Boiling Data - Comparison & Performance

2023-12-18 v5 - Dan Forsberg, Ph.D, *CEO & Founder @BoilingData*

Summary

Boiling uses Open Source DuckDB OLAP SQL database engine that is at least 10x more efficient data processing engine compared to Trino/Presto/Spark and even many DWs. Together with highly scalable AWS Lambda and with Pay As You Go (PAYG) model Boiling brings very compelling interactive analytics speed BI Tool experience and ETL processing capabilities in the same package.

You pay only for the *compute when a query is running* while enjoying de facto standard data formats on your own Data Lake (Parquet on S3) with much lower storage price as well as much smaller data sizes (highly compressed data, no replication).

There is no data import needed and no vendor specific data formats. Data import (loading) happens automatically and on-demand directly into memory next to the compute (data transfer, uncompression, and data format optimisation).

Boiling implements innovative query planning, optimisation, routing, and execution layer while utilising as many Lambda functions as needed. Boiling evenly splits/groups all the needed data so that each Lambda function has suitable amount of data to lift from S3, process, and keep warm for further queries.

Boiling offers secure (row, column, and time / schedule based security) data sharing with SQL VIEWs that are part of Boiling Data Catalog. You can run SQL queries over data on S3 even without having any AWS account(s) by consuming data sets shared to You. This plays nicely e.g. with OBTs (One Big Table) models that get warmed up in Boiling and consumed by multiple data shares. Boiling Security builds on Best Current Practices (BCP), and Least Privilege principles *on top of AWS IAM* and is fully controlled by its users and helped along by Boiling.

