10 Good Reasons to Use

ClickHouse

by Ramazan Polat

ClickHouse Meetup Ankara - Nov 19, 2019

Who I am

Ramazan Polat

Former **DBA**, currently **Software Architect** at SGK

SGK(Social Security Institution) is the largest* government institution of Turkey in terms of *in-house generated data size* and *transactions per second*.

* *Possibly, not sure about it*

Why I am here

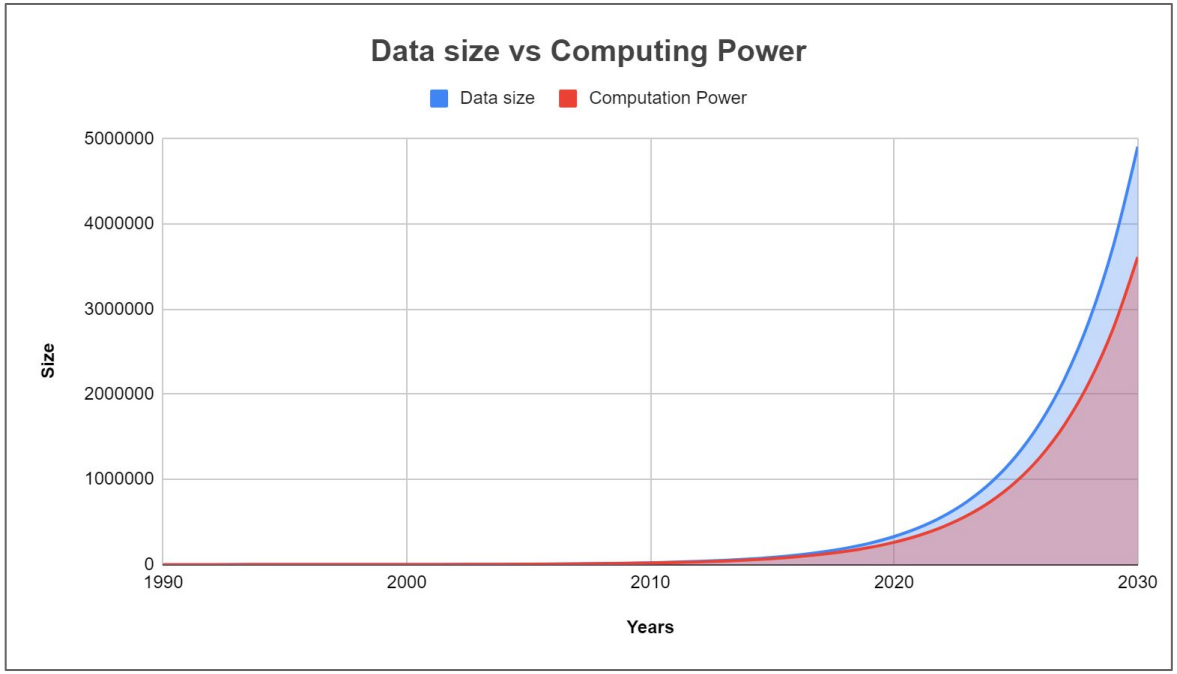
I've been using a number of database management systems for 15+ years, but ended up always using ClickHouse, even for cases where ClickHouse doesn't perfectly fit.

Why?

I have a lot of reasons, here is **10** of them, so you can use ClickHouse too.

Remember these

1) Data is growing faster than computing power*



2) Unix timestamp is currently over 1.5 Billion (in fact, as of writing this, it is 1574114808)

* https://www.newsweek.com/2014/08/15/computers-need-be-more-human-brains-262504.html

#1 Speed

- ✓ *Inserts are instant!*
- ✓ *Selects are blazing fast!*
- ✓ *Handling billions of rows sub-second!*

