Getting started with ClickHouse

Here are 13 "Deadly Sins" and how to avoid them

November 2023

ClickHouse

Introduction to ClickHouse

ClickHouse Cofounders

Aaron Katz

CEO @ ClickHouse

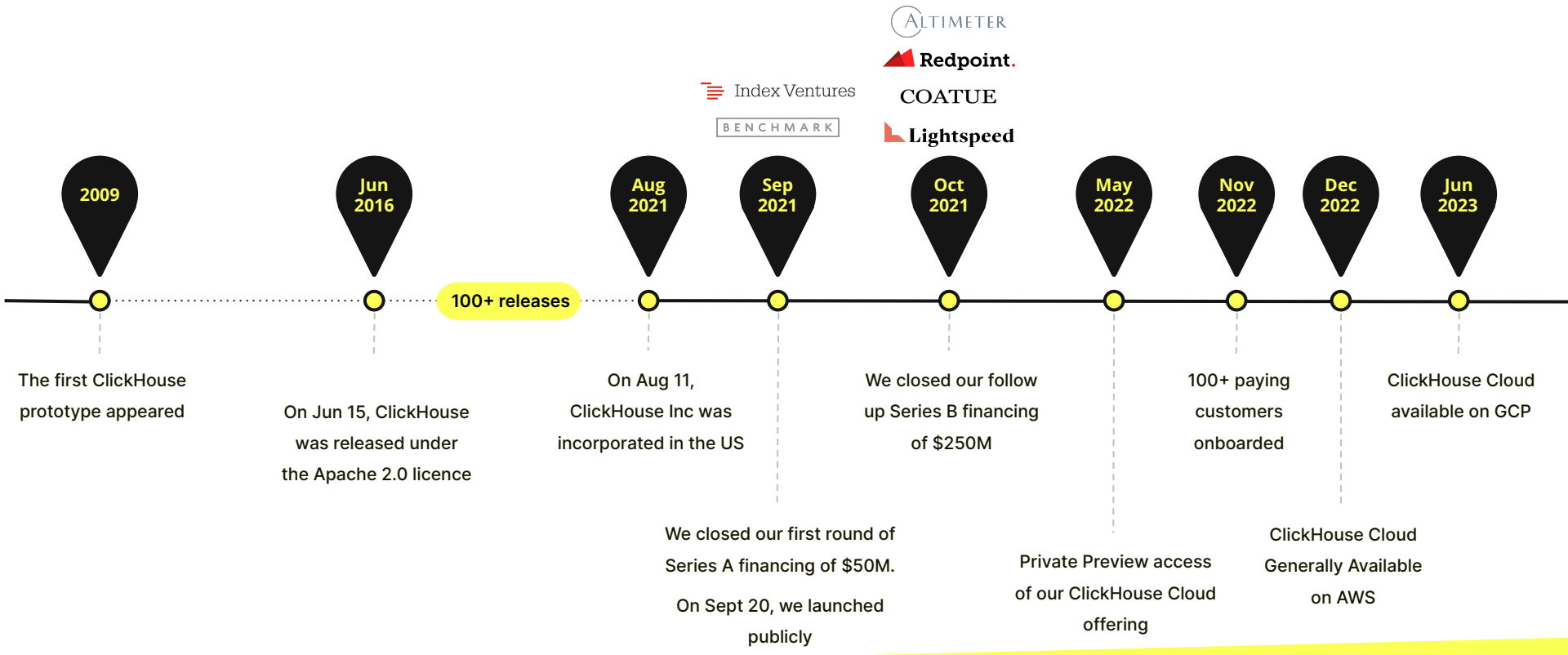
Alexey Milovidov

CTO @ ClickHouse

Yury Izrailevsky

President &
VP of Engineering

The ClickHouse Journey



What is ClickHouse?

Open source

- Developed since 2009
- OSS 2016
- 31k+ Github stars
- 1.3k+ contributors
- 500+ releases

column-oriented

- Best for aggregations
- Files per column
- Sorting and indexing
- Background merges

distributed

- Replication
- Sharding
- Multi-master
- Cross-region

OLAP database

- Analytics use cases
- Aggregations
- Visualizations
- Mostly immutable data

Who uses ClickHouse?

... & thousands more

13 "Deadly Sins" and how to avoid them

Blog / Engineering

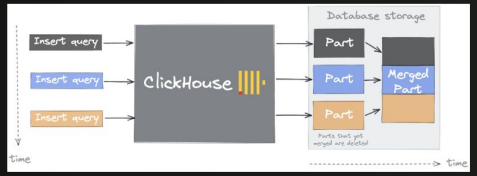
Getting started with ClickHouse? Here are 13 "Deadly Sins" and how to avoid them

Dale McDiarmid, Tom Schreiber & Geoff Genz
Oct 26, 2022

1. Too many parts

An often-seen ClickHouse error, this usually points to incorrect ClickHouse usage and lack of adherence to best practices. This error will often be experienced when inserting data and will be present in ClickHouse logs or in a response to an INSERT request. To understand this error, users need to have a basic understanding of the concept of a part in ClickHouse.

A table in ClickHouse consists of data parts sorted by the user's specified primary key (by default, the ORDER BY clause on table creation but see Index Design for the details). When data is inserted in a table, separate data parts are created, and each of them is lexicographically sorted by primary key. For example, if the primary key is `(CounterID, Date)`, the data in the part is sorted first by `CounterID`, and within each `CounterID` value by `Date`. In the background, ClickHouse merges data parts for more efficient storage, similar to a Log-structured merge tree. Each part has its own primary index to allow efficient scanning and identification of where values lie within the parts. When parts are merged, then the merged part's primary indexes are also merged.



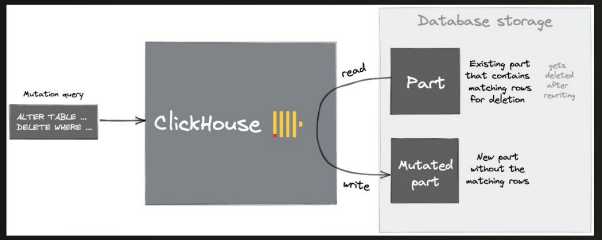
As the number of parts increases, queries invariably will slow as a result of the need to evaluate more indices and read more files. Users may also experience slow startup times in cases where the part count is high. The creation of too many parts thus results in more internal merges and "pressure" to keep the number of parts low and query performance high. While merges are concurrent, in cases of misuse or misconfiguration, the number of parts can exceed internal configurable limits[1][2]. While these limits can be adjusted, at the expense of query performance, the need to do so will more often point to issues with your usage patterns. As well as causing query performance to degrade, high part counts can also place greater pressure on ClickHouse Keeper in replicated configurations.

So, how is it possible to have too many of these parts?

3. Mutation Pain

While rare in OLAP use cases, the need to modify data is sometimes unavoidable. To address this requirement, ClickHouse offers mutation functionality which allows users to modify inserted data through ALTER queries. ClickHouse performs best on immutable data, and any design pattern which requires data to be updated post-insert should be reviewed carefully.

Internally, mutations work by rewriting whole data parts. This process relies on the same thread pool as merges. Note also that the mutation needs to be applied on all replicas by default. For this reason, mutations are both CPU and IO-intensive and should be scheduled cautiously with permission to run limited to administrators. Resource pressure as a result of mutations manifests itself in several ways. Typically, normally scheduled merges accumulate, which in turn causes our earlier "too many parts" issue. Furthermore, users may experience replication delays. The system.mutations table should give administrators an indication of currently scheduled mutations. Note that mutations can be cancelled, but not rolled back, with the KILL MUTATION query.



https://clickhouse.com/blog/common-getting-started-issues-with-clickhouse

Agenda

01 Too many parts

02 Going horizontal too early

03 Mutation Pain

04 Unnecessary use of complex types

05 Deduplication at insert time

06 Poor Primary Key Selection

07 Overuse of Data Skipping indices

08 LIMIT doesn’t always short circuit + point lookups

09 IP Filtering in Cloud

10 Readonly tables

11 Memory Limit Exceeded for Query

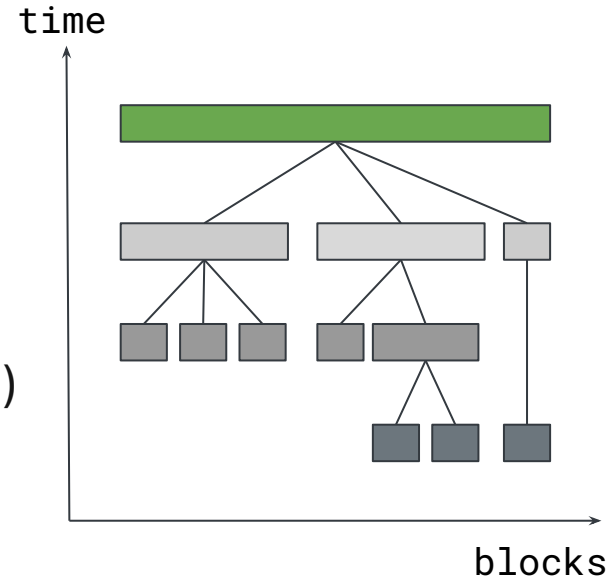
12 Issues relating to Materialized Views