Comparison to traditional DW and Data Lake Engines

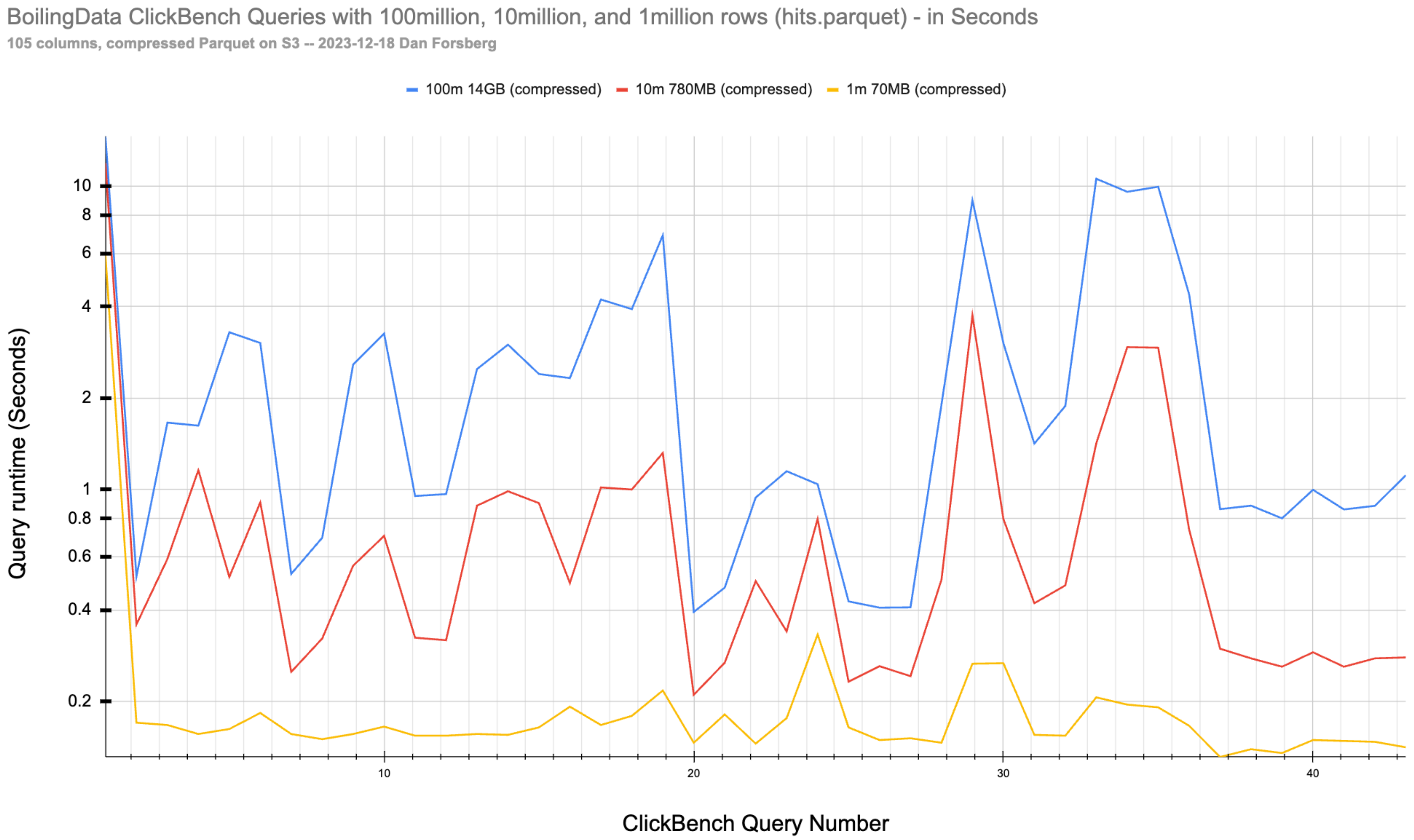
	Trad. DW	Presto / Trino / Spark	Boiling
Compute	- Cluster on some location	- Cluster on some location	- On-demand Lambda functions globally where the data is located (AWS regions)
Scaling	- Noisy neighbours - Cluster sizing (manual) - Node sizing (manual)	- Noisy neighbours - Cluster sizing (manual) - Node sizing (manual)	- Every query has dedicated Lambda functions - Rapid AWS Lambda scalability (automatic) - Number of Lambda functions allocated for a data set (automatic) - Concurrent users on the same data set will spin up more Lambda instances automatically
Storage	- Coupled with compute	- Decoupled from compute - From S3 with every query	- Decoupled from compute - Warmed up once from S3 into highly memory optimised format
Pricing	- Licenses - Compute 24/7 - Operations & Maint.	- Compute 24/7 - Operations & Maint.	- PAYG based on query execution time GBs (GB seconds)
Engine perf.	- Depends on engine, cluster, concurrent users, indexes in use, etc. - Throw money to get performance	- Slowish to very slow	- State of the art DuckDB with vectorised C++ engine (Open Source). Top of DB performance stats. - After small cold start fast; even with cold start usually faster than Presto and much faster than Spark - Shines on distributed search use case where tens of Lambda functions find the needle from the hay stack concurrently - Shines with small datasets that fit into single Lambda where all aggregations run directly against data on memory regardless of the query complexity - Supports combinable distributed queries, but also “single scan” distributed queries. - “Single scan” distributed queries require transferring data between Lambda functions. Boiling does this in compressed manner to Nx improve the performance with AWS Lambda service
Cost of BI	- Depends on amount of	- Not suitable for interactive	- Built for interactive analytics and cost efficiency with PAYG model

Tool usage	data in storage and compute cluster size	data exploration and Dashboarding. Too slow response times	
Cost of ETL usage	- Depends on amount of data in storage and compute cluster size - ETL typically needs more data and DWs replicate the data and become very expensive	- Presto is cost efficient but DuckDB is faster - Spark is very slow and costly. DuckDB is more than 10x faster data engine compared to Spark	- Cost efficient and performant to the point when Lambda compute PAYG Boiilng model become more expensive than running Spot instances ⇒ Boiling will address this “batch processing” non-interactive use case with other compute than Lambda with PAYG model as well (or with Bring Your Own Compute)

Case Study: ClickBench benchmark - 14GB Compressed clickstream data

We have a `14GB` large *compressed* Parquet file called `hits.parquet` that has `100` million rows and `105` columns (wide table) and a column with varying long URL strings (heavy). When Boiling starts to query this file, it automatically reserves 60 Lambda functions in this test run case and evenly allocates the data for all of them. User does not have to do anything but to run the queries.

