ClickHouse Performance Master Class

Tools and techniques to speed up any ClickHouse app

Presenters:
Alexander Zaitsev and Mikhail Filimonov

April 23 @ 8:00 am PDT

(c) 2024 Altinity Inc.

Let’s make some introductions

Us

Database geeks with centuries of experience in DBMS and applications

You

Applications developers looking to learn about ClickHouse

Altinity

ClickHouse support and services including [Altinity.Cloud]
Authors of [Altinity Kubernetes Operator for ClickHouse]
and other open source projects

© 2024 Altinity, Inc.

What’s a ClickHouse?

© 2024 Al

ClickHouse is a SQL Data Warehouse

Understands SQL

Runs on bare metal to cloud

Shared nothing architecture

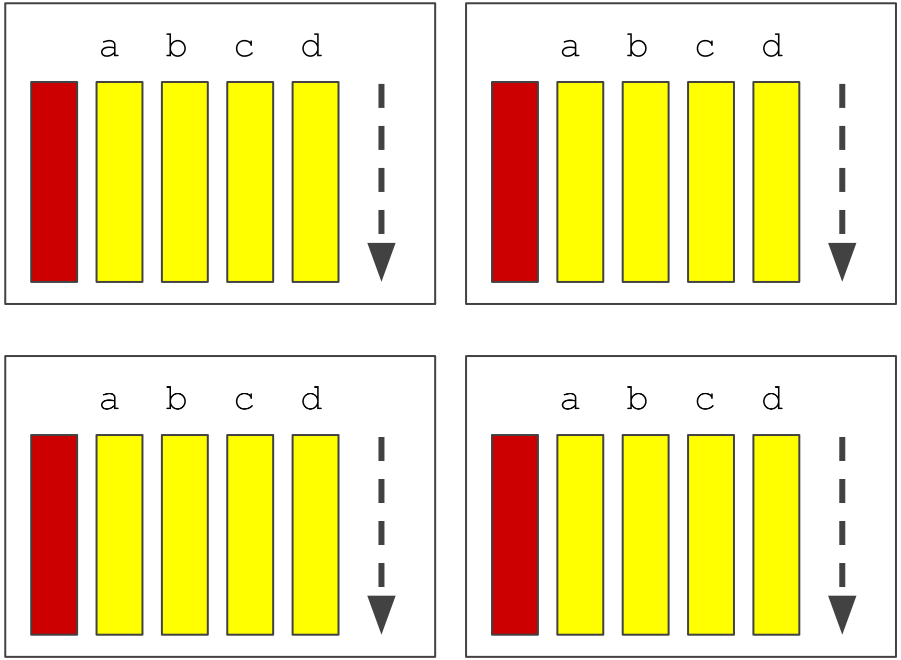
Stores data in columns

Parallel and vectorized execution

Scales to many petabytes

Is Open source (Apache 2.0)

And it's <u>really</u> fast!



© 2024 Altinity, Inc.

Performance in ClickHouse

© 2024 Al

ClickHouse is Very Fast

.. but sometimes it may go slow

© 2024 Altinity, Inc.

What does “slow” mean may be different

Execution time of a single query?

Execution time of multiple concurrent queries?

Single node or a cluster?

Data latency?

Maximum time? Median? Percentile?

© 2024 Altinity, Inc.

Bottlenecks may be different too

I/O?

CPU?

RAM?

Network?

Background operations?

ZooKeeper?

© 2024 Altinity, Inc.

Single Query Optimization

© 2024 Al

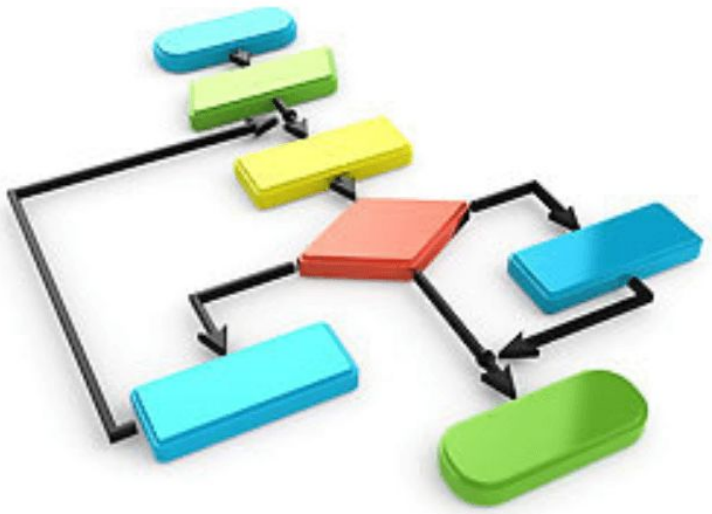
Plan of Attack

Find the slow query

Check if it is slow by itself or because of other workloads

Find the reason it is slow

Optimize



© 2024 Altinity, Inc.