13 Experimental features in production

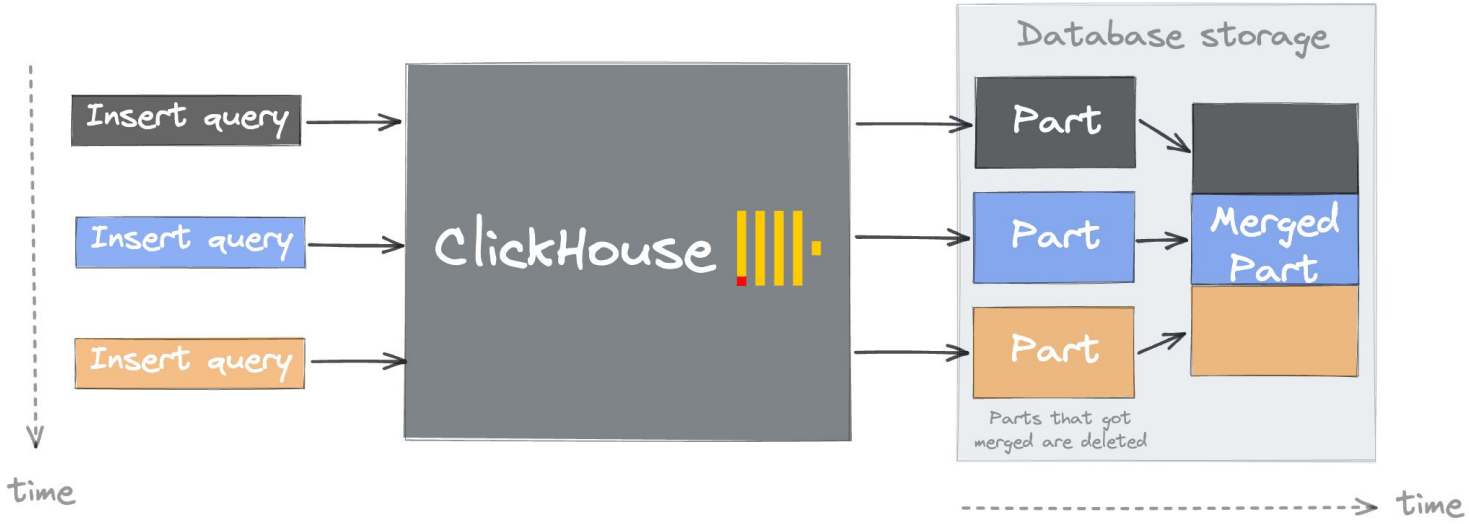
1. Too many parts - MergeTree Data Structure

```
DB::Exception: Too many parts (300). Merges are processing significantly slower than inserts
```

- Main persistent table engine
- Supports sorting key
- Consists of sorted data parts (or just parts)
- Each part contains range of blocks
- Insert query create part with single block
- Merges merge parts into bigger sorted parts
- Parts are immutable (no changes since creation)
- After merge source parts become outdated
- Outdated parts removed asynchronously

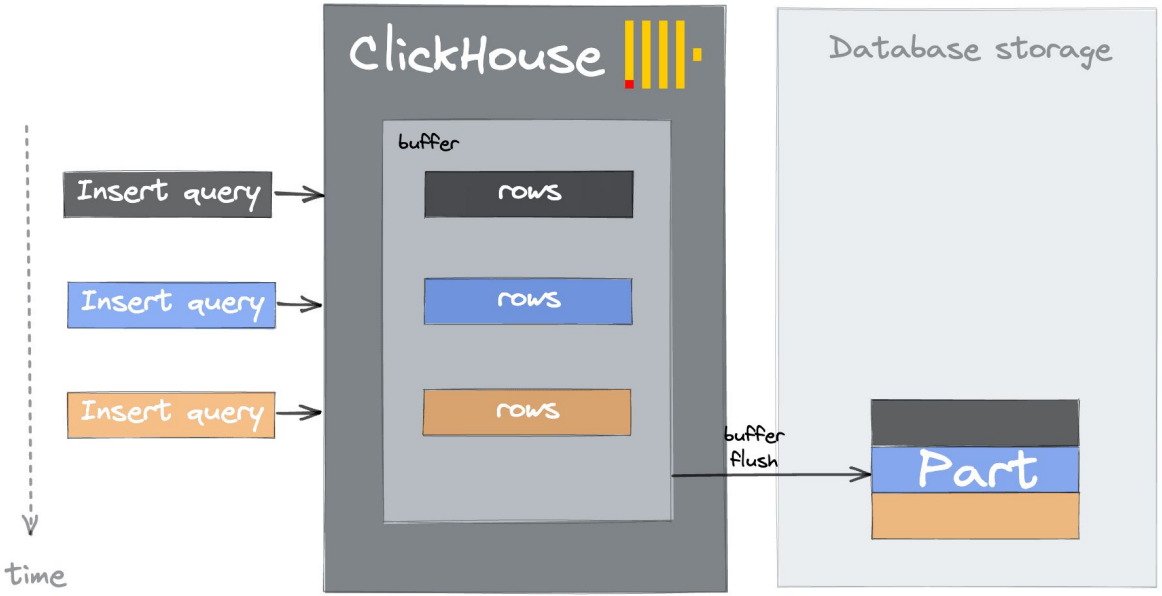


1. Too many parts - Why?



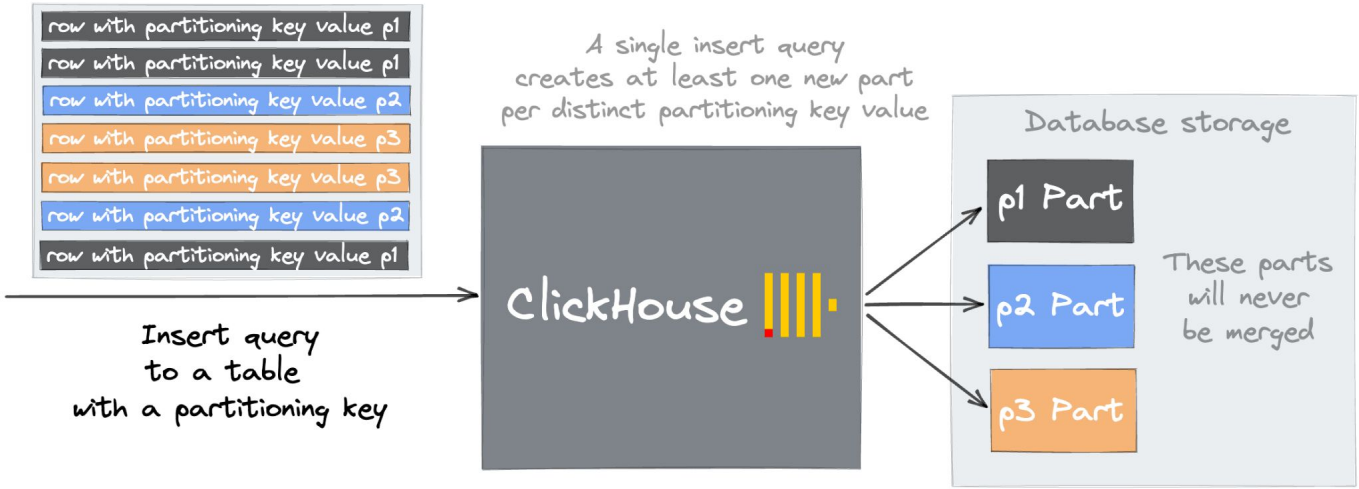
1. Many small inserts - each insert creates a new data part
2. Poorly chosen partitioning key - parts do not merge across partition
3. Excessive materialized views (MV) - each MV creates a new part