The test result timings are full-roundtrip from Laptop over WLAN from Helsinki/Finland to AWS Ireland region (`eu-west-1`). The timing includes client side NodeJS SDK connection setup and calls to Boiling as well as all the query control plane handling as well as execution and results transfer back to laptop. **It thus mimics an end user perceived BI Tool session over data size that is not typical for a DW, but for an ETL pipeline** and can be described as Exploratory Data Analysis (EDA). The queries are run in serial one-by-one, but since AWS Lambda is auto scaling and elastic we could run multiple queries in parallel - like warming up 10x of the same data set.


BoilingData ClickBench Queries with 100million, 10million, and 1million rows (hits.parquet) - in Seconds
105 columns, compressed Parquet on S3 -- 2023-12-18 Dan Forsberg
100m 14GB (compressed)
10m 780MB (compressed)
1m 70MB (compressed)
Query runtime (Seconds)
10
8
6
4
2
1
0.8
0.6
0.4
0.2
10
20
30
40
ClickBench Query Number

Query performance analysis

The cold start time + 1st query runtime in this test run took about 14s. This is comparable to data loading into a database / DW. During the cold start Boiling lifts the data from S3, uncompresses it and creates in-memory DuckDB tables. We have measured that **Boiling lifts data with the speed of 5-6 GB/s (data transfer, uncompression, and data optimisation) from S3**. This means that Boiling does not create any database indexes to speed up the query performance for some subset of queries, but uses raw database engine capabilities that DuckDB offers. This way the results are also more comparable in general.

Combinable queries that are naturally parallelisable (like `sum`, `min`, `max`, `count` etc.) run very efficiently on Boiling. Similarly, queries that do "*find needles from the haystack*" kind of searches are very fast. This is because 60 Lambda functions use totally about ~240 CPU cores altogether to crunch through the data in the memory with top of tier state of the art OLAP database engine[1].

Queries that are not combinable (like `avg`) need the whole column(s) worth of data to compute the result (we don't use approximates). This basically is a single scan over the whole data set (needed columns). For this purpose *we revert back* to "normal" DW architecture where leaves (Lambda functions) do as much work as possible (filter push downs etc.) and Aggregator/Driver does the final query result computation.

With a distributed architecture like Boiling, the bottle neck in many cases is network throughput. **Boiling optimises the network throughput vs compute time to achieve best results by choosing a suitable compression algorithm for data transfers while achieving multitude of query performance improvements compared to transferring uncompressed data**. *This is driven by the AWS Lambda network bandwidth limitations*[2]. From the query results graph you can see that the spikes are queries that require high data transfers. In these tests, we have NOT used intermediate caching so that the tests are more comparable. However, intermediate caching is an obvious improvement for the tests.

Boiling production deployment allows tuning the time the data set is kept warm *after when it sits idle*. Also the number of concurrent queries (parallelism, like number of concurrent queries against the same data set at the same time) is tunable but many times not needed as queries run fast after each other fluently on AWS Lambda service. Boiling directly benefits from AWS Lambda scaling so that if there are lots of users against the same data set, AWS automatically spins up more hot Lambda instances.

[1] We believe we can still further optimise DuckDB usage in Lambda
[2] Boiling is not limited to Lambda functions only and we'll introduce alternative compute options in the future.

APPENDIX: ClickBench Queries

Note that the queries are not 100% the same as in the ClickBench repository. We have added additional sorting columns and explicit naming of the columns so that we can verify locally with DuckDB that we get exactly the same results (correctness).