Our tools

benchmarks

query_log

ProfileEvents

metric_log, asynchronous_metric_log

EXPLAIN …

clickhouse logs, set log_level='trace'

trace_log

© 2024 Altinity, Inc.

Do benchmarks!

"But on staging it used to work fast…"

- Do you have the same amount of data on staging?
- Are you sure it's slow on every run on production?
- What are other queries running? Also merges / mutations / backups etc.

clickhouse-benchmark is your friend!

© 2024 Altinity, Inc.

Benchmarks: what can you look at?

Basic stats (execution speed, memory, bytes read etc)

ProfileEvents in query_log (you can also see them in clickhouse-client)

```
$ clickhouse-client --print-profile-events --profile-events-delay-ms=-1
```

```
SELECT 1

Query id: d1ef9149-64ea-425d-89cb-6d8fcc17fd7e

   ┌─1─┐
1. │ 1 │
   └───┘
[chi-github-github-0-0-0.chi-github-github-0-0.demo.svc.cluster.local] 2024.04.23 13:43:47 [ 0 ] ContextLock: 9 (increment)
[chi-github-github-0-0-0.chi-github-github-0-0.demo.svc.cluster.local] 2024.04.23 13:43:47 [ 0 ] InitialQuery: 1 (increment)
[chi-github-github-0-0-0.chi-github-github-0-0.demo.svc.cluster.local] 2024.04.23 13:43:47 [ 0 ] MemoryTrackerPeakUsage: 9208 (gauge)
[chi-github-github-0-0-0.chi-github-github-0-0.demo.svc.cluster.local] 2024.04.23 13:43:47 [ 0 ] MemoryTrackerUsage: 9144 (gauge)
[chi-github-github-0-0-0.chi-github-github-0-0.demo.svc.cluster.local] 2024.04.23 13:43:47 [ 0 ] NetworkSendBytes: 61 (increment)
[chi-github-github-0-0-0.chi-github-github-0-0.demo.svc.cluster.local] 2024.04.23 13:43:47 [ 0 ] NetworkSendElapsedMicroseconds: 66 (increment)
[chi-github-github-0-0-0.chi-github-github-0-0.demo.svc.cluster.local] 2024.04.23 13:43:47 [ 0 ] OSCPUVirtualTimeMicroseconds: 111 (increment)
[chi-github-github-0-0-0.chi-github-github-0-0.demo.svc.cluster.local] 2024.04.23 13:43:47 [ 0 ] OSReadChars: 491 (increment)
[chi-github-github-0-0-0.chi-github-github-0-0.demo.svc.cluster.local] 2024.04.23 13:43:47 [ 0 ] OSWriteChars: 8 (increment)
[chi-github-github-0-0-0.chi-github-github-0-0.demo.svc.cluster.local] 2024.04.23 13:43:47 [ 0 ] QueriesWithSubqueries: 1 (increment)
[chi-github-github-0-0-0.chi-github-github-0-0.demo.svc.cluster.local] 2024.04.23 13:43:47 [ 0 ] Query: 1 (increment)
[chi-github-github-0-0-0.chi-github-github-0-0.demo.svc.cluster.local] 2024.04.23 13:43:47 [ 0 ] RWLockAcquiredReadLocks: 1 (increment)
[chi-github-github-0-0-0.chi-github-github-0-0.demo.svc.cluster.local] 2024.04.23 13:43:47 [ 0 ] RealTimeMicroseconds: 111 (increment)
[chi-github-github-0-0-0.chi-github-github-0-0.demo.svc.cluster.local] 2024.04.23 13:43:47 [ 0 ] SelectQueriesWithSubqueries: 1 (increment)
[chi-github-github-0-0-0.chi-github-github-0-0.demo.svc.cluster.local] 2024.04.23 13:43:47 [ 0 ] SelectQuery: 1 (increment)
[chi-github-github-0-0-0.chi-github-github-0-0.demo.svc.cluster.local] 2024.04.23 13:43:47 [ 0 ] SelectedBytes: 1 (increment)
[chi-github-github-0-0-0.chi-github-github-0-0.demo.svc.cluster.local] 2024.04.23 13:43:47 [ 0 ] SelectedRows: 1 (increment)
[chi-github-github-0-0-0.chi-github-github-0-0.demo.svc.cluster.local] 2024.04.23 13:43:47 [ 0 ] SystemTimeMicroseconds: 9 (increment)
[chi-github-github-0-0-0.chi-github-github-0-0.demo.svc.cluster.local] 2024.04.23 13:43:47 [ 0 ] UserTimeMicroseconds: 103 (increment)
```


