Welcome to the ClickHouse meetup!

India Bangalore

4 May, 2024

Table of contents

Since we were last here!

We move quickly -> Last ~3 months in one slide

Analyzer on by default

- Treat tuples like columns
- Multiple ARRAY JOINs same query

ATTACH PARTITION from a remote disk e.g. web

- Copies data locally (faster than insert select)

S3 Express One Zone Support

- Handle many small files and latency dominates the query
- larger files less usefully as sufficient parallelism in the query

Nested CTEs now supported

- Allows recursive queries

Qualify Clause Operator

- Filter on the value of window functions

Join Performance Improvements

- Push predicate down to both side of join
- OUTER to INNER JOIN if filter on joined rows

1️⃣ 🌲llion Row Challenge 🏎️

ClickHouse

May, 2024

ClickHouse

It all started....

With 1 billion

- In January, Gunnar Morling created the 1 billion row challenge
- Simple CSV File - city, temp
- Compute the min, max and avg of each city
- Java challenge, but Gunnar welcomed contributions from other tools
- Initial baseline ~2 min
- Some incredible results with Java code < 2s. Great read here.
- We offered a clickhouse local solution for 20s using just SQL!

```
Flores,  Petén;25.5
Port Moresby;18.5
Accra;24.9
Abha;7.6
Chicago;-0.2
Kampala;24.0
Seoul;18.2
Reykjavík;-3.8
Lviv;17.0
Hargeisa;29.9
```

Solution ~ 20s

Main Tip: *LineAsString* **over** *CSV*

```
SELECT format('{}={}/{}/{}', city, min(temperature), round(avg(temperature), 2), max(temperature))
FROM
(
   SELECT
            substringIndex(line, ';', 1) AS city,
            substringIndex(line, ';', -1)::Decimal(8, 1) AS temperature
   FROM file('measurements.txt', LineAsString)
)
GROUP BY city
ORDER BY city ASC FORMAT CustomSeparated
SETTINGS
 format_custom_result_before_delimiter = '{',
 format_custom_result_after_delimiter = '}',
 format_custom_row_between_delimiter = ', ',
 format_custom_row_after_delimiter = '',
 format_csv_delimiter = ';'

413 rows in set. Elapsed: 19.907 sec. Processed 1.00 billion rows, 13.79 GB (50.23 million rows/s., 692.86 MB/s.)
Peak memory usage: 132.20 MiB.
```

“

But 1 billion rows is tiny for ClickHouse!

Every ClickHouse user. Ever.

Scaling up

- Dask fortunately suggested a 1 trillion row challenge:
  - 5.8 minutes run time + 8 mins to provision hardware
  - ~ $3
- Challenge is a little different, 100k parquet files of 10m rows each. 2.4TiB of data in S3.
- Same query

Can we do better?

Step 1 - Plan an approach

Initial tests

- **Cost and runtime** - Runtime is one variable in cost
- **Compute in same region as bucket** - Minimize AWS egress costs
- **Sample for testing** - 11b for tuning & demo purposes
- **s3 function** - Query the data in place (think Athena)
- **Requester-pays headers** - S3 function supports

Demo 1 - Querying in place

Step 2 - Choosing an instance type

Initial tests

- **AWS Spot Instance types** - more cost efficient and we don’t need the uptime guarantees. Fast to provision.
- **Educated guess on query profile and 1b challenge:**
  - **Simple aggregation with low cardinality station (1000s)** - Likely CPU or maybe network bound
- **ARM** - Fast and cheaper per core than Intel and c7g have performed well in our public benchmarks and tests.