Query Num	100m 14GB	10m 780MB	1m 70MB	Queries
1	14091	11017	5832	`SELECT COUNT(*) FROM parquet_scan('s3://boilingdata-demo/hits.parquet');`,
2	1213	491	188	`SELECT COUNT(*) FROM parquet_scan('s3://boilingdata-demo/hits.parquet') WHERE AdvEngineID <> 0;`,
3	2181	681	208	`SELECT SUM(AdvEngineID), COUNT(*), AVG(ResolutionWidth) FROM parquet_scan('s3://boilingdata-demo/hits.parquet');`,
4	2562	711	170	`SELECT AVG(UserID) FROM parquet_scan('s3://boilingdata-demo/hits.parquet');`,
5	2799	727	186	`SELECT COUNT(DISTINCT UserID) FROM parquet_scan('s3://boilingdata-demo/hits.parquet');`,
6	3781	1157	190	`SELECT COUNT(DISTINCT SearchPhrase) FROM parquet_scan('s3://boilingdata-demo/hits.parquet');`,
7	1096	521	166	`SELECT MIN(EventDate), MAX(EventDate) FROM parquet_scan('s3://boilingdata-demo/hits.parquet');`,
8	1283	510	174	`SELECT AdvEngineID, COUNT(*) FROM parquet_scan('s3://boilingdata-demo/hits.parquet') WHERE AdvEngineID <> 0 GROUP BY AdvEngineID ORDER BY COUNT(*) DESC;`,
9	3250	781	202	`SELECT RegionID, COUNT(DISTINCT UserID) AS u FROM parquet_scan('s3://boilingdata-demo/hits.parquet') GROUP BY RegionID ORDER BY u DESC LIMIT 10;`,
10	3823	942	198	`SELECT RegionID, SUM(AdvEngineID), COUNT(*) AS c, AVG(ResolutionWidth), COUNT(DISTINCT UserID) FROM parquet_scan('s3://boilingdata-demo/hits.parquet') GROUP BY RegionID ORDER BY c DESC LIMIT 10;`,
11	1601	528	173	`SELECT MobilePhoneModel, COUNT(DISTINCT UserID) AS u FROM parquet_scan('s3://boilingdata-demo/hits.parquet') WHERE MobilePhoneModel <> '' GROUP BY MobilePhoneModel ORDER BY u DESC LIMIT 10;`,
12	1633	583	181	`SELECT MobilePhone, MobilePhoneModel, COUNT(DISTINCT UserID) AS u FROM parquet_scan('s3://boilingdata-demo/hits.parquet') WHERE MobilePhoneModel <> '' GROUP BY MobilePhone, MobilePhoneModel ORDER BY u DESC LIMIT 10;`,
13	3141	1201	169	`SELECT SearchPhrase, COUNT(*) AS c FROM parquet_scan('s3://boilingdata-demo/hits.parquet') WHERE SearchPhrase <> '' GROUP BY SearchPhrase ORDER BY c DESC LIMIT 10;`,