© 2024 Altinity, Inc.

Benchmarks: A/B tests of the same query?

```
WITH
      query_id='8c050082-428e-4523-847a-caf29511d6ba' AS first,
      query_id='618e0c55-e21d-4630-97e7-5f82e2475c32' AS second,
      arrayConcat(mapKeys(ProfileEvents), ['query_duration_ms', 'read_rows', 'read_bytes', 'written_rows',
'written_bytes', 'result_rows', 'result_bytes', 'memory_usage', 'normalized_query_hash', 'peak_threads_usage',
'query_cache_usage']) AS metrics,
      arrayConcat(mapValues(ProfileEvents), [query_duration_ms, read_rows, read_bytes, written_rows, written_bytes,
result_rows, result_bytes, memory_usage, normalized_query_hash, peak_threads_usage, toUInt64(query_cache_usage)]) AS
metrics_values
SELECT
      metrics[i] AS metric,
      anyIf(metrics_values[i], first) AS v1,
      anyIf(metrics_values[i], second) AS v2,
      formatReadableQuantity(v1 - v2)
FROM clusterAllReplicas(default, system.query_log)
ARRAY JOIN arrayEnumerate(metrics) AS i
WHERE (first OR second) AND (type = 2)
GROUP BY metric
HAVING v1 != v2
ORDER BY
      (v2 - v1) / (v1 + v2) DESC,
      v2 DESC,
      metric ASC
```

Altinity gratefully acknowledges this nice example code developed by Alexey Milovidov
© 2024 ClickHouse Inc.

© 2024 Altinity, Inc.

Benchmarks: A/B tests of the same query?

	metric	v1	v2	formatReadableQuantity(minus(v1, v2))
1.	MarkCacheHits	2704	3054	-350.00
2.	WaitMarksLoadMicroseconds	31395123	13442	31.38 million
3.	DiskS3GetObject	188	0	188.00
4.	DiskS3ReadMicroseconds	16685167	0	16.69 million
5.	DiskS3ReadRequestsCount	188	0	188.00
6.	LoadedMarksCount	1719631	0	1.72 million
7.	LoadedMarksMemoryBytes	2975448	0	2.98 million
8.	MarkCacheMisses	350	0	350.00
9.	ReadBufferFromS3Bytes	271336302	0	271.34 million
10.	ReadBufferFromS3InitMicroseconds	16966980	0	16.97 million
11.	ReadBufferFromS3Microseconds	23233740	0	23.23 million
12.	S3GetObject	188	0	188.00
13.	S3ReadMicroseconds	16685167	0	16.69 million
14.	S3ReadRequestsCount	188	0	188.00

© 2024 Altinity, Inc.

Benchmarks: What changed / what was the impact?

You can easily compare the ‘before’ and ‘after’ query by query…

https://kb.altinity.com/altinity-kb-useful-queries/compare_query_log_for_2_intervals/

© 2024 Altinity, Inc.

Finding the slow query

CPU usage	OSCPUVirtualTimeMicroseconds / UserTimeMicroseconds
Disk throughput	read_bytes / written_bytes / DiskReadElapsedMicroseconds / DiskWriteElapsedMicroseconds
Network	NetworkReceiveBytes / NetworkSendBytes
RAM	memory_usage
Zookeeper	ZooKeeperTransactions
Load Average	number of concurrent queries (count & CurrentMetric_Query) & threads (peak_threads_usage & CurrentMetric_GlobalThreadActive)

© 2024 Altinity, Inc.