1. Too many parts - Many Small Inserts



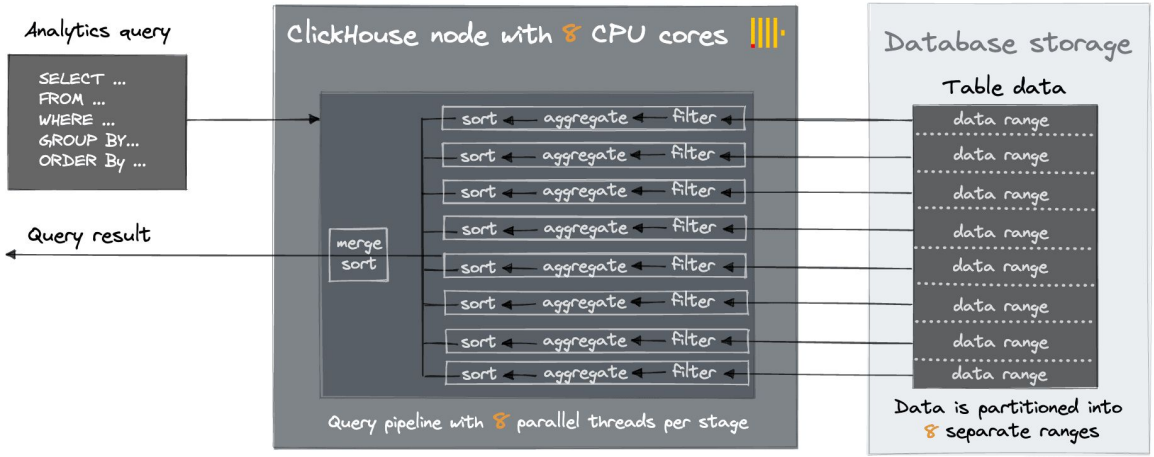
- Each insert block creates a part (be it for 1 row or 10,000 rows)
- Recommend to have at least 1,000 rows per insert
- Consider buffering at client-side
- Use async_insert to buffer inserts on memory before flushing regularly to disk

1. Too many parts - Poorly chosen partitioning key



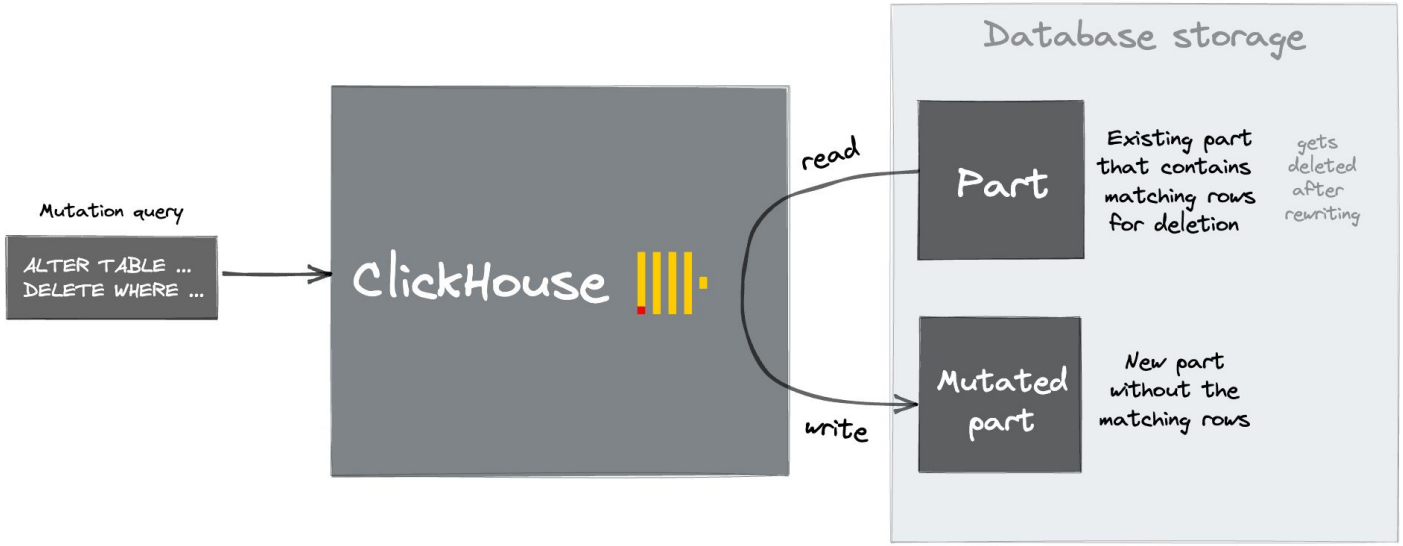
- Partition is <u>mainly used for data management</u>
  - e.g. Operations such as DROP PARTITION allow fast deletion of data subsets
- Single INSERT with rows belonging to multiple partitions will create multiple parts
- Parts belonging to different partitions are never merged

2. Going horizontal too early



- Analytical queries with sort, filter, and aggregation are parallelized independently and, by default, use as many threads as cores
- Scaling vertically first is cost efficient, easier to operate, better query performance due to the minimization of data on the network for operations such as JOINs

3. Mutation Pain



Each part is immutable. Changes a part means rewriting the whole data part

- ALTER UPDATE/DELETE queries are implemented as mutations, and are both CPU and IO-intensive and should be scheduled cautiously with permission to run limited to administrators
- Use lightweight delete or lightweight update (cloud-only) instead
- Check `system.mutations` for status, or use `KILL MUTATION` to stop mutation

4. Unnecessary use of complex types

ClickHouse Data Types

ClickHouse data types include:

- **Integer types**: signed and unsigned integers (`UInt8`, `UInt16`, `UInt32`, `UInt64`, `UInt128`, `UInt256`, `Int8`, `Int16`, `Int32`, `Int64`, `Int128`, `Int256`)
- **Floating-point numbers**: floats(`Float32` and `Float64`) and `Decimal` values
- **Boolean**: ClickHouse has a `Boolean` type
- **Strings**: `String` and `FixedString`
- **Dates**: use `Date` and `Date32` for days, and `DateTime` and `DateTime64` for instances in time
- **JSON**: the `JSON` object stores a JSON document in a single column
- **UUID**: a performant option for storing `UUID` values
- **Low cardinality types**: use an `Enum` when you have a handful of unique values, or use `LowCardinality` when you have up to 10,000 unique values of a column
- **Arrays**: any column can be defined as an `Array` of values
- **Maps**: use `Map` for storing key/value pairs
- **Aggregation function types**: use `SimpleAggregateFunction` and `AggregateFunction` for storing the intermediate status of aggregate function results
- **Nested data structures**: A `Nested` data structure is like a table inside a cell
- **Tuples**: A `Tuple` of elements, each having an individual type.
- **Nullable**: `Nullable` allows you to store a value as `NULL` when a value is "missing" (instead of the column settings its default value for the data type)
- **IP addresses**: use `IPv4` and `IPv6` to efficiently store IP addresses
- **Geo types**: for geographical data, including `Point`, `Ring`, `Polygon` and `MultiPolygon`
- **Special data types**: including `Expression`, `Set`, `Nothing` and `Interval`

JSON Object type is experimental and is undergoing improvement

Avoid using Nullable as it requires an additional Uint8 column

[1] https://clickhouse.com/docs/en/sql-reference/data-types

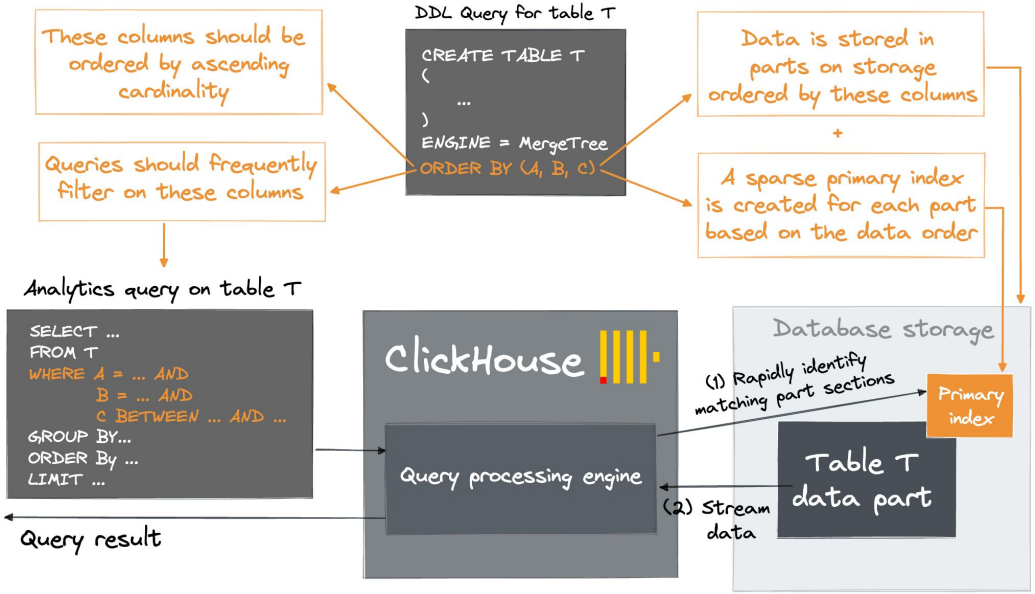
5. Deduplication at insert time

```
CREATE TABLE temp
(
   `timestamp` DateTime,
   `value` UInt64
)
ENGINE = MergeTree
ORDER BY tuple()

INSERT INTO temp VALUES ('2022-10-21', 10), ('2022-10-22', 20), ('2022-10-23', 15), ('2022-10-24', 18)
INSERT INTO temp VALUES ('2022-10-21', 10), ('2022-10-22', 20), ('2022-10-23', 15), ('2022-10-24', 18)

clickhouse-cloud :) SELECT * FROM temp

SELECT *
FROM temp

┌─────────timestamp─┬─value─┐
│ 2022-10-21 00:00:00 │    10 │
│ 2022-10-22 00:00:00 │    20 │
│ 2022-10-23 00:00:00 │    15 │
│ 2022-10-24 00:00:00 │    18 │
└─────────────────────┴───────┘
```