14	3704	1346	189	`SELECT SearchPhrase, COUNT(DISTINCT UserID) AS u FROM parquet_scan('s3://boilingdata-demo/hits.parquet') WHERE SearchPhrase <> '' GROUP BY SearchPhrase ORDER BY u DESC LIMIT 10;`,
15	3168	1186	192	`SELECT SearchEngineID, SearchPhrase, COUNT(*) AS c FROM parquet_scan('s3://boilingdata-demo/hits.parquet') WHERE SearchPhrase <> '' GROUP BY SearchEngineID, SearchPhrase ORDER BY c DESC LIMIT 10;`,
16	3894	767	210	`SELECT UserID, COUNT(*) FROM parquet_scan('s3://boilingdata-demo/hits.parquet') GROUP BY UserID ORDER BY COUNT(*) DESC LIMIT 10;`,
17	4757	1410	199	`SELECT UserID, SearchPhrase, COUNT(*) FROM parquet_scan('s3://boilingdata-demo/hits.parquet') GROUP BY UserID, SearchPhrase ORDER BY COUNT(*) DESC LIMIT 10;`,
18	4467	1364	177	`SELECT UserID, SearchPhrase, COUNT(*) FROM parquet_scan('s3://boilingdata-demo/hits.parquet') GROUP BY UserID, SearchPhrase LIMIT 10;`,
19	7131	1585	218	`SELECT UserID, DATE_TRUNC('minute', make_timestamp(EventTime*1000000::bigint)) AS m, SearchPhrase, COUNT(*) FROM parquet_scan('s3://boilingdata-demo/hits.parquet') GROUP BY UserID, m, SearchPhrase ORDER BY COUNT(*) DESC LIMIT 10;`,
20	1138	456	182	`SELECT UserID FROM parquet_scan('s3://boilingdata-demo/hits.parquet') WHERE UserID = 435090932899640449;`,
21	1134	569	210	`SELECT COUNT(*) FROM parquet_scan('s3://boilingdata-demo/hits.parquet') WHERE URL LIKE '%google%';`,
22	1455	701	177	`SELECT SearchPhrase, MIN(URL), COUNT(*) AS c FROM parquet_scan('s3://boilingdata-demo/hits.parquet') WHERE URL LIKE '%google%' AND SearchPhrase <> '' GROUP BY SearchPhrase ORDER BY c DESC LIMIT 10;`,
23	1721	597	188	`SELECT SearchPhrase, MIN(URL), MIN(Title), COUNT(*) AS c, COUNT(DISTINCT UserID) FROM parquet_scan('s3://boilingdata-demo/hits.parquet') WHERE Title LIKE '%Google%' AND URL NOT LIKE '%.google.%' AND SearchPhrase <> '' GROUP BY SearchPhrase ORDER BY c DESC LIMIT 10;`,
24	1589	884	353	`SELECT * FROM parquet_scan('s3://boilingdata-demo/hits.parquet') WHERE URL LIKE '%google%' ORDER BY EventTime LIMIT 10;`,
25	1032	481	184	`SELECT SearchPhrase FROM parquet_scan('s3://boilingdata-demo/hits.parquet') WHERE SearchPhrase <> '' ORDER BY EventTime LIMIT 10;`,
26	1091	489	168	`SELECT SearchPhrase FROM parquet_scan('s3://boilingdata-demo/hits.parquet') WHERE SearchPhrase <> '' ORDER BY SearchPhrase LIMIT 10;`,
27	997	466	173	`SELECT SearchPhrase FROM parquet_scan('s3://boilingdata-demo/hits.parquet') WHERE SearchPhrase <> '' ORDER BY EventTime, SearchPhrase LIMIT 10;`,
28	2413	757	189	`SELECT CounterID, AVG(STRLEN(URL)) AS l, COUNT(*) AS c FROM parquet_scan('s3://boilingdata-demo/hits.parquet') WHERE URL <> '' GROUP BY CounterID HAVING COUNT(*) > 100000 ORDER BY l DESC LIMIT 25;`,