ARM cost efficiency

https://instances.vantage.sh

Vantage | EC2 | RDS | ElastiCache | Redshift | OpenSearch

Optimize **Kubernetes Costs** with pod efficiency reports →

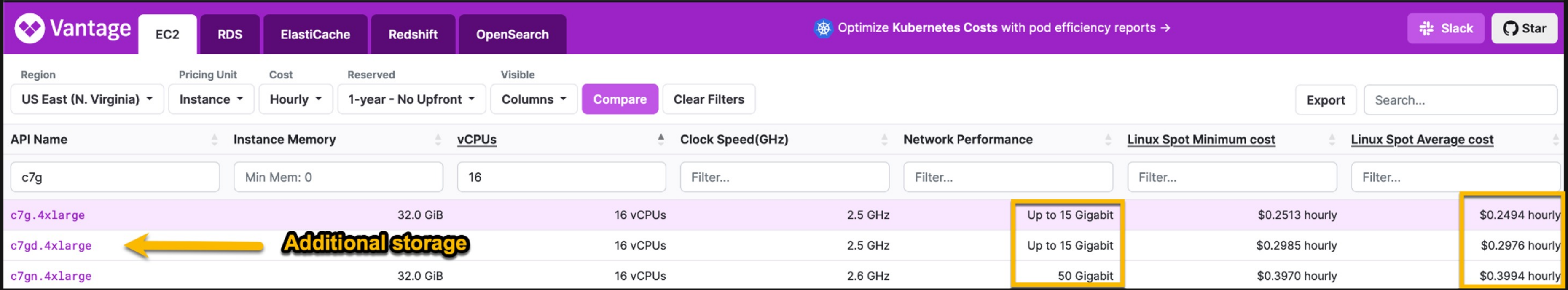
Slack | Star

Region: US East (N. Virginia) | Pricing Unit: Instance | Cost: Hourly | Reserved: 1-year - No Upfront | Visible: Columns | Compare | Clear Filters | Export | Search...

API Name	Instance Memory	vCPUs	Clock Speed(GHz)	Network Performance	Linux Spot Minimum cost	Linux Spot Average cost
c7	Min Mem: 0	Min vCPUs: 0	Filter...	Filter...	Filter...	Filter...
c7g.medium	2.0 GiB	1 vCPUs	2.5 GHz	Up to 12.5 Gigabit	$0.0172 hourly	$0.0171 hourly
c7gd.medium	2.0 GiB	1 vCPUs	2.5 GHz	Up to 12.5 Gigabit	$0.0189 hourly	$0.0191 hourly
c7a.medium	2.0 GiB	1 vCPUs	3.7 GHz	Up to 12.5 Gigabit	$0.0213 hourly	$0.0216 hourly
c7gn.medium	2.0 GiB	1 vCPUs	2.6 GHz	Up to 25 Gigabit	$0.0245 hourly	$0.0243 hourly
c7g.large	4.0 GiB	2 vCPUs	2.5 GHz	Up to 12.5 Gigabit	$0.0328 hourly	$0.0326 hourly
c7gd.large	4.0 GiB	2 vCPUs	2.5 GHz	Up to 12.5 Gigabit	$0.0368 hourly	$0.0372 hourly
c7i.large	4.0 GiB	2 vCPUs	3.2 GHz	Up to 12.5 Gigabit	$0.0414 hourly	$0.0411 hourly
c7a.large	4.0 GiB	2 vCPUs	3.7 GHz	Up to 12.5 Gigabit	$0.0442 hourly	$0.0441 hourly
c7gn.large	4.0 GiB	2 vCPUs	2.6 GHz	Up to 30 Gigabit	$0.0514 hourly	$0.0512 hourly
c7g.xlarge	8.0 GiB	4 vCPUs	2.5 GHz	Up to 12.5 Gigabit	$0.0701 hourly	$0.0696 hourly
c7gd.xlarge	8.0 GiB	4 vCPUs	2.5 GHz	Up to 12.5 Gigabit	$0.0739 hourly	$0.0744 hourly
c7i.xlarge	8.0 GiB	4 vCPUs	3.2 GHz	Up to 12.5 Gigabit	$0.0811 hourly	$0.0803 hourly
c7gn.xlarge	8.0 GiB	4 vCPUs	2.6 GHz	Up to 40 Gigabit	$0.0978 hourly	$0.0973 hourly
c7a.xlarge	8.0 GiB	4 vCPUs	3.7 GHz	Up to 12.5 Gigabit	$0.1024 hourly	$0.1026 hourly
c7g.2xlarge	16.0 GiB	8 vCPUs	2.5 GHz	Up to 15 Gigabit	$0.1198 hourly	$0.1189 hourly
c7gd.2xlarge	16.0 GiB	8 vCPUs	2.5 GHz	Up to 15 Gigabit	$0.1480 hourly	$0.1488 hourly
c7i.2xlarge	16.0 GiB	8 vCPUs	3.2 GHz	Up to 12.5 Gigabit	$0.1608 hourly	$0.1606 hourly
c7gn.2xlarge	16.0 GiB	8 vCPUs	2.6 GHz	Up to 50 Gigabit	$0.1915 hourly	$0.1897 hourly
c7a.2xlarge	16.0 GiB	8 vCPUs	3.7 GHz	Up to 12.5 Gigabit	$0.2089 hourly	$0.2094 hourly