Same value

- Hash of an inserted block is written to Keeper (in replicated setup) and to disk (in non-replicated)
- By default, hash of the last 100 inserted blocks are compared and deduplicated
  - Configurable through `replicated_deduplication_window` / `non_replicated_deduplication_window`

6. Poor Primary Key Selection




```
CREATE TABLE test
(
    user_id UInt32,
    message String,
    day Date,
    metric Decimal(30,2)
)
ENGINE = MergeTree
PRIMARY KEY (???)
```

- Columns should be ordered by ascending cardinality
- Queries should frequently filter on these columns

[1] https://clickhouse.com/docs/en/optimize/sparse-primary-indexes

11. Memory Limit Exceeded for Query

```
DB::Exception: Memory limit (total) exceeded: would use 14.40 GiB (attempt to allocate
chunk of 4365249 bytes), maximum: 14.40 GiB ... (MEMORY_LIMIT_EXCEEDED)
```

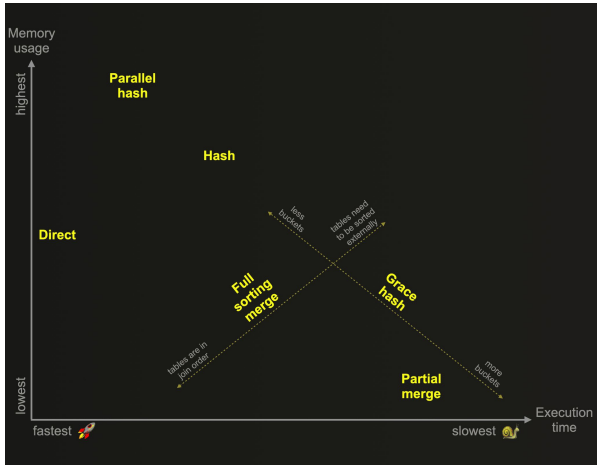
Aggregation

Aggregation and sorting can be memory-intensive

Set a maximum amount of memory any aggregation or sorting can use before "spilling" out to disk

- max_bytes_before_external_group_by
- max_bytes_before_external_sort

JOINs



Read our 5-part series blog on JOINs in ClickHouse

Rouge Queries

In an unrestricted environment, rouge queries can consume more memory than available

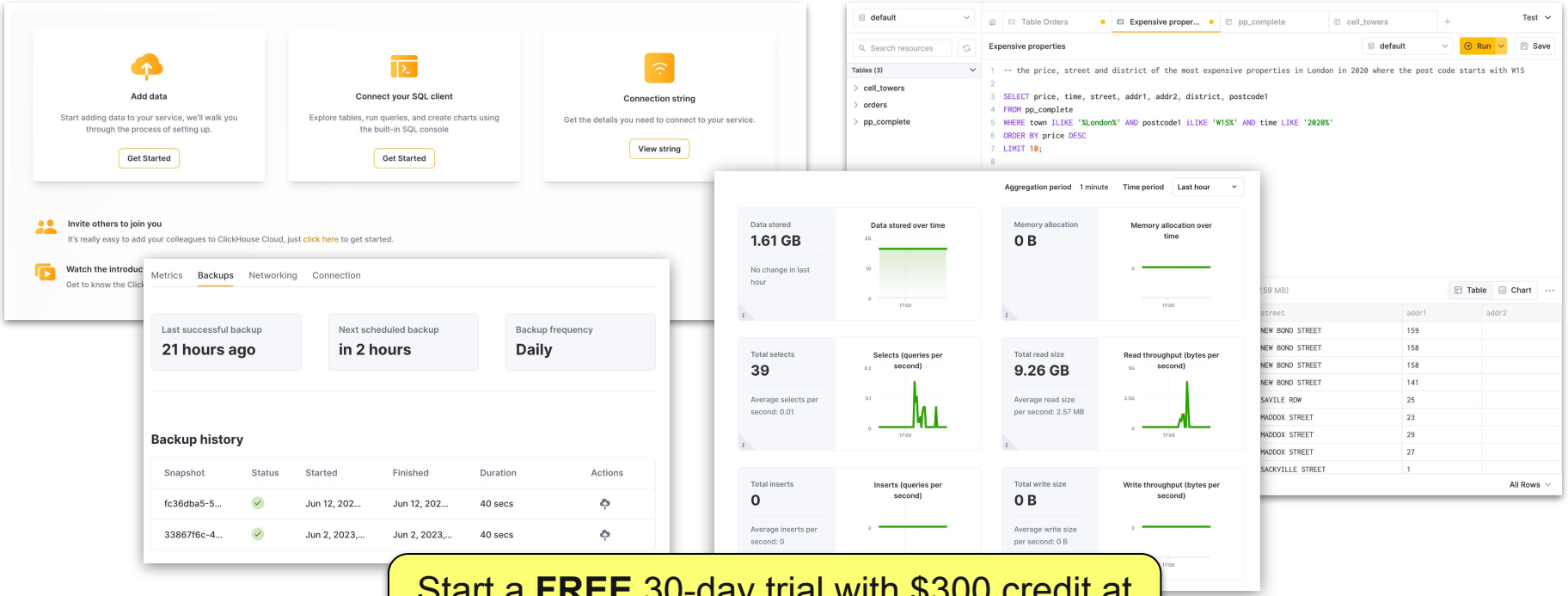
1/ Quotas - limit resource usage over a period of time

2/ Restrictions on query complexity - limit complexity of a single query

ClickHouse Cloud

aws

Soon



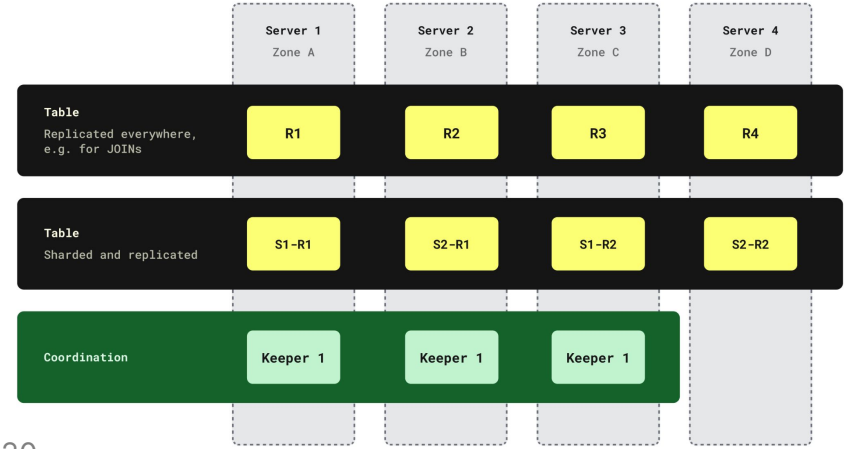
Start a **FREE** 30-day trial with $300 credit at https://clickhouse.com

ClickHouse

Self-managed

- ✓ Flexible architecture
- ✓ Feature-rich
- ✓ Easy to use
- ✓ Fast, scalable, reliable

Sample self-managed architecture

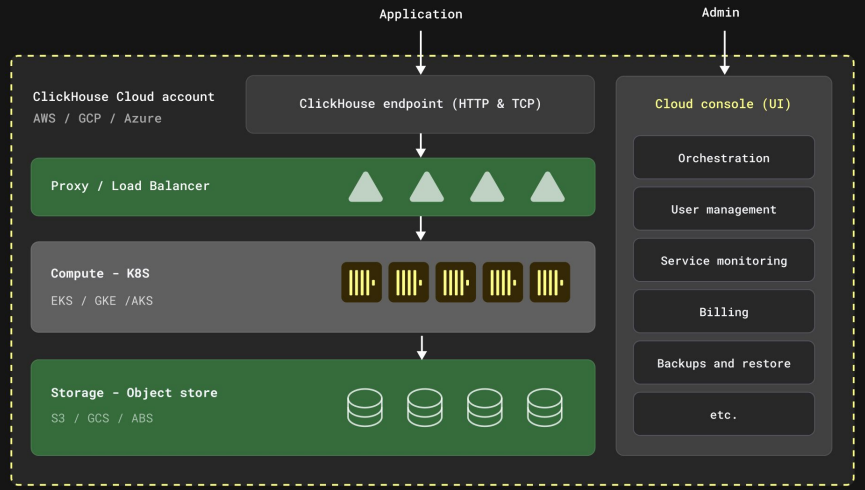


ClickHouse

Cloud

- ✓ Flexible — Managed for you
- ✓ Feature-rich — Cloud-first features & tooling
- ✓ Easy to use — Get started in minutes
- ✓ Fast — Automatically maximize performance / efficiency
- ✓ Scalable — Scale seamlessly
- ✓ Reliable — Ensure reliability