29	9452	3815	287	`SELECT REGEXP_REPLACE(Referer, '^https?://(?:www\.)?([^/]+)/.*$', '\x01') AS k, AVG(STRLEN(Referer)) AS l, COUNT(*) AS c, MIN(Referer) FROM parquet_scan('s3://boilingdata-demo/hits.parquet') WHERE Referer <> '' GROUP BY k HAVING COUNT(*) > 100000 ORDER BY l DESC LIMIT 25;`,
30	5075	1034	279	`SELECT SUM(ResolutionWidth), SUM(ResolutionWidth + 1), SUM(ResolutionWidth + 2), SUM(ResolutionWidth + 3), SUM(ResolutionWidth + 4), SUM(ResolutionWidth + 5), SUM(ResolutionWidth + 6), SUM(ResolutionWidth + 7), SUM(ResolutionWidth + 8), SUM(ResolutionWidth + 9), SUM(ResolutionWidth + 10), SUM(ResolutionWidth + 11), SUM(ResolutionWidth + 12), SUM(ResolutionWidth + 13), SUM(ResolutionWidth + 14), SUM(ResolutionWidth + 15), SUM(ResolutionWidth + 16), SUM(ResolutionWidth + 17), SUM(ResolutionWidth + 18), SUM(ResolutionWidth + 19), SUM(ResolutionWidth + 20), SUM(ResolutionWidth + 21), SUM(ResolutionWidth + 22), SUM(ResolutionWidth + 23), SUM(ResolutionWidth + 24), SUM(ResolutionWidth + 25), SUM(ResolutionWidth + 26), SUM(ResolutionWidth + 27), SUM(ResolutionWidth + 28), SUM(ResolutionWidth + 29), SUM(ResolutionWidth + 30), SUM(ResolutionWidth + 31), SUM(ResolutionWidth + 32), SUM(ResolutionWidth + 33), SUM(ResolutionWidth + 34), SUM(ResolutionWidth + 35), SUM(ResolutionWidth + 36), SUM(ResolutionWidth + 37), SUM(ResolutionWidth + 38), SUM(ResolutionWidth + 39), SUM(ResolutionWidth + 40), SUM(ResolutionWidth + 41), SUM(ResolutionWidth + 42), SUM(ResolutionWidth + 43), SUM(ResolutionWidth + 44), SUM(ResolutionWidth + 45), SUM(ResolutionWidth + 46), SUM(ResolutionWidth + 47), SUM(ResolutionWidth + 48), SUM(ResolutionWidth + 49), SUM(ResolutionWidth + 50), SUM(ResolutionWidth + 51), SUM(ResolutionWidth + 52), SUM(ResolutionWidth + 53), SUM(ResolutionWidth + 54), SUM(ResolutionWidth + 55), SUM(ResolutionWidth + 56), SUM(ResolutionWidth + 57), SUM(ResolutionWidth + 58), SUM(ResolutionWidth + 59), SUM(ResolutionWidth + 60), SUM(ResolutionWidth + 61), SUM(ResolutionWidth + 62), SUM(ResolutionWidth + 63), SUM(ResolutionWidth + 64), SUM(ResolutionWidth + 65), SUM(ResolutionWidth + 66), SUM(ResolutionWidth + 67), SUM(ResolutionWidth + 68), SUM(ResolutionWidth + 69), SUM(ResolutionWidth + 70), SUM(ResolutionWidth + 71), SUM(ResolutionWidth + 72), SUM(ResolutionWidth + 73), SUM(ResolutionWidth + 74), SUM(ResolutionWidth + 75), SUM(ResolutionWidth + 76), SUM(ResolutionWidth + 77), SUM(ResolutionWidth + 78), SUM(ResolutionWidth + 79), SUM(ResolutionWidth + 80), SUM(ResolutionWidth + 81), SUM(ResolutionWidth + 82), SUM(ResolutionWidth + 83), SUM(ResolutionWidth + 84), SUM(ResolutionWidth + 85), SUM(ResolutionWidth + 86), SUM(ResolutionWidth + 87), SUM(ResolutionWidth + 88), SUM(ResolutionWidth + 89) FROM parquet_scan('s3://boilingdata-demo/hits.parquet');`,
31	2105	645	185	`SELECT SearchEngineID, ClientIP, COUNT(*) AS c, SUM(IsRefresh), AVG(ResolutionWidth) FROM parquet_scan('s3://boilingdata-demo/hits.parquet') WHERE SearchPhrase <> '' GROUP BY SearchEngineID, ClientIP ORDER BY c DESC LIMIT 10;`,
32	2633	1306	180	`SELECT WatchID, ClientIP, COUNT(*) AS c, SUM(IsRefresh), AVG(ResolutionWidth) FROM parquet_scan('s3://boilingdata-demo/hits.parquet') WHERE SearchPhrase <> '' GROUP BY WatchID, ClientIP ORDER BY c DESC LIMIT 10;`,
33	11145	1584	233	`SELECT WatchID, ClientIP, COUNT(*) AS c, SUM(IsRefresh), AVG(ResolutionWidth) FROM parquet_scan('s3://boilingdata-demo/hits.parquet') GROUP BY WatchID, ClientIP ORDER BY c DESC LIMIT 10;`,
34	10306	2877	227	`SELECT URL, COUNT(*) AS c FROM parquet_scan('s3://boilingdata-demo/hits.parquet') GROUP BY URL ORDER BY c DESC