Finding the slow query

```
SELECT
    normalized_query_hash,
     any(query),
    count(),
    sum(ProfileEvents['OSCPUVirtualTimeMicroseconds']) AS
OSCPUVirtualTime
FROM clusterAllReplicas('{cluster}', system.query_log)
WHERE event_time between ...
  AND type in (2,4)
GROUP BY normalized_query_hash
ORDER BY OSCPUVirtualTime DESC
LIMIT 30
FORMAT Vertical
```

Groups similar queries!

Shows one sample

Shows the top of 'metric'-intensive

More complicated example: https://kb.altinity.com/altinity-kb-useful-queries/query_log/

© 2024 Altinity, Inc.

I/O is typically the key metric for performance

"Good" Queries:

- Read "little" GB
- Read it fast: >1GB/sec

"Bad" Queries:

- Read "a lot" GBs
- Read it slow: 10s-100s MB/Sec

```
1 row in set. Elapsed: 4.002 sec. Processed 2.31 billion rows, 28.06 GB (577.66 million rows/s., 7.01 GB/s.)
Peak memory usage: 389.17 MiB.
```

```
1 row in set. Elapsed: 160.315 sec. Processed 2.31 billion rows, 868.76 GB (14.42 million rows/s., 5.42 GB/s.)
Peak memory usage: 11.58 GiB.
```

```
1 row in set. Elapsed: 289.591 sec. Processed 2.31 billion rows, 28.06 GB (7.98 million rows/s., 96.90 MB/s.)
Peak memory usage: 277.09 MiB.
```


© 2024 Altinity, Inc.

What if a query reads a lot…

Full Scan?

- EXPLAIN indexes=1
- EXPLAIN ESTIMATE
- set send_logs_level = 'debug'
- force_primary_key, force_index_by_date, force_data_skipping_indices, force_optimize_projection, force_optimize_projection_name

What about this query?

```
SELECT toString(date) as date FROM table WHERE date = '2023-01-01'
```


© 2024 Altinity, Inc.

EXPLAIN indexes = 1 SELECT ...

```
explain
Expression ((Project names + Projection))
  Aggregating
    Expression (Before GROUP BY)
      Expression
        ReadFromMergeTree (default.ontime)
        Indexes:
          MinMax
            Condition: true
            Parts: 35/35
            Granules: 24727/24727
          Partition
            Condition: true
            Parts: 35/35
            Granules: 24727/24727
          PrimaryKey
            Keys:
              FlightDate
            Condition: and((FlightDate in (-Inf, 16841]), (FlightDate in [16801, +Inf)))
            Parts: 35/35
            Granules: 540/24727
```


© 2024 Altinity, Inc.

EXPLAIN ESTIMATE ...

```
EXPLAIN ESTIMATE
SELECT
    Dest AS d,
    Name AS n,
    count(*) AS c,
    avg(ArrDelayMinutes)
FROM ontime
INNER JOIN airports ON airports.IATA = ontime.Dest
GROUP BY
    d,
    n
HAVING c > 100000
ORDER BY d DESC
LIMIT 10

Query id: 4ebd2eb3-09b7-4cc0-8fec-6f549bed4641
```

	database	table	parts	rows	marks
1.	default	airports	1	7543	1
2.	default	ontime	35	201575308	24727

© 2024 Altinity, Inc.

Fixing Full Scan

Causes:

'Missing' the WHERE condition
    WHERE non_pk_col=10

Bad ORDER BY / PRIMARY KEY

    ORDER BY (unique_id)

Complex logical expressions

Complex (non-monotonic) functions

Fixes:

Add the WHERE condition
    WHERE … AND pk_col='foo'

Fix ORDER BY / PRIMARY KEY

    ORDER BY (tenant, category, event)

Simplify expressions

Rewrite use of functions
    WHERE cityHash64(col) = cityHash64('expr')

© 2024 Altinity, Inc.

Not a full scan but still reads a lot...

Just a lot of data

Inefficient reading of columns

CTE reuse

→

Use pre-aggregations of projects

Force PREWHERE

Avoid CTE reuse, or move it to temporary table

© 2024 Altinity, Inc.