ClickHouse Cloud architecture



Join the ClickHouse Community

120k+
community members

31k+

https://github.com/ClickHouse/ClickHouse

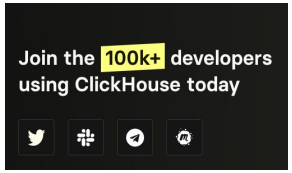
5k+
members

https://clickhouse.com/slack

32k+
followers

https://www.linkedin.com/company/clickhouseinc

5k
members



6k+
subscribers

https://www.meetup.com/pro/clickhouse/

https://www.youtube.com/c/ClickHouseDB

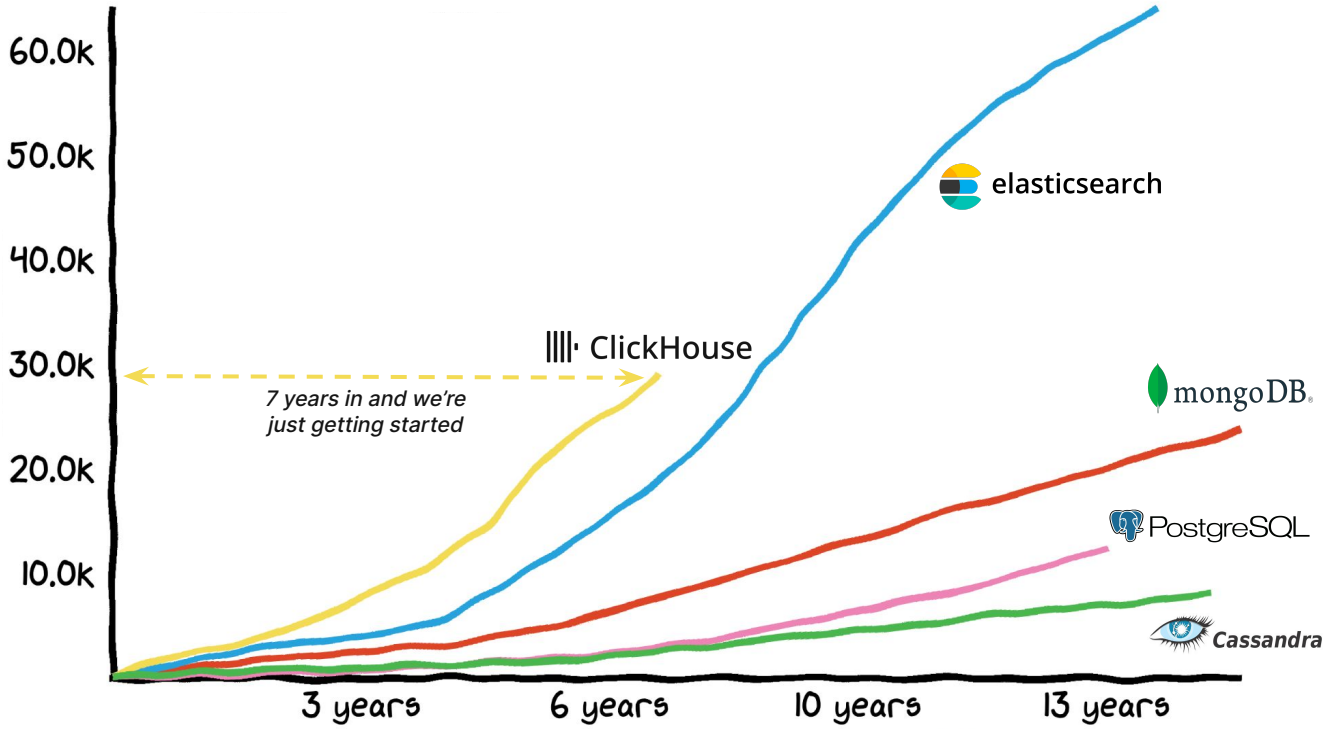
+ many more
https://github.com/ClickHouse/ClickHouse#useful-links

How can we help?

Let us know @ *sales@clickhouse.com*

ClickHouse

Growth in the community



ClickHouse OpenSource

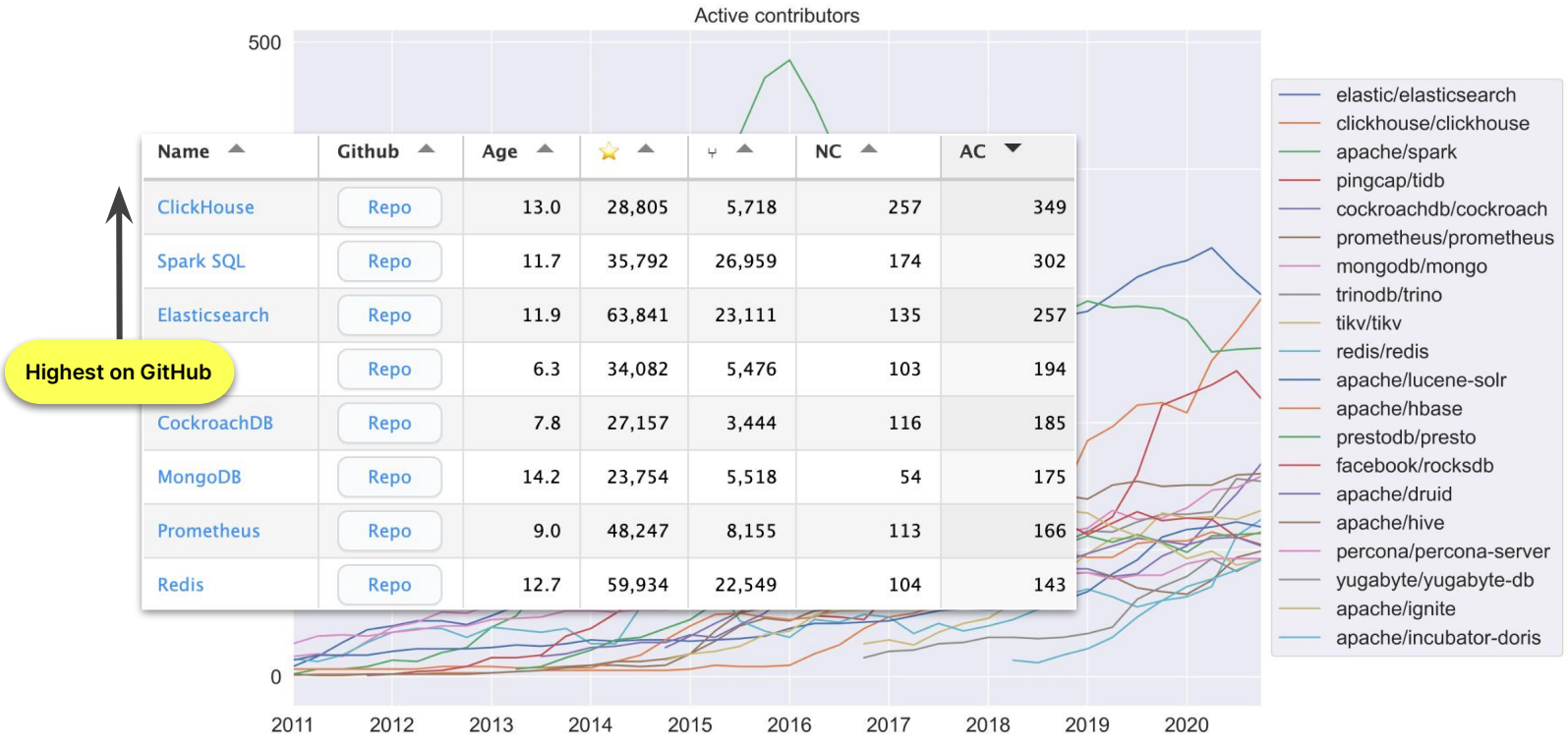
- 30.5k Stars
- 6.3k Forks
- 1.3k Contributors
- 349 Active con
- 4.4k Slack mem

Highest on GitHub

ClickHouse Cloud

- Processed hundreds of millions of queries
- Thousands of cloud trial and paid customers since Dec 2022