AWS Azure | Updated 2024-03-29 00:26:33 UTC | EC2Instances.info - Easy Amazon **EC2** Instance Comparison | Docs | By Vantage | Email | Get API Key

c7g vs c7gd vs c7gn



API Name	Instance Memory	vCPUs	Clock Speed(GHz)	Network Performance	Linux Spot Minimum cost	Linux Spot Average cost
c7g	Min Mem: 0	16	Filter...	Filter...	Filter...	Filter...
c7g.4xlarge	32.0 GiB	16 vCPUs	2.5 GHz	Up to 15 Gigabit	$0.2513 hourly	$0.2494 hourly
c7gd.4xlarge		16 vCPUs	2.5 GHz	Up to 15 Gigabit	$0.2985 hourly	$0.2976 hourly
c7gn.4xlarge	32.0 GiB	16 vCPUs	2.6 GHz	50 Gigabit	$0.3970 hourly	$0.3994 hourly

- c7g - ARM processor, minimal storage, lowest spot price
- c7gd - additional storage. We don't need!
- c7gn - faster networking. Possibly needed for larger instance
  types where network will be a bottleneck but 1.5x spot cost

We start with c7g, if network bound consider c7gd

https://instances.vantage.sh

c7g - number of vCPUs

Vantage | EC2 | RDS | ElastiCache | Redshift | OpenSearch

Region: US East (N. Virginia) · Pricing Unit: Instance · Cost: Hourly · Reserved: 1-year - No Upfront · Visible: Columns · Compare · Clear Filters · Export

API Name	Instance Memory	vCPUs	Clock Speed(GHz)	Network Performance	Linux Spot Minimum cost	Linux Spot Average cost
^c7g\..*x	Min Mem: 0	Min vCPUs: 0	Filter...	Filter...	Filter...	Filter...
c7g.xlarge	8.0 GiB	4 vCPUs	2.5 GHz	Up to 12.5 Gigabit	$0.0701 hourly	$0.0696 hourly
c7g.2xlarge	16.0 GiB	8 vCPUs	2.5 GHz	Up to 15 Gigabit	$0.1198 hourly	$0.1189 hourly
c7g.4xlarge	32.0 GiB	16 vCPUs	2.5 GHz	Up to 15 Gigabit	$0.2513 hourly	$0.2494 hourly
c7g.8xlarge	64.0 GiB	32 vCPUs	2.5 GHz	15 Gigabit	$0.5394 hourly	$0.5336 hourly
c7g.12xlarge	96.0 GiB	48 vCPUs	2.5 GHz	22.5 Gigabit	$0.7508 hourly	$0.7482 hourly
c7g.16xlarge	128.0 GiB	64 vCPUs	2.5 GHz	30 Gigabit	$0.9729 hourly	$0.9744 hourly

- c7g.4xlarge - $0.2494/16 = $0.0155875 / vCPU
- c7g.8xlarge - $0.5336/32 = $0.016675 / vCPU
- c7g.12xlarge - $0.7482/48 = $0.01558 / vCPU
- c7g.16xlarge - $0.9729/64 = $0.0152 / vCPU

larger instances ≈ more cost efficient

One crucial factor…

- Availability
- So we use c7g.12xlarge (48vCPUs, 96GiB RAM)

