Yandex

Yandex

String optimization in ClickHouse

Nikolai Kochetov
ClickHouse developer

String storage in ClickHouse

String datatypes

String

- Default case
- Overhead — 9 bytes per string (in RAM)
- Use it till it’s fast enough

FixedString

- If size in bytes is fixed and never changes (IP, MD5)
- Arbitrary binary data

String datatypes

Queries from tables with the same data.

```
SELECT sum(ignore(val)) FROM table_1

Processed 1.00 billion rows, 4.00 GB (1.86 billion rows/s., 7.46 GB/s.)

SELECT sum(ignore(val)) FROM table_2

Processed 1.00 billion rows, 17.89 GB (683.57 million rows/s., 12.23 GB/s.)
```

> Tables store first billion numbers into UInt64 and String types

String datatypes

Compressed data size

```
SELECT
    table,
    formatReadableSize(sum(data_compressed_bytes)) AS compressed_size
FROM system.parts
WHERE active AND (table LIKE 'table_%')
GROUP BY table
```

```
┌─table───┬─compressed_size─┐
│ table_1 │ 3.90 GiB        │
│ table_2 │ 3.74 GiB        │
└─────────┴─────────────────┘
```

> The second query deals with string decompression

Low granularity strings

Enum8, Enum16

- Set of strings is known beforehand
- Set of strings (almost) never changes

Advantages

- Storage and processing numeric data
- Cheap GROUP BY, IN, DISTINCT, ORDER BY
- optimized for individual cases (e.g. comparison with constant string)

Disadvantages

- Altering the datatype

ALTER Enum

Why can it be slow?

- Enum structure is stored into a table scheme
- Wait for selects to be able to change structure

Can we do better?

- Store Enum structure somewhere else (ZooKeeper)
- Do not wait for selects just in this case

Possible problems

- Synchronization
- Fetching a part with new data from another replica

External dictionaries

Store strings in a dictionary, indices in a table

Advantages

- Dynamically changeable set of strings
- No alterations (no problems)
- A variety of dictionary sources

Disadvantages

- Bulky (explicit) syntax
- Difficult to optimize
- Delayed updates from external source

Local dictionaries

Getting rid of global dictionaries

No synchronization — no problem

Store dictionaries locally

- Per block (in memory)
- Per part (on file system)
- In caches (during query processing)