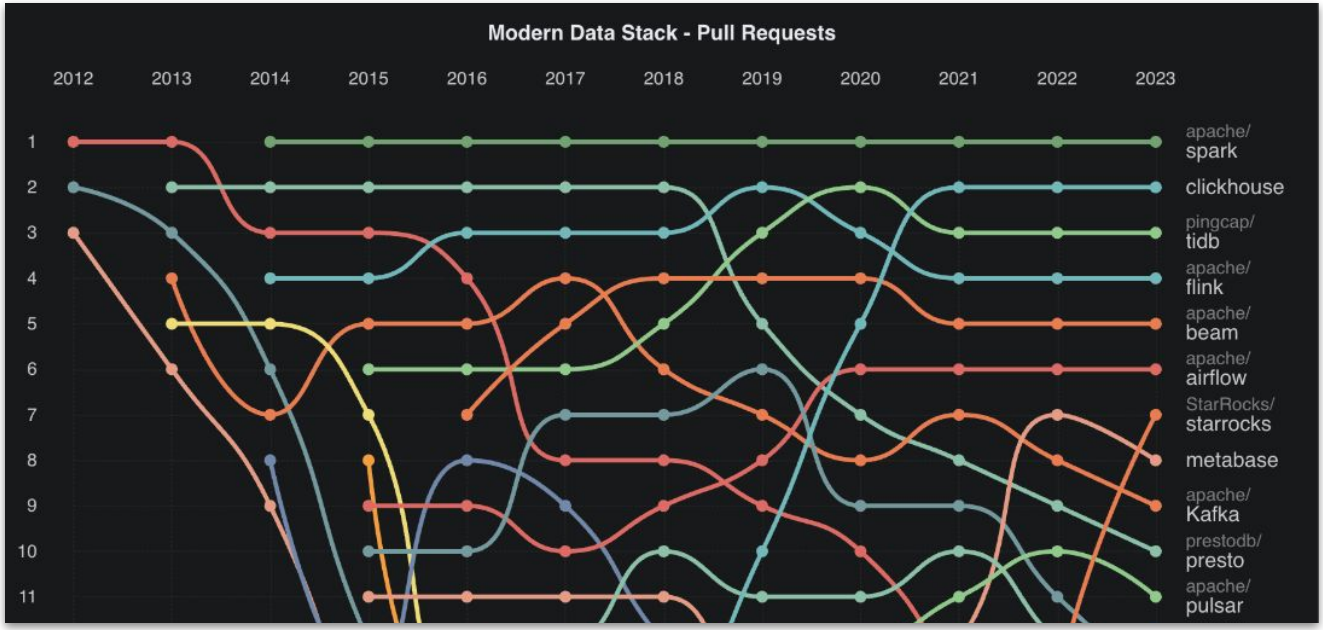
Growth in the community



Name	Github	Age	⭐	⑂	NC	AC
ClickHouse	Repo	13.0	28,805	5,718	257	349
Spark SQL	Repo	11.7	35,792	26,959	174	302
Elasticsearch	Repo	11.9	63,841	23,111	135	257
	Repo	6.3	34,082	5,476	103	194
CockroachDB	Repo	7.8	27,157	3,444	116	185
MongoDB	Repo	14.2	23,754	5,518	54	175
Prometheus	Repo	9.0	48,247	8,155	113	166
Redis	Repo	12.7	59,934	22,549	104	143

Highest on GitHub

1,300 contributors, 100,000+ commits



Top most active repo on GitHub

What are our common use cases?

Real-time Dashboards

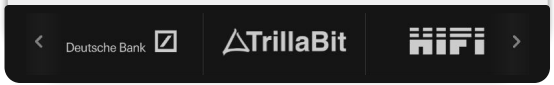
Powering interactive applications that analyze and aggregate large amounts of data on the fly. Often, these are customer facing.

Real-time Analytics

Managing continuous streams of events where real-time analysis is key.

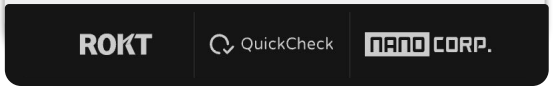
Business Intelligence

Interactively slicing and dicing data for analysis, reporting, and building internal applications.



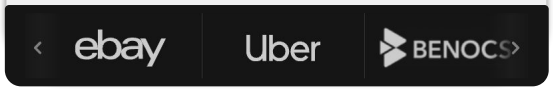
Data Warehouse Speed Layer

Offloading analytical workloads from a Data Warehouse to maximize resource efficiency and optimize for blazing speed.



Logging & Metrics

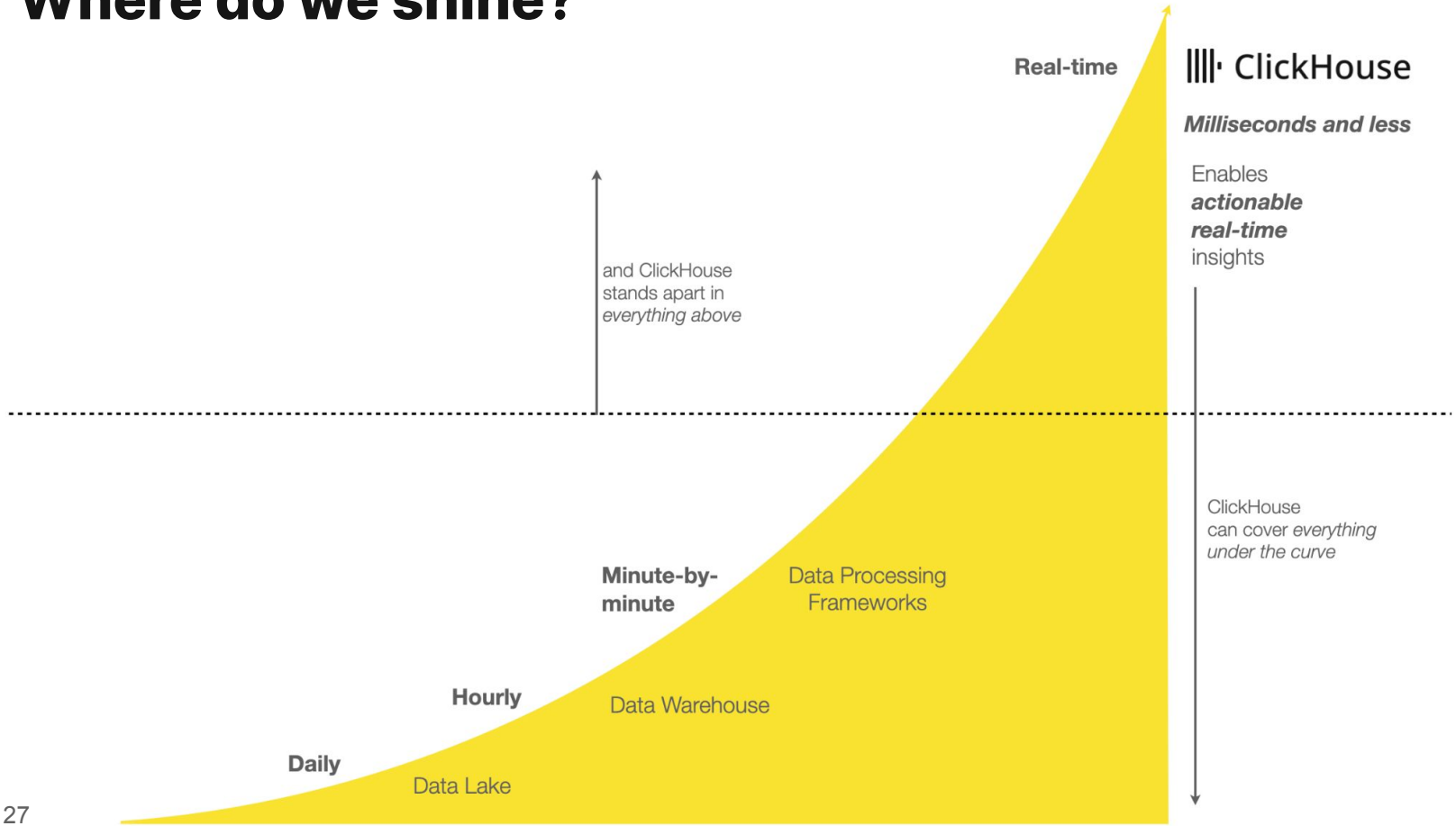
Used to monitor logs, metrics, and traces, and to detect anomalies, fraud, network or infrastructure issues, and more.



ML & Data Science

Leveraged to train models over massive data sets, and for fast and efficient vector queries.

Where do we shine?

