- 2 x Intel Xeon E5-2660
- 128 GB DDR3 RAM
- 300 GB SSD
- Ubuntu 12.04.4 LTS

Table 1 shows the results for databases containing 1M documents.

7.2 Yandex.Auto

Having implemented and integrated YoctoDB into Yandex.Auto we started comparing it with the previous Lucene-based version of the application using real user queries in equal environment. The search index contains approximately 2M documents divided into 1024 partitions placed onto SSD.

Table 2 shows response time under constant 50 rps load. YoctoDB percentiles are 2x better and it consumes 2x less CPU at the same time. Also we noticed that CPU load is much steadier in the YoctoDB case.

Under extreme load the YoctoDB implementation handles 400 rps and the Lucene based approach handles 170 rps with metrics conforming to service level objectives.

[20]http://www.h2database.com
[21]http://www.sqlite.org/

Table 2: Constant 50 rps Yandex.Auto latency (ms)

Percentile	YoctoDB	Lucene
98%	150	300
95%	100	200
75%	45	100
mean	28	50

Table 3: Auto.ru 500 rps latency (ms)

Metric	50%	75%	90%	95%	99%	99.9%
Value	6	20	40	50	70	200

Index size in YoctoDB case occupies 2.3 GB. Apache Lucene database needs 3.4 GB. Raw document payloads serialized to Protobuf in both cases occupy 1.8 GB.

All the metrics listed above were measured using early YoctoDB releases. Latest YoctoDB versions demonstrate even higher performance which is supported by stress tests in Auto.ru described below.

7.3 Auto.ru

Auto.ru search index contains 0.5 M documents divided into 1024 partitions occupying 3-4 GB in total. New partition updates are incorporated into the index every minute. All the partitions are updated every several minutes. Each document has a payload containing car offer serialized using Protobuf and almost hundred filterable and/or sortable fields.

Table 3 shows YoctoDB response time under constant 500 rps load. CPU usage doesn't exceed 30% despite YoctoDB partitions being received and reopened in background.

The critical load YoctoDB can handle using real Auto.ru requests while conforming to the SLA is 1200 rps (see Figure 13). Having reached this limit YoctoDB saturates CPU and the latency noticeably degrades (see Figure 14).

Queries

It is not obvious, but many of the aforementioned HTTP requests actually transform into multiple and often sequential YoctoDB queries.

Some typical query examples:

- Find car offers in a geo bounding box
- Build search result page with commercial offers using several queries with additional filters and sorts and subsequent merging of the results

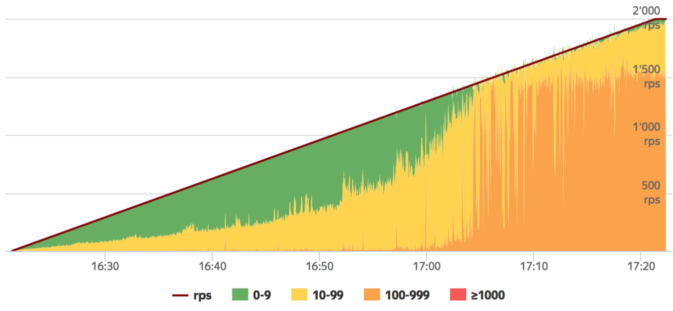

Figure 13: Auto.ru response time under load (ms)

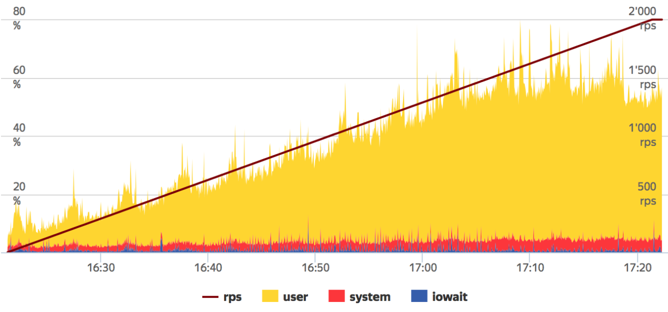

Figure 14: Auto.ru CPU usage under load

- Find car offers in YoctoDB offer index, find car configurations in YoctoDB car catalog, join the sets using `configuration_id`

7.4 Other

YoctoDB is used in many other services including:

- Yandex.Realty[22] database of buildings
- Yandex.Travel[23] partitioned hotel database
- Read-only resources that need periodic rebuilding and distributing: dictionaries, catalogues, lists, etc.
- Auxiliary resources used in Hadoop Map-Reduce jobs and delivered through Distributed Cache

8. FUTURE WORK

We are going to describe some features we consider implementing in the nearest future.