How PREWHERE works

Normal WHERE logic:

```
SELECT * FROM table
WHERE col1=...
```

PREWHERE logic:

```
SELECT * FROM table
WHERE (pk) IN (SELECT pk FROM
table WHERE col1=...)
```

name	value
optimize_move_to_prewhere	1
optimize_move_to_prewhere_if_final	0
move_all_conditions_to_prewhere	1
enable_multiple_prewhere_read_steps	1
move_primary_key_columns_to_end_of_prewhere	1
query_plan_optimize_prewhere	1
merge_tree_determine_task_size_by_prewhere_columns	1

© 2024 Altinity, Inc.

Other possible reasons for slow reads

- Slow disk
- Saturated disk (merges? mutations? backup?)
- S3 (is it needed? add cache)
- Overly aggressive compression:
  - `CODEC(Gorilla, ZSTD(16))` – excellent compression. Never do it!

© 2024 Altinity, Inc.

Reads are fast – query is slow

- Prefer simple things
- Learn ‘ClickHouse-ways’
  - Grace Hopper: "The most dangerous phrase in the language is, 'We’ve always done it this way.'"
  - There Is More Than One Way To Do It - Perl’s motto is often true for SQL
- Computations: query time vs insert time
  - MATERIALIZED columns
- Process every row & every column only once

© 2024 Altinity, Inc.

Slow expression on every row

lowerUTF8(column) = ‘foo’ => lower(column) = ‘foo’

column IN (‘foo’,’FOO’)

Or maybe just normalize (do lowercase) once at the insert time?

© 2024 Altinity, Inc.

Multiple evaluations

```
WHERE lower(logline) like '%f4079%'
   or lower(logline) like '%f00004079%'
   or lower(logline) like '%f04079%'
   or lower(logline) like '%f004079%'
   or lower(logline) like '%f0004079%'
   or lower(logline) like
'%f000004079%'
```

```
 WHERE match(logline, '[Ff]0*4079')
```

```
SELECT
 JSONExactString(json, 'a'),
 JSONExactString(json, 'b'),
 JSONExactString(json, 'c')
```

```
WITH
 JSONExtract(json, 'Tuple(a String, b
String, c String') as json_parsed
SELECT
 tupleElement(json_parsed, 'a') as a,
 tupleElement(json_parsed, 'b') as b,
 tupleElement(json_parsed, 'c') as c
```


© 2024 Altinity, Inc.

Slow aggregation / sorting

- Benchmark it: do simple A/B test without ORDER BY / GROUP BY
- When possible do computations on the aggregated data

  sum(10*col) => 10*sum(col) (in simple cases ClickHouse will do it automatically)

- Injective functions / injective dictionaries - apply them after the group by

  select dictGet(dict,'attr',col) as col_undict group by col_undict
  vs
  select dictGet(dict,'attr',col) as col group by col?

- Datatypes matters (prefer simpler)
- Some aggregate functions states can be huge & expensive

  Are you sure you need uniqExact not uniqCombined ?

- Low level: two-level aggregation, max_bytes_before_external, distributed_memory_efficient_ etc.

© 2024 Altinity, Inc.

Slow JOINs

No cost-based optimizer!

Do you need JOIN at all?

- Denormalization (= insert-time join)
- Dictionaries (~ always in RAM)

settings join_algorithm = 'direct', 'grace_hash', 'parallel_hash', 'prefer_partial_merge', 'hash', 'partial_merge', 'full_sorting_merge'

© 2024 Altinity, Inc.

Join Optimizations: GROUP BY key first

```
SELECT zone,
       sum(passenger_count)
FROM tripdata
INNER JOIN taxi_zones ON
taxi_zones.location_id =
pickup_location_id
WHERE toYear(pickup_date) = 2016
GROUP BY 1 ORDER BY 2 desc
LIMIT 10



400ms
```

```
SELECT zone,
       sum(passenger_count)
FROM
(SELECT
    pickup_location_id,
    sum(passenger_count) passenger_count
 FROM tripdata
 WHERE toYear(pickup_date) = 2016
 GROUP BY 1) t
INNER JOIN taxi_zones ON
taxi_zones.location_id =
pickup_location_id
GROUP BY 1 ORDER BY 2 desc
LIMIT 10

100ms
```

Altinity

© 2024 Altinity, Inc.

