# Detect and terminate long-running queries

ClickHouse is a high-performance, column-oriented OLAP database, but poorly optimized or long-running queries can still impact performance especially in resource-constrained environments like Elestio. Because ClickHouse executes large queries across multiple threads and can consume high memory and disk I/O, monitoring and controlling slow or blocking operations is essential.

This guide explains how to **detect**, **analyze**, and **terminate** long-running queries using terminal tools, Docker Compose setups, and ClickHouse's internal system tables. It also outlines **prevention strategies** to help maintain system health.

# Monitoring Long-Running Queries

ClickHouse exposes query execution data through system tables like system.processes and system.query_log. These allow you to monitor currently executing and historical queries for duration, memory usage, and user activity.

## Check Active Queries via Terminal

To list currently running queries and their duration:

```
SELECT
  query_id,
  user,
  elapsed,
  memory_usage,
  query
FROM system.processes
ORDER BY elapsed DESC;
```

- elapsed is the query runtime in seconds.
- memory_usage is in bytes.
- This lets you pinpoint queries that are taking too long or consuming excessive memory.

## Monitor Query Load in Real Time

ClickHouse doesn't have a MONITOR-like command, but you can simulate real-time monitoring by repeatedly querying system.processes:

```
watch -n 2 'clickhouse-client --query="SELECT elapsed, query FROM system.processes ORDER BY elapsed DESC LIMIT 5"'
```

This updates every 2 seconds and shows the top 5 longest-running queries.

# Terminating Problematic Queries Safely

If you identify a query that is consuming too many resources or blocking critical workloads, you can terminate it by its query_id.

## Kill a Query by ID

```
KILL QUERY WHERE query_id = '<id>';
```

- The <id> can be found in the system.processes table.
- This forces termination of the query while leaving the user session intact.

To forcibly kill all long-running queries (e.g., >60 seconds):

```
KILL QUERY WHERE elapsed > 60 SYNC;
```

> Use SYNC to wait for the termination to complete before proceeding.

# Managing Inside Docker Compose

If ClickHouse is running inside Docker Compose on Elestio, follow these steps:

## Access the ClickHouse Container

```
docker-compose exec clickhouse bash
```

Then run:

```
clickhouse-client --user default
```

If authentication is enabled, add --password <your_password>.

You can now run queries like:

```
SELECT query_id, elapsed, query FROM system.processes;
```

Or terminate:

```
KILL QUERY WHERE query_id = '<id>';
```

# Analyzing Query History

ClickHouse logs completed queries (including failures) in the system.query_log table.

### View Historical Long-Running Queries

```
SELECT
  query_start_time,
  query_duration_ms,
  user,
  query
FROM system.query_log
WHERE type = 'QueryFinish'
  AND query_duration_ms > 1000
ORDER BY query_start_time DESC
LIMIT 10;
```

This helps identify patterns or repeat offenders.

# Understanding Query Latency with Profiling Tools

ClickHouse provides advanced metrics via system.metrics, system.events, and system.asynchronous_metrics.

### Generate a Performance Snapshot

```
SELECT * FROM system.metrics WHERE value != 0 ORDER BY value DESC;
```

- Use to analyze memory pressure, merge operations, disk reads/writes, and thread usage.

To examine detailed breakdowns of CPU usage or IO latency:

```
SELECT * FROM system.events WHERE value > 0 ORDER BY value DESC;
```

# Best Practices to Prevent Long-Running Queries

Preventing long-running queries is vital for maintaining ClickHouse performance, especially under high concurrency or on shared infrastructure.

- **Avoid Full Table Scans:** Use filters on primary key or indexed columns. Avoid queries without WHERE clauses on large tables.

```
SELECT count() FROM logs WHERE date >= '2024-01-01';
```

- **Limit Result Set Sizes:** Avoid returning millions of rows to clients. Use LIMIT and paginated access.

```
SELECT * FROM logs ORDER BY timestamp DESC LIMIT 100;
```

- **Optimize Joins and Aggregations:** Use ANY INNER JOIN for faster lookups. Avoid joining two huge datasets unless one is pre-aggregated or dimensionally small.
- **Avoid High Cardinality Aggregates:** Functions like uniqExact() are CPU-intensive. Prefer approximate variants (uniq()) when precision isn’t critical.
- **Set Query Timeouts and Memory Limits:** Limit resource usage per query:

```
SET max_execution_time = 30;
SET max_memory_usage = 1000000000;
```

- **Use Partitions and Projections:** Partition large datasets by time (e.g., toYYYYMM(date)) to reduce scanned rows. Use projections for fast pre-aggregated access.

---

Revision #1
Created 2025-06-11 08:54:54 UTC
Updated 2025-06-11 09:10:26 UTC