Tracing and Introspection

YoctoDB is definitely lacking tools for database binary file introspection and conversion to human readable formats. Dynamic query tracing capabilities are rather limited yet. This toolchain will help developers with finding and fixing inefficiencies in database representation and querying.

Better Collections

There is a huge space for improving YoctoDB performance by implementing more compact and/or faster cache friendly persistent collections. We plan to improve binary value search in sorted sequence of values by implementing persistent cache friendly tree-based or sampled data structures that look promising [7, 6, 3, 11]. Another direction of possible optimization is leveraging CPU SIMD instructions[14].

Parallel Querying

Parallel querying a partitioned database might be advantageous in certain use cases when a client is willing to spend more CPU time to improve query latency. Parallel filtering can be easily implemented in the current architecture using Java `ForkJoinPool`. However, `BitSets` will become shared between different threads during the merge stage that might cause some performance penalties. This feature demands more experiments and measurements on real load.

[22]https://realty.yandex.ru
[23]https://travel.yandex.ru

Block Compression

Documents in YoctoDB partitions naturally share common payload subsequences, so it might be useful to implement some type of fast compression. We plan to implement optional payload block compression using LZ4 or Snappy codec and evaluate the effect.

Optimized Roaring Bit Sets

As described in Section 6.4 the existing implementation of the concept didn't produce any profit in YoctoDB case. The idea still looks promising and we are going to implement a prototype of Java heap friendly roaring bit set to better compress document IDs and to improve query performance.

Bloom Filters

Bloom filters might be advantageous for speeding up a document lookup using equality condition with highly selective document field, for instance, user provided document IDs, due to fast dropping of partitions not containing the document.

Database Merge

Finally, YoctoDB has been designed with possibilities for implementation of database merge algorithms in mind. It might be useful from the maintenance point of view, because constructing a YoctoDB database requires Java heap size linear of the database size. Managing and tuning large heap may be troublesome, so it might be easier to build smaller partitions and then merge them into bigger ones. Merge process may be implemented by joining payload segments without extra memory and remapping IDs in filtering and sorting segments using much smaller amount of memory.

9. RELATED WORK

There are many implementations of immutable key-value file storages stemming from log-structured merge-tree approach [12] made popular by Google Bigtable [2], Apache Cassandra[24] [9] and Apache HBase[25] [5]. Embedded databases implementing the same approach exist, for instance, LevelDB[26].

Despite being popular, these solutions are designed to achieve different goals including high write throughput and low latency of key-value requests. At the same time YoctoDB provides a facility for complex queries including many indexed and sorted fields without spending much memory for request processing. Nevertheless it might be possible to replace YoctoDB filtering and sorting segments with SSTables (one SSTable per each indexed field).

Foundational results for universal in memory database systems exist [4], but most of these approaches are focused on mutable relational databases. These systems have to use safe in the worst case algorithms, because they don't know the distribution and amount of data in advance and they try to optimize IO. YoctoDB is able to choose optimal data structures and query execution strategies, because much more information about data is available. Usually SQL DBMSes are IO bound, while YoctoDB is CPU bound [8].

[24]http://cassandra.apache.org
[25]http://hbase.apache.org
[26]https://github.com/google/leveldb

Finally, there are many notable results in persistent data structures [11], but they are focused on effective sharing of substructures when mutating collections. On the contrary, YoctoDB builds efficient immutable data structures once and queries them multiple times.