Dictionary encoded strings

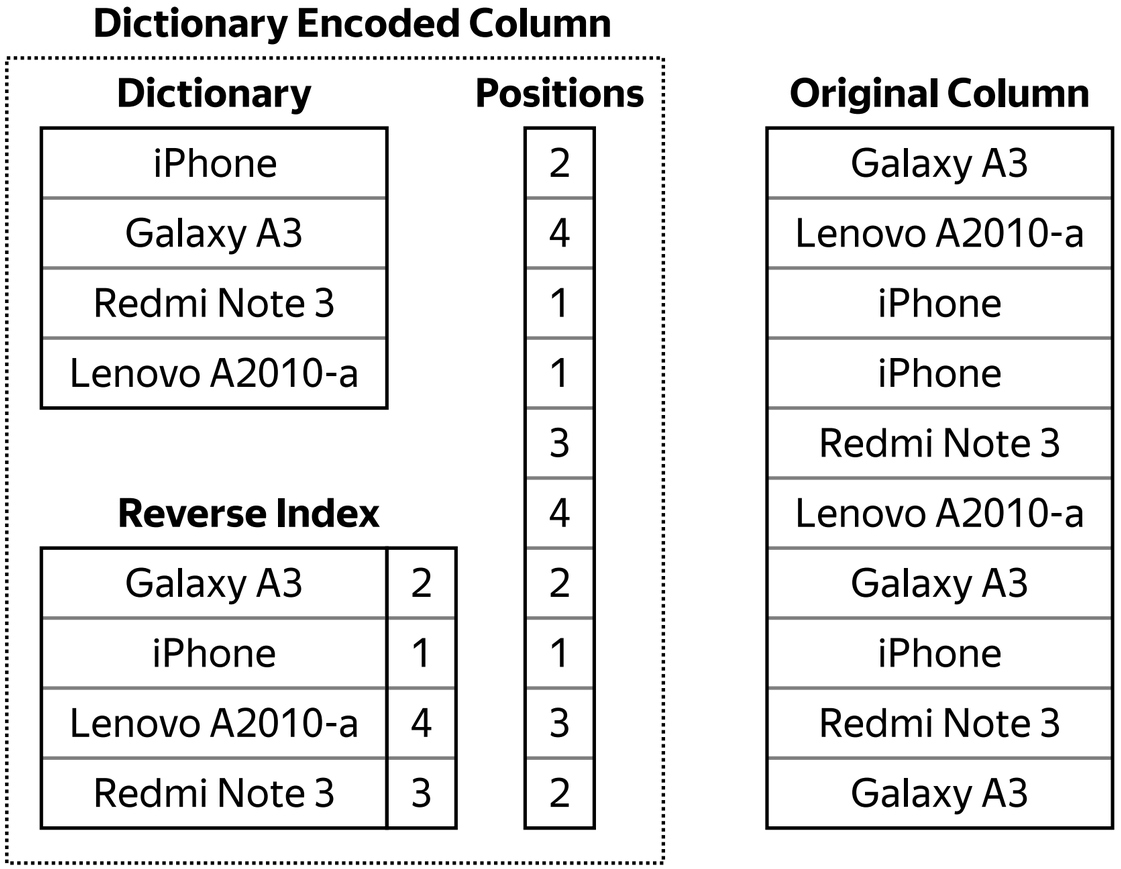
StringWithDictionary

Datatype for dictionary encoded strings

- Serialization
- Representation in memory
- Data processing

Content:

- Dictionary
- Column with positions
- Reversed index



LowCardinality(Type)

Is a general datatype with dictionary encoding

- Is implemented for strings, numbers, Date, DateTime, Nullable.
- StringWithDictionary is an alias for LowCardinality(String).
- Remains for some functions

```
SELECT
    toLowCardinality('') AS s,
    toTypeName(s),
    toTypeName(length(s))
```

```
┌─s─┬─toTypeName(toLowCardinality(''))─┬─toTypeName(length(toLowCardinality('')))─┐
│   │ LowCardinality(String)           │ LowCardinality(UInt64)                   │
└───┴──────────────────────────────────┴──────────────────────────────────────────┘
```