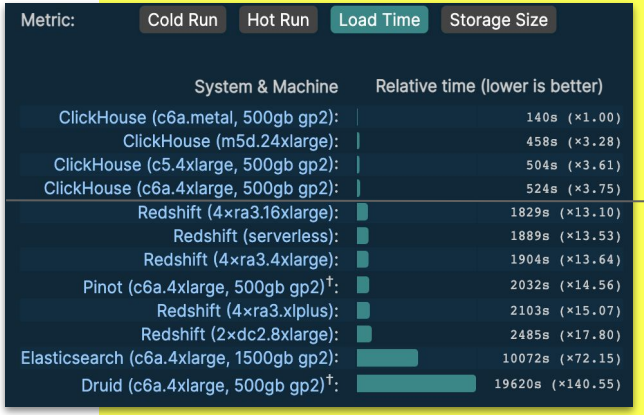


The ClickHouse Difference

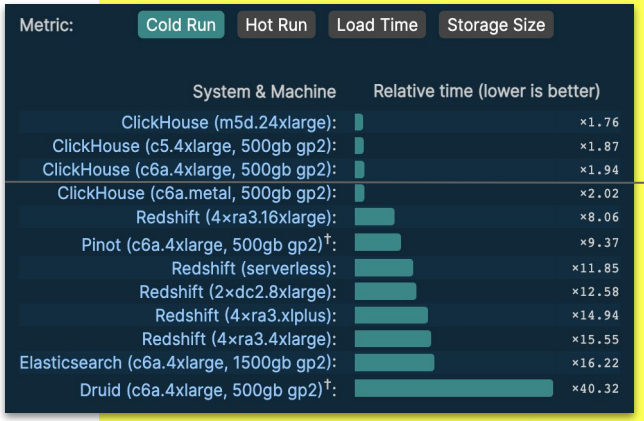
1

Very fast OLAP queries

Analytical queries - aggregations over large sets of data



37x faster loading data



20x faster querying data

benchmark.clickhouse.com

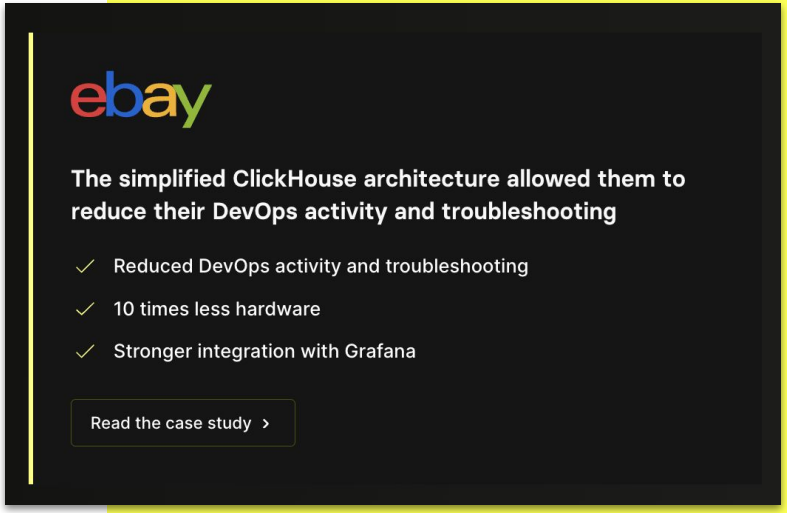
The ClickHouse Difference

1 Very fast OLAP queries

Analytical queries - aggregations over large sets of data

2 Highly resource efficient

Disruptive data compression - 10-100x storage efficiency over alternatives



The ClickHouse Difference

1 Very fast OLAP queries

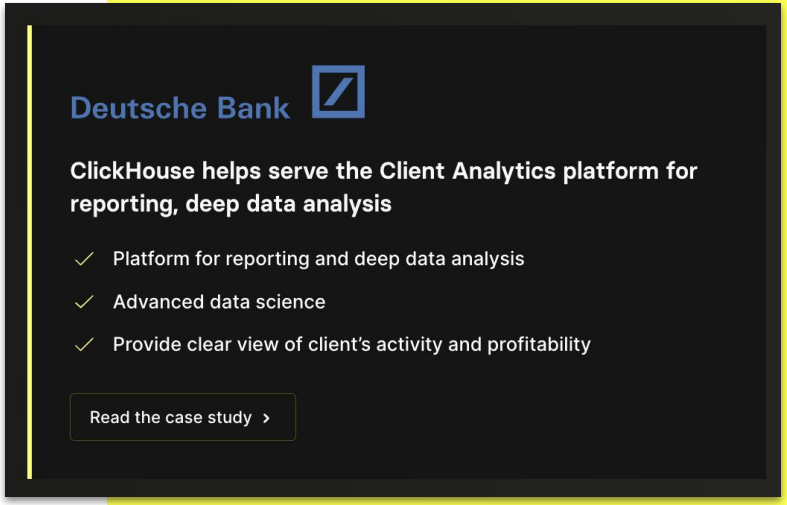
Analytical queries - aggregations over large sets of data

2 Highly resource efficient

Disruptive data compression - 10-100x storage efficiency over alternatives

3 Easy to use

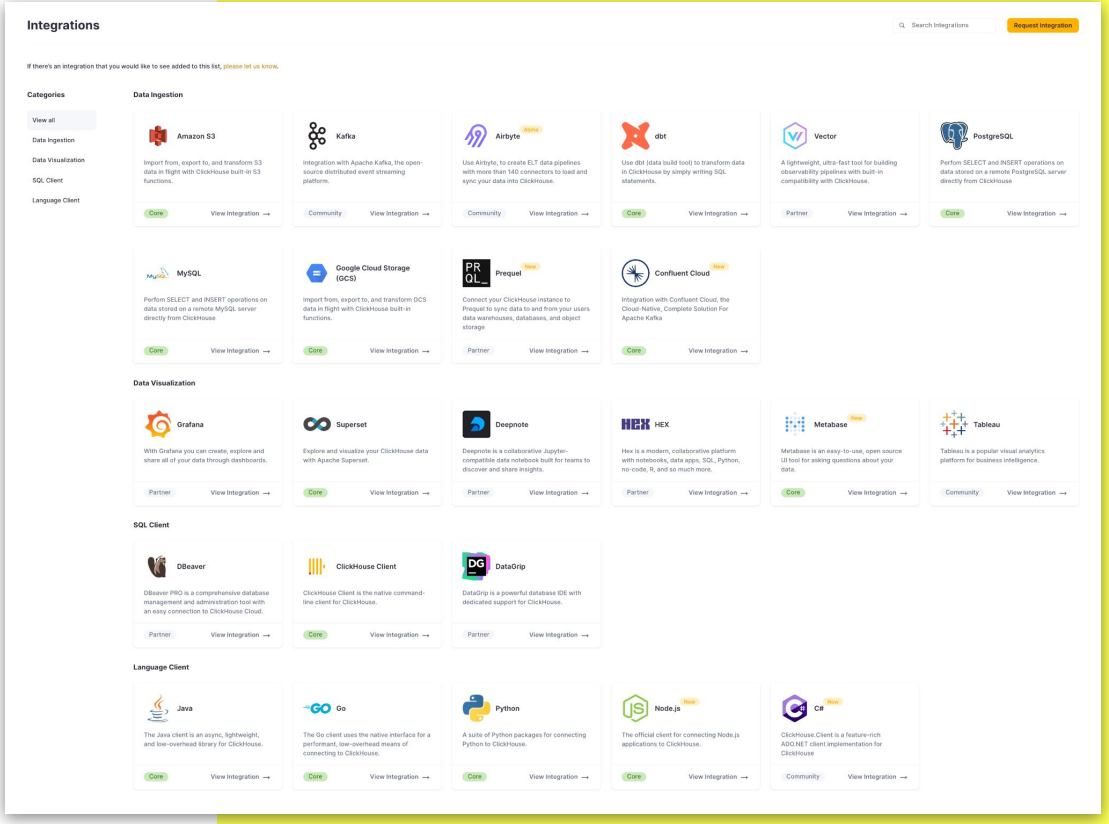
Self-service data onboarding from many sources (S3, Delta Lake, Iceberg, Hudi), analyst-friendly standard SQL syntax.