Step 3 - Tune the query

A simple query

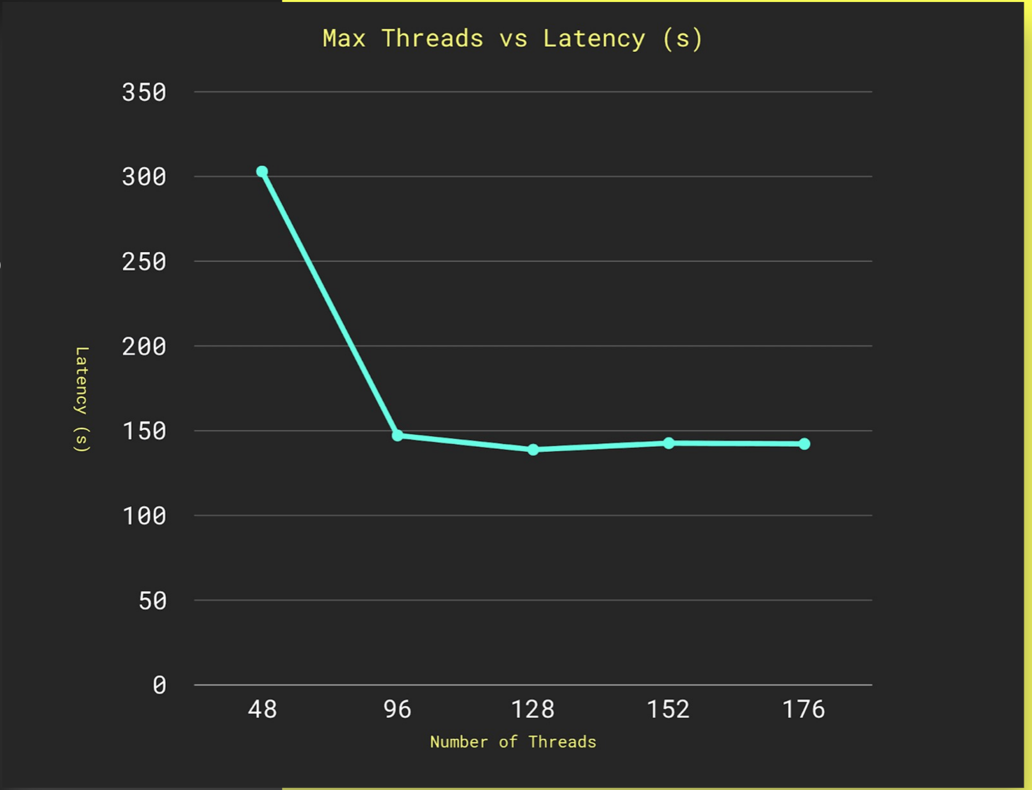
```
SELECT station, min(measure), max(measure), round(avg(measure), 2)
FROM s3('https://coiled-datasets-rp.s3.us-east-
1.amazonaws.com/1trc/measurements-10*.parquet', 'AWS_ACCESS_KEY_ID',
'AWS_SECRET_ACCESS_KEY', headers('x-amz-request-payer' =
'requester'))
GROUP BY station
ORDER BY station ASC
FORMAT `Null`
```