10. CONCLUSIONS

We have described YoctoDB, an embedded engine for extremely fast partitioned immutable-after-construction databases. YoctoDB has been used in production since 2014 and still obtains new customers.

Despite being fast, YoctoDB has several limitations rooting in the design that might affect its usage and need to be specified clearly. There is no enforced schema support yet. YoctoDB core represents all values as byte arrays, so a user might mix them up by using a wrong conversion or a wrong field name without getting any runtime error. There is no query optimizer, so a user should be careful when constructing queries. Nested queries are not supported, that might be a limiting factor. Finally, all the presented benchmarks work with databases that fit into disk cache.

Having replaced Apache Lucene by YoctoDB we have substantially improved application latencies and got rid of tens of machines due to much higher throughput of each node. Nevertheless, there are vast opportunities for future improvements we are going to work on.

11. ACKNOWLEDGEMENTS

We thank the anonymous reviewers. Also we thank Dmitry Kachmar for believing in the project, assigning resources and making it possible. YoctoDB benefited greatly from the feedback of many users within Yandex including but not limited to Vladislav Dolbilov, Ivan Kozhevnikov, Petr Kerzum, Ruslan Muzhikov.

12. REFERENCES

[1] S. Chambi, D. Lemire, O. Kaser, and R. Godin. Better bitmap performance with roaring bitmaps. *Software: practice and experience*, 2015.

[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[3] D. Comer. Ubiquitous b-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

[4] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. *Implementation techniques for main memory database systems*, volume 14. ACM, 1984.

[5] L. George. *HBase: the definitive guide*. O'Reilly Media, Inc., 2011.

[6] G. Graefe. Modern b-tree techniques. *Foundations and Trends in Databases*, 3(4):203–402, 2011.

[7] J. E. Hopcroft. *Data structures and algorithms*. Addison-Wesley Boston, MA, USA, 1983.

[8] M. Kleppmann. *Designing data-intensive applications*. O'Reilly Media, 2016.

[9] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[10] D. Lemire, G. Ssi-Yan-Kai, and O. Kaser. Consistently faster and smaller compressed bitmaps with roaring. *Software: Practice and Experience*, 2016.

[11] C. Okasaki. *Purely functional data structures.* Cambridge University Press, 1999.

[12] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[13] V. Tsesko. Akka at Yandex (in Russian) // JPoint 2014. http://2014.javapoint.ru/talks/07/, Apr. 2014. [Online; accessed 18-July-2016].

[14] J. Zhou and K. A. Ross. Implementing database operations using simd instructions. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 145–156. ACM, 2002.

Listing 3: YoctoDB session example

```
// Get current database format
DatabaseFormat format =
  DatabaseFormat.getCurrent();

// Create a mutable database
DatabaseBuilder dbBuilder =
  format.newDatabaseBuilder();

// Add first document
dbBuilder.merge(
  format.newDocumentBuilder()
    .withField("id", 1, FILTERABLE)
    .withField("score", 0, SORTABLE)
    .withPayload("payload1".getBytes()));

// Add another document
dbBuilder.merge(
  format.newDocumentBuilder()
    .withField("id", 2, FILTERABLE)
    .withField("score", 1, SORTABLE)
    .withPayload("payload2".getBytes()));

// Build and serialize the immutable database
ByteArrayOutputStream os =
  new ByteArrayOutputStream();
dbBuilder.buildWritable().writeTo(os);

// Open the immutable database
Database db =
  format.getDatabaseReader().from(
    Buffer.from(
      os.toByteArray()));

// Find the second document
Query doc2 =
  select().where(eq("id", from(2)));
assertTrue(db.count(doc2) == 1);

// Filter and sort

Query sorted =
  select().where(
    and(
      gte("id", from(1)),
      lte("id", from(2))))
    .orderBy(desc("score"));

List<Integer> ids =
  new LinkedList<Integer>();
db.execute(
        sorted,
        new DocumentProcessor() {
            @Override
            public boolean process(
                    final int document,
                    final Database database) {
                ids.add(document);
                return true;
            }
        });

assertEquals(asList(1, 0), ids);
```