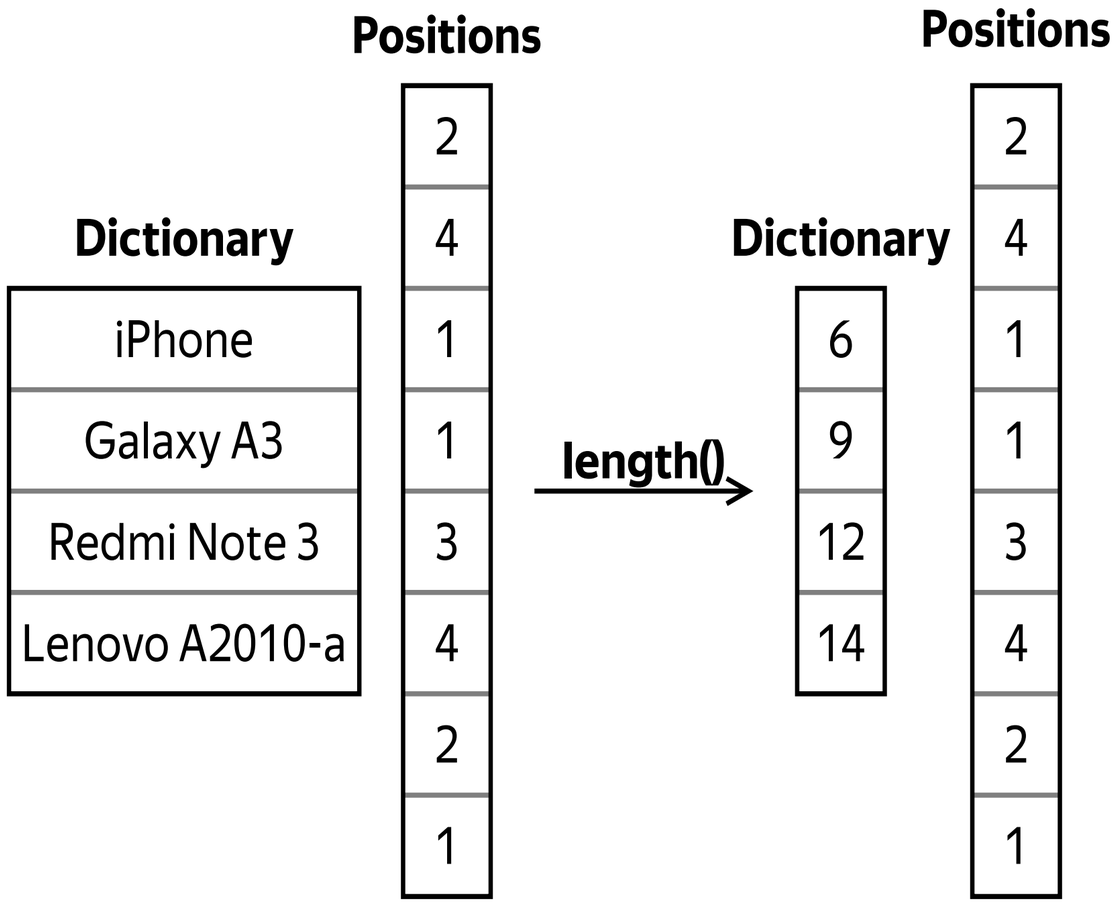
Queries optimizations

Implemented

- Functions executed on dictionaries if it’s possible
- Calculations are cached for same dictionaries
- GROUP BY optimization

To be done

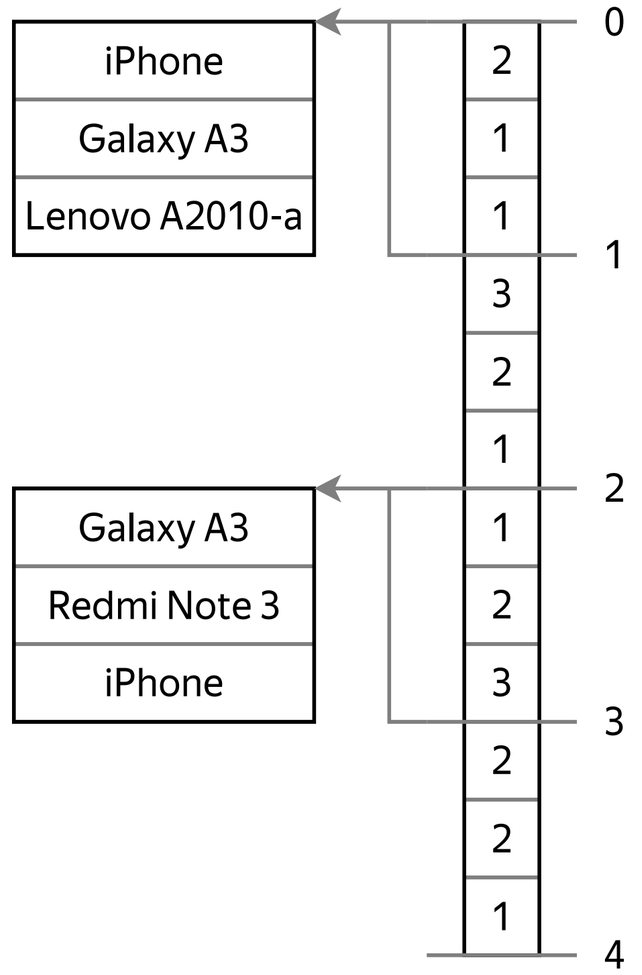
- Specializations for aggregate functions



High cardinality strings

What if we insert a lot of different strings?

- Serialization limit: low_cardinality_max_dictionary_size
- Store excessive keys locally
- Fall back to ordinary column (in plans)



Storage volume

Can we decrease it?

Column	COUNT DISTINCT	String	Dictionary	Enum
CodePage	62	72.18 MiB	26.97 MiB	26.20 MiB
PhoneModel	48044	439.20 MiB	440.61 MiB	-
URL	137103569	13.15 GiB	11.28 GiB	-

lz4, zstd use dictionary encoding

Performance estimation

TLC trip record dataset

Dataset with NYC taxi and Uber trip data
https://github.com/toddwschneider/nyc-taxi-data

More than 1.1 billion trips from January 2009 to July 2015

- Start and end time of the trip
- Location names
- Payment type
- The number of passengers
- Taxi type (yellow taxi, green taxi, Uber)

TLC trip record dataset

> What is the most popular pickup place?

```
SELECT pickup_ntaname FROM trips
GROUP BY pickup_ntaname
ORDER BY count() DESC
```

pickup_ntaname	count()
Midtown-Midtown South	
Hudson Yards-Chelsea-Flatiron-Union Square	
West Village	
Upper East Side-Carnegie Hill	
Turtle Bay-East Midtown	
SoHo-TriBeCa-Civic Center-Little Italy	
Upper West Side	
Murray Hill-Kips Bay	
Clinton	
Lenox Hill-Roosevelt Island	

TLC trip record dataset

Store pickup locations into 3 different types:

- String
- StringWithDictionary
- Enum16

Query	String	Dictionary	Enum16
Most popular location	4.890 sec.	0.548 sec.	0.783 sec.