Join Optimizations: replace JOIN with IN

```
SELECT
     toYear(pickup_date),
    sum(passenger_count)
FROM tripdata
INNER JOIN taxi_zones ON
taxi_zones.location_id =
pickup_location_id
WHERE zone = 'Union Sq'
GROUP BY 1 ORDER BY 1
```

680ms

```
SELECT
     toYear(pickup_date),
    sum(passenger_count)
FROM tripdata
WHERE pickup_location_id in (SELECT
location_id from taxi_zones WHERE zone =
'Union Sq')
GROUP BY 1 ORDER BY 1
```

40ms

© 2024 Altinity, Inc.

Distributed Queries

- How Distributed get rewritten into shard query?
  - deep-most subquery!
- JOIN / IN - distributed_product_mode - be careful!
- Data locality - join on shards etc.
  - sharding key - choice can be non-obvious
  - distributed_group_by_no_merge
  - optimize_skip_unused_shards
- Check how much data do they exchange
- prefer_localhost_replica=1 (default) sometimes can create suboptimal pipelines

© 2024 Altinity, Inc.

ATTENTION in 24.3

```
allow_experimental_analyzer = 1
```

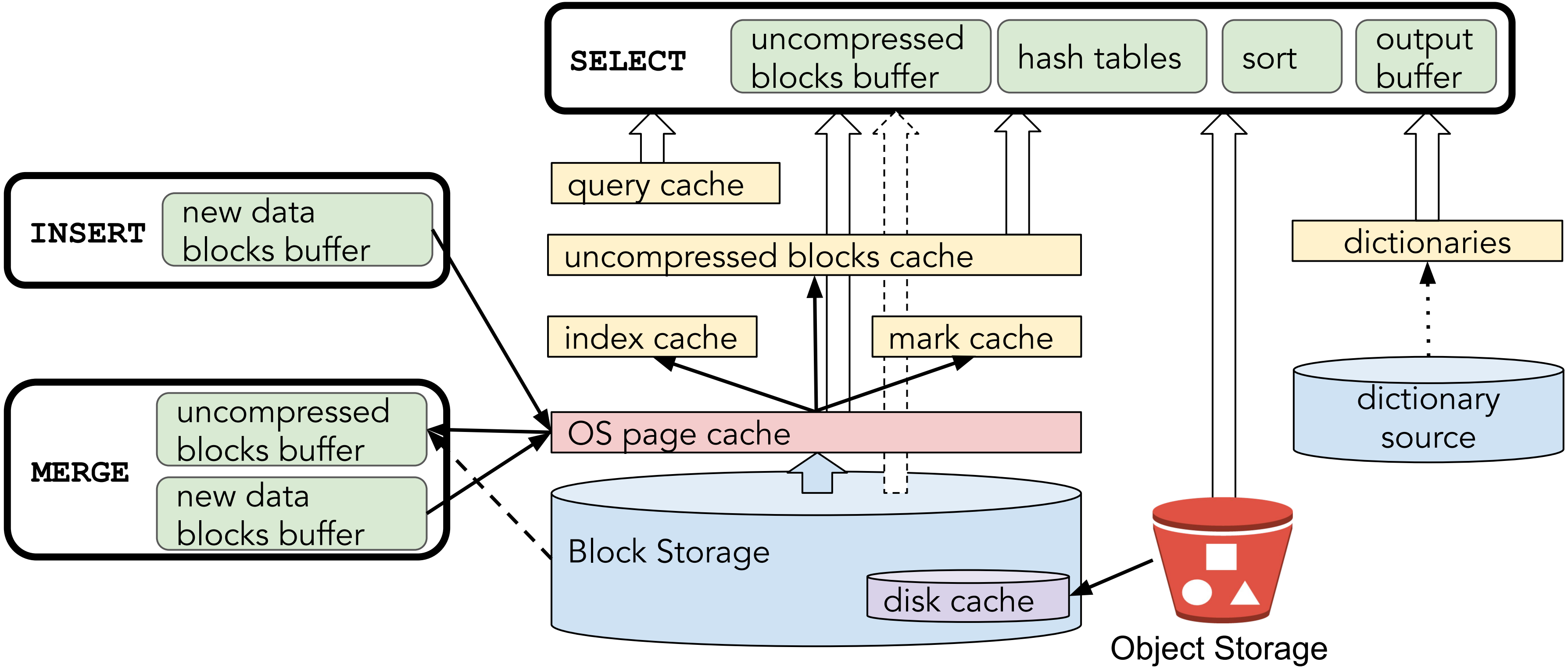
© 2024 Altinity, Inc.

