Sparse data storage and query patterns in Clickhouse

Amit Sanjay Sadafule

About Me

Amit Sanjay Sadafule

amit-sadafule-366b7225

@amit.sadafule

Co-founder and Head of Technology @Manthhan

Built and scaled platforms handling millions of vehicle data-points for India’s largest automobile manufacturer. Real-time track & trace systems powering vehicle production plants across the country.

- Love challenges of distributed systems
- Pragmatic Engineer
- Treks and exploration

Happy to jam on distributed systems, code quality, or trekking stories.
Coffee ☕ invitations always welcome!

The Problem

Store and query data from thousands of vehicles

Each vehicle can transmit nearly 5,000 keys every minute, referred to as PIDs

Characteristics of these PIDs:

- PID Data is Time-Series Data
- Sparse Data
- 3 data types: Integer, Float and String
- Consistent Data Types
- Append-Only Data

Query Pattern

Retrieving PID Values for a group of vehicles within a specified time range

Aggregation Queries: average, count, sum on PID values.

But... What is Sparse Data?

OBD data					
vehicle_id	timestamp	p1	p2	...	pn
1	123456	x	x		
2	123456		x		
3	123456	x			x
...	...				
n	123456				x

How to efficiently store the pid data and retrieve it?

Choose right schema

Horizontal (Wide) Table Structure

Vertical Table Structure

horizontal table structure					
vehicle_id	timestamp	p1	p2	...	pn
4	1686830420666	x		...	y

verticle table structure			
vehicle_id	timestamp	pid	value
4	1686830420666	p1	x
...	...	...	...
4	1686830420666	pn	y

Option 1 - Sparse Matrix

sparse data					
vehicle_id	timestamp	p1	p2	...	pn
1	1686830420666	x	x		
2	1686830420666		x		
3	1686830420666	x			x
...	...				
n	1686830420666				x

Pros:

- Different compression algorithms per column
- Primary key can consists of vehicle id and timestamp

Cons:

- High RAM requirements
- Slower querying on PIDs

Option 2 - Separate Tables for Each Data Type

integer data			
vehicle_id	timestamp	pid	value
1	1686830420666	p1	22
2	1686830420666	p2	36
3	1686830420666	p44	1
...	...		
n	1686830420666	p1888	31

string data			
vehicle_id	timestamp	pid	value
1	1686830420666	p50	"r"
2	1686830420666	p60	"an"
3	1686830420666	p61	"do"
...	...		
n	1686830420666	p1999	"m"

float data			
vehicle_id	timestamp	pid	value
1	1686830420666	p40	12.344534
2	1686830420666	p1490	34.2323
3	1686830420666	p1600	1.00
...	...		
n	1686830420666	p876	43.12

Pros:

- Faster queries
- Significantly lower RAM usage during data insertion
- No sparseness

Cons:

- Caller code must distinguish between different tables
- Joins may be required to get PIDs of different data types

Option 3 - Data Type-Specific Columns in the Same Table

data					
vehicle_id	timestamp	pid	int_value	float_value	string_value
1	1686830420666	p1	22	NULL	NULL
2	1686830420666	p2	36	NULL	NULL
3	1686830420666	p40	NULL	51.6	NULL
4	1686830420666	p1490	NULL	413.25	NULL
...	...	...	...	...	...
1200	1686830420670	p50	NULL	NULL	"r"
1201	1686830420670	p60	NULL	NULL	"an"
...	...	...	...	...	...
n	1686830420670	p369	31	NULL	NULL

Pros:

- Simplified caller code
- Faster queries

Cons:

- Potential sparseness in the data
- Higher RAM requirements during data insertion than Option 2

Optimizations and Best Practices

Avoid Nullable Columns

Using default values instead of nulls can improve query performance. This because without default values ClickHouse have to store and refer to extra storage required for null references.

Compression

Efficient compression algorithms like ZSTD can significantly reduce storage requirements. Ensure that the chosen compression method aligns with the characteristics of your data.

Query Performance

Optimize queries by ensuring the primary key includes the PID, which can significantly speed up query performance.

Option 3.1 - Data Type-Specific Columns in the Same Table - with default values

data					
vehicle_id	timestamp	pid	int_value	float_value	string_value
1	1686830420666	p1	22	0.0	""
2	1686830420666	p2	36	0.0	""
3	1686830420666	p40	0	51.6	""
4	1686830420666	p1490	0	413.25	""
...	...	...	...	...	...
1200	1686830420670	p50	0	0.0	"r"
1201	1686830420670	p60	0	0.0	"an"
...	...	...	...	...	...
n	1686830420670	p369	31	0.0	""

Pros:

- Simplified caller code
- Faster queries as null values are avoided

Cons:

- Higher RAM requirements during data insertion than Option 2

Experimental analysis

Data Generation: 2000 PIDs per vehicle, with 70% float values, 29% integer values, and 1% string values.

Data Volume: 3 billion PID values for 1000 vehicles over a 24-hour period.

Machine Setup: Mac OS Sonoma, Apple M3 MacBook Air, 16GB RAM, 8 cores.

ClickHouse Version: v24.8.1.

Experimental analysis - Query Performance

Get all PIDs and values for a vehicle and for entire 2hr duration

Table	Option	Rows scanned	Max RAM used (MiB)	Time taken (Sec.)	Uncompressed data size (MB)
pid_float	Option 2	196.61 thousand	4.23	0.007	7.22
pid_integer	Option 2	73.73 thousand	1.05	0.005	2.86
pid_string	Option 2	16.38 thousand	73.53 KiB	0.007	425.91 KB
pid_without_defaults	Option 3	262.14 thousand	4.08	0.253	14.57
pid_with_defaults	Option 3.1	253.95 thousand	1.08	0.169	10.88

References

[ClickHouse - Lightning Fast Analytics for Everyone]
https://www.vldb.org/pvldb/vol17/p3731-schulze.pdf

[Extending RDBMSs To Support Sparse Datasets]
https://pages.cs.wisc.edu/~alanh/sparse.pdf

[Too Wide or Not Too Wide | That is the ClickHouse Question]
https://altinity.com/blog/too-wide-or-not-too-wide-that-is-the-clickhouse-question

[Avoid Nullable Columns]
https://clickhouse.com/docs/en/cloud/bestpractices/avoid-nullable-columns