				LIMIT 10;`,
35	10725	2926	226	`SELECT 1, URL, COUNT(*) AS c FROM parquet_scan('s3://boilingdata-demo/hits.parquet') GROUP BY 1, URL ORDER BY c DESC LIMIT 10;`,
36	4147	946	187	`SELECT ClientIP, ClientIP - 1, ClientIP - 2, ClientIP - 3, COUNT(*) AS c FROM parquet_scan('s3://boilingdata-demo/hits.parquet') GROUP BY ClientIP, ClientIP - 1, ClientIP - 2, ClientIP - 3 ORDER BY c DESC LIMIT 10;`,
37	1588	520	155	`SELECT URL, COUNT(*) AS PageViews FROM parquet_scan('s3://boilingdata-demo/hits.parquet') WHERE CounterID = 62 AND EventDate >= '2013-07-01' AND EventDate <= '2013-07-31' AND DontCountHits = 0 AND IsRefresh = 0 AND URL <> '' GROUP BY URL ORDER BY PageViews DESC LIMIT 10;`,
38	1450	539	180	`SELECT Title, COUNT(*) AS PageViews FROM parquet_scan('s3://boilingdata-demo/hits.parquet') WHERE CounterID = 62 AND EventDate >= '2013-07-01' AND EventDate <= '2013-07-31' AND DontCountHits = 0 AND IsRefresh = 0 AND Title <> '' GROUP BY Title ORDER BY PageViews DESC LIMIT 10;`,
39	1508	533	155	`SELECT URL, COUNT(*) AS PageViews FROM parquet_scan('s3://boilingdata-demo/hits.parquet') WHERE CounterID = 62 AND EventDate >= '2013-07-01' AND EventDate <= '2013-07-31' AND IsRefresh = 0 AND IsLink <> 0 AND IsDownload = 0 GROUP BY URL ORDER BY PageViews DESC LIMIT 10 OFFSET 1000;`,
40	1645	570	158	`SELECT TraficSourceID, SearchEngineID, AdvEngineID, CASE WHEN (SearchEngineID = 0 AND AdvEngineID = 0) THEN Referer ELSE '' END AS Src, URL AS Dst, COUNT(*) AS PageViews FROM parquet_scan('s3://boilingdata-demo/hits.parquet') WHERE CounterID = 62 AND EventDate >= '2013-07-01' AND EventDate <= '2013-07-31' AND IsRefresh = 0 GROUP BY TraficSourceID, SearchEngineID, AdvEngineID, Src, Dst ORDER BY PageViews DESC LIMIT 10 OFFSET 1000;`,
41	1570	500	154	`SELECT URLHash, EventDate, COUNT(*) AS PageViews FROM parquet_scan('s3://boilingdata-demo/hits.parquet') WHERE CounterID = 62 AND EventDate >= '2013-07-01' AND EventDate <= '2013-07-31' AND IsRefresh = 0 AND TraficSourceID IN (-1, 6) AND RefererHash = 3594120000172545465 GROUP BY URLHash, EventDate ORDER BY PageViews DESC LIMIT 10 OFFSET 100;`,
42	1535	485	170	`SELECT WindowClientWidth, WindowClientHeight, COUNT(*) AS PageViews FROM parquet_scan('s3://boilingdata-demo/hits.parquet') WHERE CounterID = 62 AND EventDate >= '2013-07-01' AND EventDate <= '2013-07-31' AND IsRefresh = 0 AND DontCountHits = 0 AND URLHash = 2868770270353813622 GROUP BY WindowClientWidth, WindowClientHeight ORDER BY PageViews DESC LIMIT 10 OFFSET 10000;`,
43	1717	674	154	`SELECT DATE_TRUNC('minute', make_timestamp(EventTime*1000000::bigint)) AS M, COUNT(*) AS PageViews FROM parquet_scan('s3://boilingdata-demo/hits.parquet') WHERE CounterID = 62 AND EventDate >= '2013-07-14' AND EventDate <= '2013-07-15' AND IsRefresh = 0 AND DontCountHits = 0 GROUP BY DATE_TRUNC('minute', make_timestamp(EventTime*1000000::bigint)) ORDER BY DATE_TRUNC('minute', make_timestamp(EventTime*1000000::bigint)) LIMIT 10 OFFSET 1000;`,