RAM and Caches

RAM is your friend

© 2024 Al

What's in memory?



All about caches https://altinity.com/blog/caching-in-clickhouse-the-definitive-guide-part-1

© 2024 Altinity, Inc.

Page Cache and Disk Cache – raw data caches

With page cache – 7 seconds:

```
SELECT event_type, count()
  FROM github_events
 WHERE repo_name ilike
'ClickHouse/ClickHouse'
   AND title ilike '%cache%'
GROUP BY 1
```

```
┌─event_type────────────────────────┬─count()─┐
│ IssueCommentEvent                 │    1410 │
│ IssuesEvent                       │     307 │
│ PullRequestEvent                  │    1348 │
│ PullRequestReviewCommentEvent     │    1296 │
│ PullRequestReviewEvent            │    1498 │
└───────────────────────────────────┴─────────┘
```

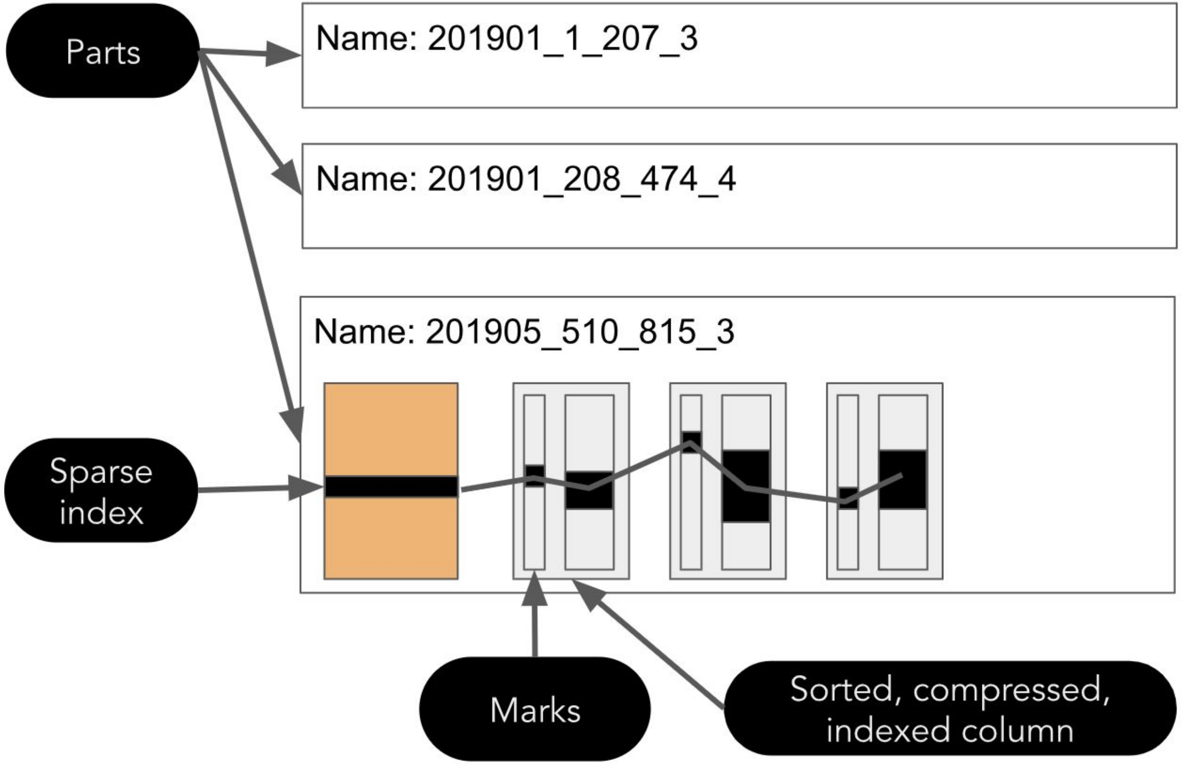
Without page cache – 20 seconds:

```
SELECT event_type, count()
  FROM github_events
 WHERE repo_name ilike
'ClickHouse/ClickHouse'
   AND title ilike '%cache%'
GROUP BY 1
SETTINGS min_bytes_to_use_direct_io=1
```

Metrics: OSReadChars - OSReadBytes = amount of data read from the page cache

© 2024 Altinity, Inc.