ClickHouse

Integrations made easy

- Leverage an ever-growing, curated list of core, partner, and community integrations vetted to work with ClickHouse Cloud
- Add data sources, data visualization, and other ecosystem tools with a few clicks
- Enterprise grade security enforced on all integrations under our control

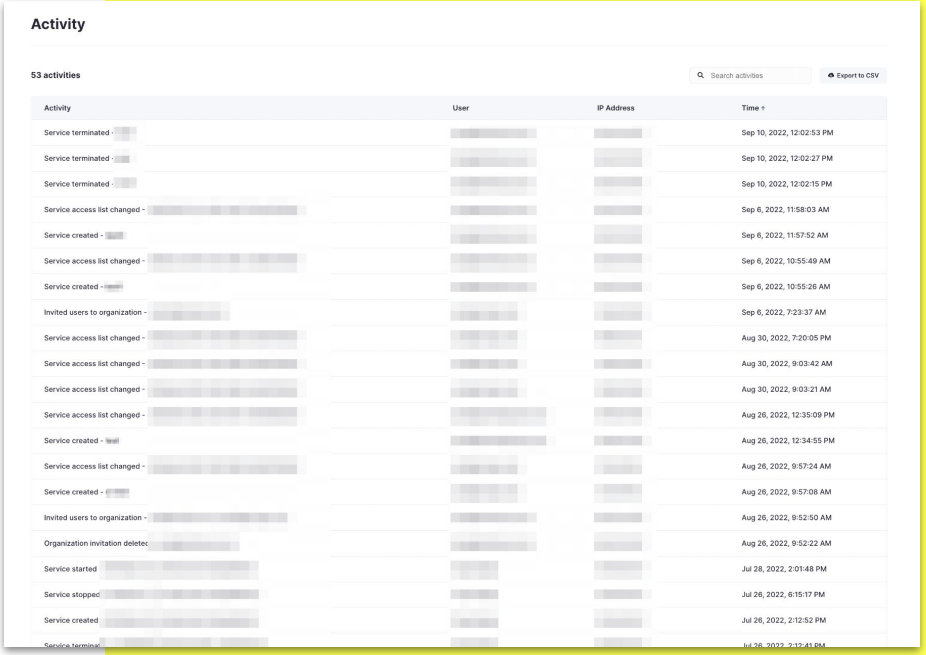


ClickHouse

Security first

- Dedicated InfoSec Team
- Cloud platform native security tools used to monitor for vulnerabilities, misconfigurations, threats
- Code and 3rd party vulnerability detection tools run continuously. Active Bug Bounty program
- Strong network access control and encryption at rest and in transit

***GDPR** and **SOC 2 Type II** Compliant*

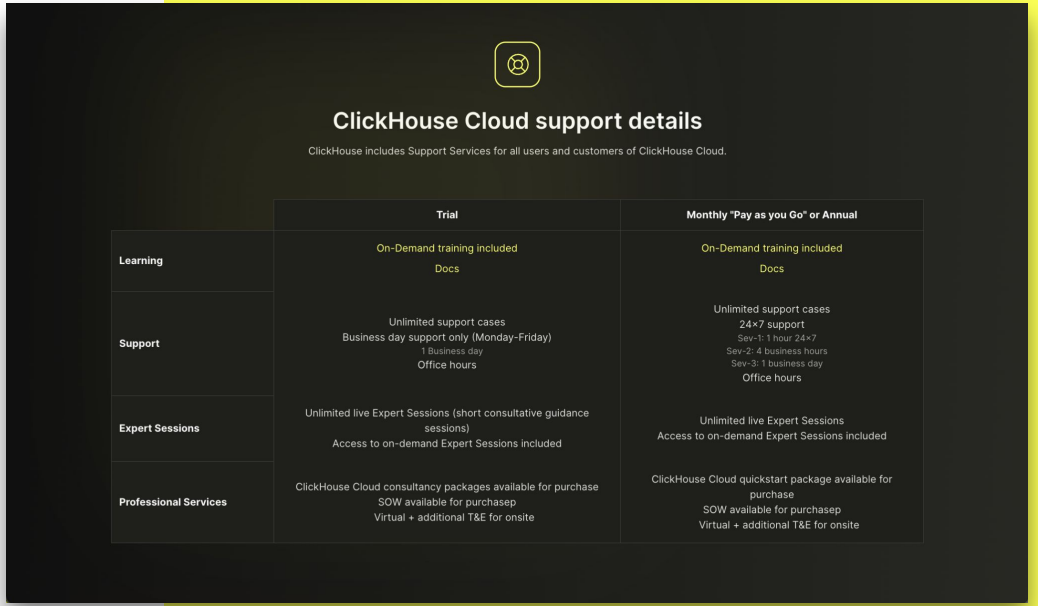


ClickHouse Cloud supports many security features, such as IP filtering, AWS Private Link, user activity logging, federated authentication, and more

ClickHouse

World class support

- Dedicated support team
- Best-in-class academy and workshops
- Abundance of resources and community support
- Obsession with user experience and ease-of-use features
- Preview: query analyzer



ClickHouse Cloud support details

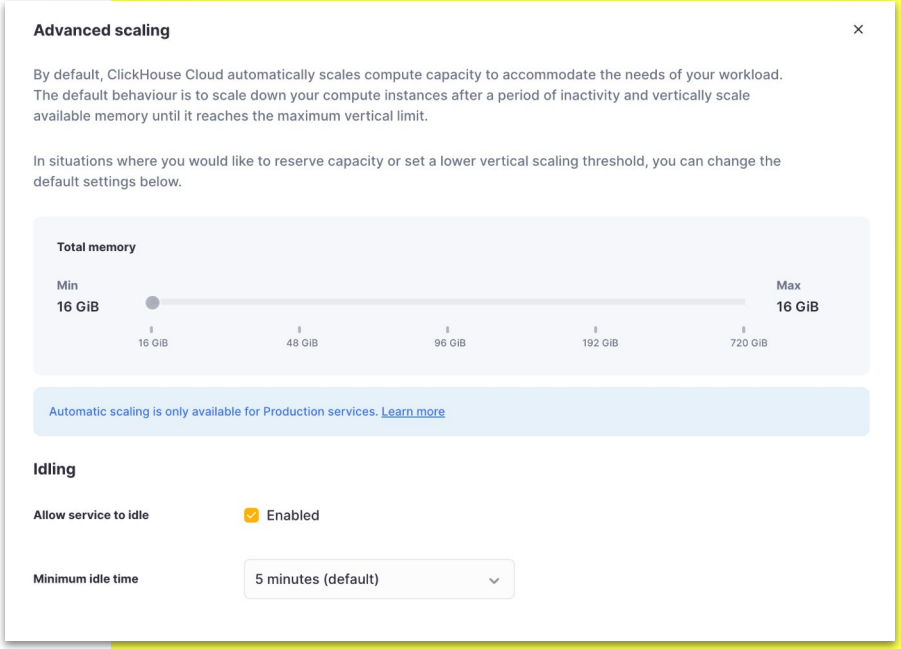
ClickHouse includes Support Services for all users and customers of ClickHouse Cloud.

	Trial	Monthly "Pay as you Go" or Annual
Learning	On-Demand training included Docs	On-Demand training included Docs
Support	Unlimited support cases Business day support only (Monday-Friday) 1 Business day Office hours	Unlimited support cases 24×7 support Sev-1: 1 hour 24×7 Sev-2: 4 business hours Sev-3: 1 business day Office hours
Expert Sessions	Unlimited live Expert Sessions (short consultative guidance sessions) Access to on-demand Expert Sessions included	Unlimited live Expert Sessions Access to on-demand Expert Sessions included
Professional Services	ClickHouse Cloud consultancy packages available for purchase SOW available for purchasep Virtual + additional T&E for onsite	ClickHouse Cloud quickstart package available for purchase SOW available for purchasep Virtual + additional T&E for onsite

ClickHouse

Advanced cost & scaling controls

- Set auto-scaling min: for always-on user-facing applications that need reserved compute capacity for peak performance.
- Set auto-scaling max: for internal application where limits on compute can prevent unexpected bills due to un-optimized ad-hoc queries.



Cloud Pricing

Development

Great for smaller workloads and starter projects

$50 - $193 / Month

- ✓ Up to 1 TB storage
- ✓ 16 GiB total memory
- ✓ Burstable CPU
- ✓ Backups every 24h, retained 1 day
- ✓ Replicated across 2 AZs
- ✓ Expert support with 24h response time

Storage	Compute
$35.33 per TB/mo	$0.2160 per unit/hr

Production

Designed to handle production workloads

Usage Based

- ✓ Unlimited storage
- ✓ 24GiB+ total memory
- ✓ Dedicated CPU
- ✓ Backups every 24h, retained 1 day
- ✓ Replicated across 3 AZs
- ✓ Expert support with 1h SLA
- ✓ AWS private link support
- ✓ Automatic scaling

Storage	Compute
$47.10 per TB/mo	$0.6888 per unit/hr

Dedicated

Designed for the most demanding latency-sensitive workloads

Capacity Based

- ✓ Unlimited storage
- ✓ Custom compute options
- ✓ Dedicated environment
- ✓ Advanced isolation and security
- ✓ Customized backup controls
- ✓ Scheduled upgrades
- ✓ Uptime SLAs
- ✓ Consultative migration guidance
- ✓ Dedicated support engineers

Contact us