Format Null - we don’t need results

Demo 2 - Tuning Performance

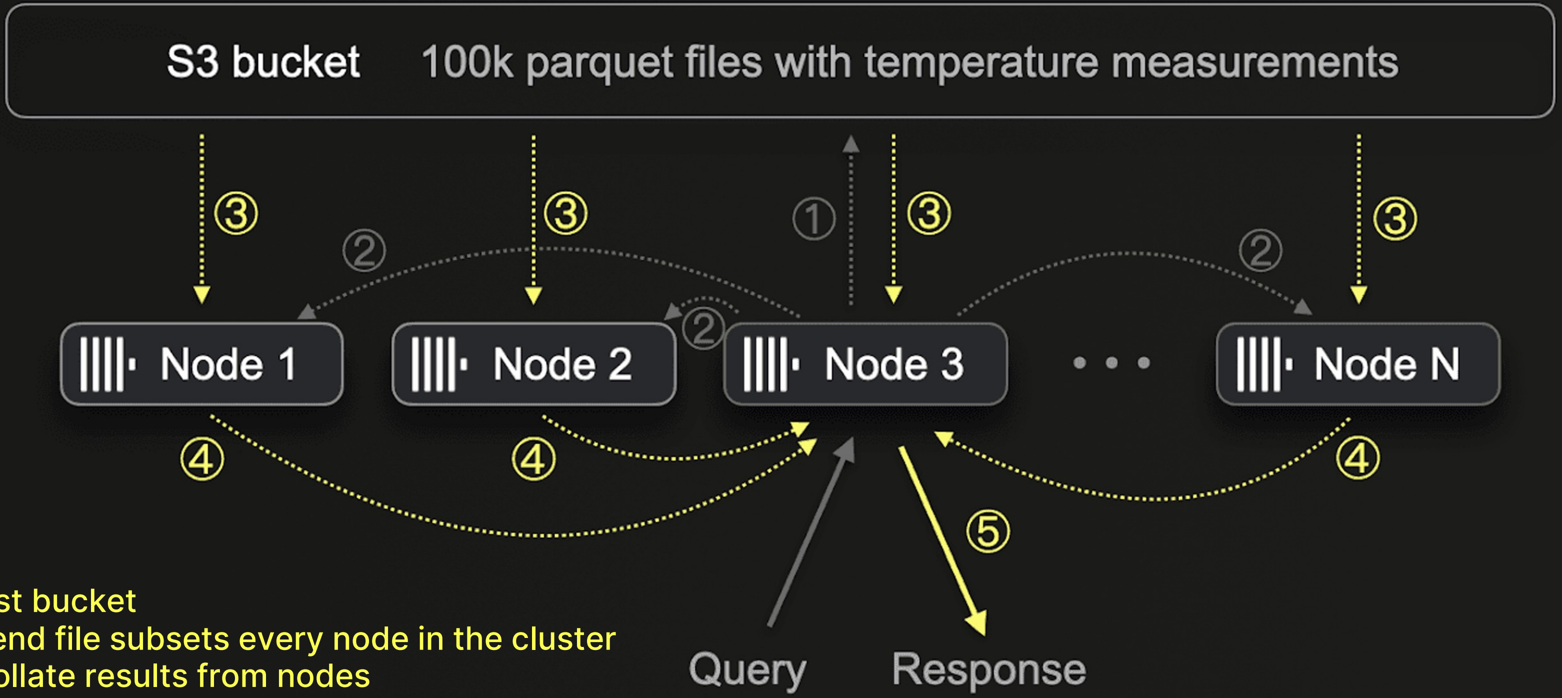
Summary of improvements

- We have reduced the S3 wait and improved throughput by 50% by increasing buffer max_download_buffer_size=52428800
- We tried turning buffer sizes - no further gain
- Increased threads to 110 to half performance again
- This tuning scales to 1 trillion



Step 4 - Moving to a cluster

S3Cluster



(1) List bucket
(2) Send file subsets every node in the cluster
(3) Collate results from nodes
(4) Merge and Sort
(5) Return Results

Final query

```
SELECT station,
        min(measure),
        max(measure),
        round(avg(measure), 2)
FROM s3Cluster('default', 'https://coiled-datasets-rp.s3.us-east-
1.amazonaws.com/1trc/measurements-*.parquet', 'AWS_ACCESS_KEY_ID',
'AWS_SECRET_ACCESS_KEY', headers('x-amz-request-payer' = 'requester'))
GROUP BY station
ORDER BY station ASC
FORMAT `Null`
SETTINGS max_download_buffer_size = 52428800, max_threads = 110
```

Step 5 - Orchestration

Orchestrating a cluster

```
config:
  aws:region: us-east-1
  1trc:aws_zone: us-east-1b
  1trc:instance_type: "c7g.12xlarge"
  1trc:number_instances: 8
  1trc:key_name: "dalem"
  1trc:cluster_password: "a_super_password"
  # AMD ami (us-east-1)
  1trc:ami: "ami-05d47d29a4c2d19e1"
  1trc:query: "SELECT station, min(measure),
max(measure),…" # full query with settings
```

- Pulumi to orchestrate a number of workers in the cluster + keeper
- Simple Python code at https://github.com/clickhouse/1trc
- Doesn’t bid on spot, just takes current price
- us-east-1b cheapest region - $0.7162 per hr

Step 6 - Final results + Costs

Running

```
(venv) ./run.sh

Diagnostics:
  pulumi:pulumi:Stack (aws-starter-dev):
    info: checking cluster is ready...
    info: cluster is ready!
    info: running query...
    info: query took 178.94s ← query time!

Resources:
    + 80 created

Duration: 5m10s ← startup time + query time!

…

Destroying (dev)

...

Duration: 2m58s ← time to remove resources
Total time: 499 seconds
```

Simple price estimate

Manual cost (estimate):

- We can compute from 499s runtime
- 8 instances at $0.7162 each
- (499/3600) * 8 * $0.7162

$0.79

But...

- Previous was estimate as not all instances are active the whole time (startup + shutdown)
- AWS Spot instance data feed provides (eventually)

$0.56

Summary

1x10^12	178s	$0.56
Rows	Query time	Cost

Final thoughts

- Query time will scale linearly with instances - eventually provisioning time will dominate
- CAVEAT: Doesn’t consider GET request calls for S3
- But if running many queries, this will be a small portion of total time
- Not a production solution - we don’t handle spot interruptions
- AMI instance would help provisioning time
- We don’t bid on Spot price
- Build your own AWS Athena?

and with MergeTree ?

$1x10^{12}$	300	16.5s
Rows	**Cores**	**Query time**

credit: Alexey Milovidov

ClickHouse

END

Thank you!

Keep in touch!

clickhouse.com/slack

#clickhouseDB @clickhouseinc

clickhouse

Icons for dark background