Mark Cache and Index Cache – query pipeline caches



Index is used to select marks – always in RAM

Marks are used to fseek data in a column – 5GB by default

```
SELECT event, value FROM
system.events WHERE event LIKE
'Mark%';
```

event	value
MarkCacheHits	5566956
MarkCacheMisses	84063

Query Cache – caches final results for repetitive queries

```
SELECT event_type, count()
  FROM github_events
 WHERE repo_name ilike
'ClickHouse/ClickHouse'
   AND title ilike '%cache%'
SETTINGS use_query_cache=1
```

First run: cache warm up
Second run: 0.001s

Server configuration:

```
<query_cache>
  <max_size_in_bytes>1073741824</max_size_in_bytes>
  <max_entries>1024</max_entries>
  <max_entry_size_in_bytes>1048576</max_entry_size_in_bytes>
  <max_entry_size_in_rows>30000000</max_entry_size_in_rows>
</query_cache>
```

Query/profile settings:

```
SELECT * from system.settings WHERE name LIKE 'query_cache%'
```

query_cache_ttl
query_cache_min_query_runs
query_cache_min_query_duration

© 2024 Altinity, Inc.

Summary: Things to keep in mind

- More memory is better. 'Unused' memory goes to page cache.
- Using swap slows ClickHouse down significantly. Disable it.
- ClickHouse process is locked in memory (`config.xml:mlock_executable`).
- Use max_server_memory_usage_to_ram_ratio to avoid OOM killer
- ClickHouse does not release memory immediately.
- ClickHouse uses the memory overcommit technique
- ClickHouse requires tuning to work in systems with low amount of memory

© 2024 Altinity, Inc.

Optimizing for Concurrency

© 2024 Al

100000 concurrent queries…

- May I increase max_concurrent_queries? Not too much.
  - High contention, numerous context switches, elevated load averages, and suboptimal performance
- High concurrency is possible if queries execute almost instantaneously
- Enabling a queue (queue_max_wait_ms) provides a buffer during peak times, helping to manage overflow and maintain system stability
- Decrease max_threads (even to 1) or use concurrent_threads_soft_limit_num
- Load balancing
  - Multiple replicas increase QPS
  - Instead of distributed queries consider intelligent balancing strategies, which will send direct queries to the specific node, instead of running cluster-wide queries.

© 2024 Altinity, Inc.

100000 concurrent queries…

Maybe you need some caching layer on the app side?

Know your load - plan the background jobs carefully

Continuously review and refine every query for performance

Have ‘plan B’ - it can be throttling, showing cached data or disabling some non-important loads, or plan the dynamic cluster rescaling

© 2024 Altinity, Inc.

Query overhead (high QPS)

simplify queries!

log_queries_probability=0..1

log level=information

© 2024 Altinity, Inc.

Wrap-up and more information

© 2024 Al

Where is the documentation?

ClickHouse official docs – https://clickhouse.com/docs/

Altinity Blog – https://altinity.com/blog/

Altinity Youtube Channel – https://www.youtube.com/channel/UCE3Y2lDKl_ZfjaCrh62onYA

Altinity Knowledge Base – https://kb.altinity.com/

Meetups, other blogs, and external resources. Use your powers of Search!

© 2024 Altinity, Inc.

Where can I get help?

Telegram - ClickHouse Channel

Slack

- ClickHouse Public Workspace - clickhousedb.slack.com
- Altinity Public Workspace - altinitydbworkspace.slack.com

Education - Altinity ClickHouse Training

Support - Altinity offers support for ClickHouse in all environments

Free Consultation - https://altinity.com/free-clickhouse-consultation/

© 2024 Altinity, Inc.

Thank you and good luck!

Altinity.Cloud

Altinity Support

Altinity Stable Builds

We're hiring!

Website: https://altinity.com
Email: info@altinity.com
Slack: altinitydbworkspace.slack.com

(c) 2024 Altinity Inc.