TLC trip record dataset

> Where is the most popular park?

```
SELECT pickup_ntaname FROM trips
WHERE lower(pickup_ntaname) like '%park%'
GROUP BY pickup_ntaname
ORDER BY count() DESC
```

```
┌─pickup_ntaname─────────────────────────┐
 Battery Park City-Lower Manhattan
 park-cemetery-etc-Manhattan
 Park Slope-Gowanus
 park-cemetery-etc-Queens
 Rego Park
 Sunset Park West
 park-cemetery-etc-Brooklyn
 Baisley Park
 Bedford Park-Fordham North
```

TLC trip record dataset

Query	String	Dictionary	Enum16
Most popular location	4.890 sec.	0.548 sec.	0.783 sec.
Most popular park	3.934 sec.	0.440 sec.	4.776 sec.

Why is query with Enum is slow?

- LIKE is not optimized for Enum
- Enum is converted to string

Enum needs manual optimization in code

TLC trip record dataset

The number of different locations.

```
SELECT uniq(pickup_ntaname) FROM trips
```

```
┌─uniq(pickup_ntaname)─┐
│                  196 │
└──────────────────────┘
```

The number of different locations in Manhattan

```
SELECT uniq(pickup_ntaname) FROM trips where pickup_boroname='Manhattan'
```

```
┌─uniq(pickup_ntaname)─┐
│                   29 │
└──────────────────────┘
```

TLC trip record dataset

Query	String	Dictionary	Enum16
Most popular location	4.890 sec.	0.548 sec.	0.783 sec.
Most popular park	3.934 sec.	0.440 sec.	4.776 sec.
Unique locations	4.136 sec.	3.432 sec.	1.050 sec.
Unique locations in Manhattan	5.425 sec.	3.497 sec.	1.328 sec.

Why is the last query is two times faster for StringWithDictionary?

StringWithDictionary filtration works only for indices

TLC trip record dataset

Slow function example

```
SELECT
    hex(SHA256(pickup_ntaname)) AS hash,
    count()
FROM trips_dict
GROUP BY hash
ORDER BY count() DESC
```

```
┌─hash─────────────────────────────────────────────────────────────┬─count()─┐
│ 924AAA8D24075B327D16A53E39EE56FFA33AD8A3FE822F647A7E3765CD754DCA │ 207582585 │
│ B1E4D0E42D25F1341D9AA327CD59838B29F31D09CC34C9A25287679DD19359B2 │ 114945944 │
│ EBA433E6A9487BD2030D4623D86330B8C89C60319E410FBE035450D63CD92652 │  88277252 │
│ E4EEEEA4D816773D94F09BE59144EC1EE7B65052B040689606237D3F8EE18344 │  86192276 │
│ 9E74963DCB63099B44C7AD5B132F9144D80C6A4E1776B2DFB0B502A1CB5E853D │  83692525 │
│ FB19C1C65FE9F2490C4D9AE45FD40679375CB26F289075420C33C8C4A318C046 │  62524265 │
└──────────────────────────────────────────────────────────────────┴─────────┘
```

TLC trip record dataset

Query	String	Dictionary	Enum16
Most popular location	4.890 sec.	0.548 sec.	0.783 sec.
Most popular park	3.934 sec.	0.440 sec.	4.776 sec.
Unique locations	4.136 sec.	3.432 sec.	1.050 sec.
Unique locations in Manhattan	5.425 sec.	3.497 sec.	1.328 sec.
Slow function	31.566 sec.	2.440 sec.	32.608 sec.

Summary

LowCardinality type is available in last release

- Experimental (`set allow_experimental_low_cardinality_type = 1` to enable)
- Test performance on your dataset
- Just replace `String` with `StringWithDictionary`

Goals

- Make datatype with dictionary better than String in all cases
- **Implicitly replace String with StringWithDictionary**