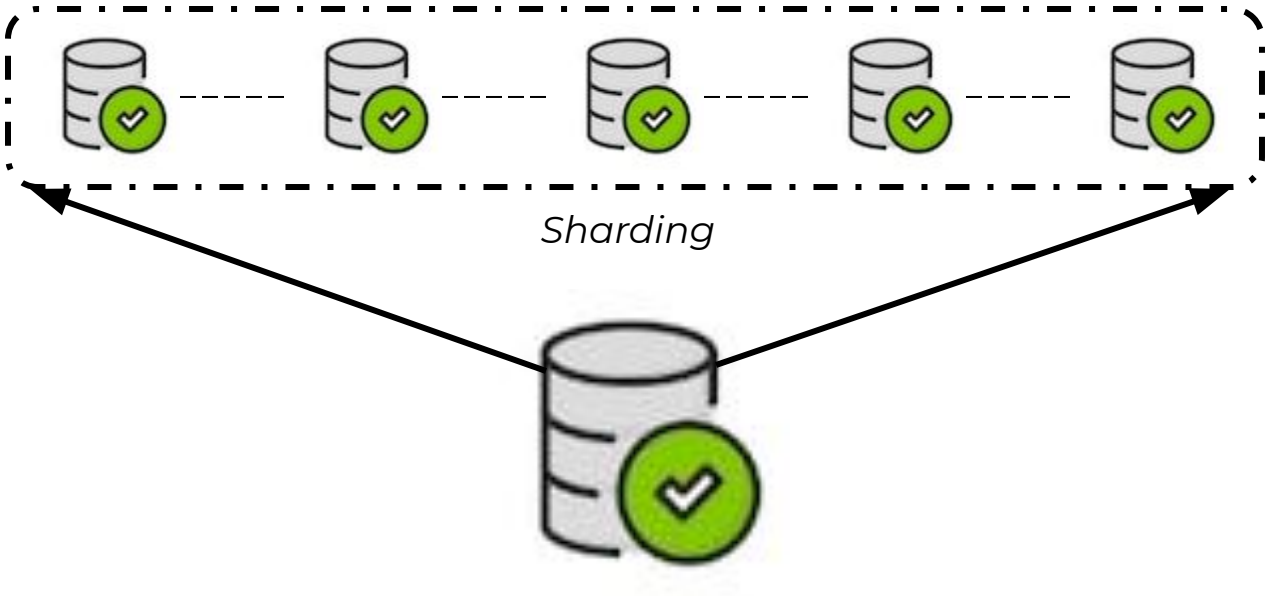
#2 Scalability

✓ *Uses all CPU cores in single machine*

*Thanks to **vectorized execution** and **parallel processing***

#2 Scalability

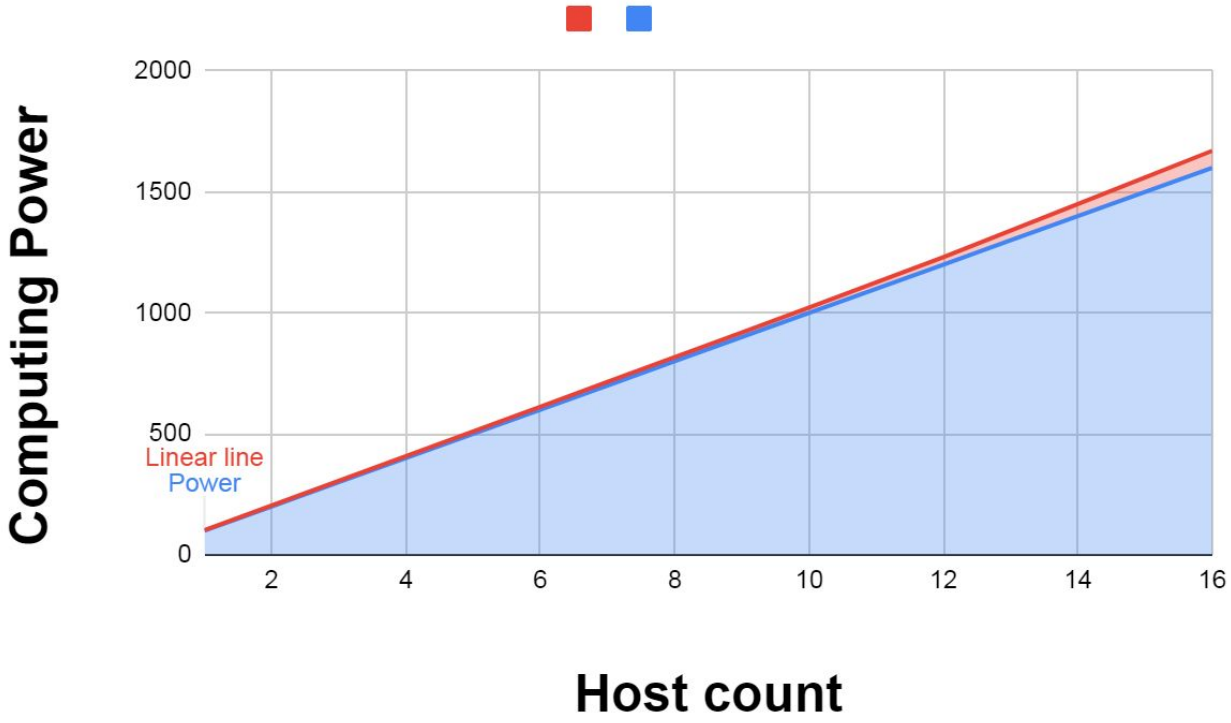
✓ *Scales horizontally across multiple hosts*



From clients, it looks like one giant database

#2 Scalability

✓ *Linearly scaling.*



#3 Compression

- ✓ *Compression is column based*
- ✓ *Encoding helps data compress better*
- ✓ *Ritch compression options*
- ✓ *Custom compression for different type of data*

#3 Compression

Encoding maps data in a different bit layout

Encodings	
LowCardinality	String with a few values(up to 10K)
Delta	Difference between consecutive values
DoubleDelta	Difference between consecutive deltas, good for slowly changing sequences
Gorilla	Efficient for values that does not change often
T64	Strips lower and higher bits that does not change, good for big numbers in a small range

#3 Compression

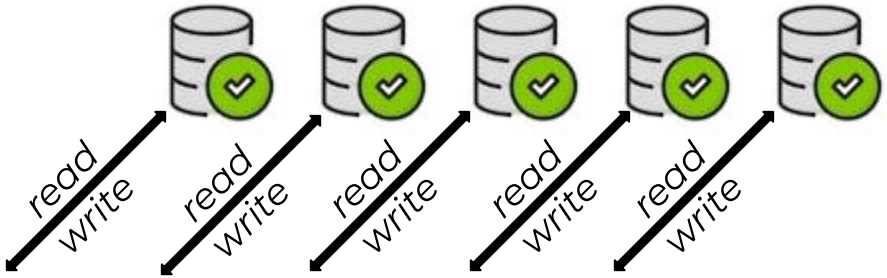
There is a trade-off between speed and compression ratio

Compression algos	
LZ4	Faster compression with smaller compression ratio
ZSTD	Slower compression but compresses better

```
CREATE TABLE test (
    city LowCardinality(String),
    speed UInt32 CODEC(Delta),
    acceleration UInt32 CODEC(DoubleDelta),
    humidity UInt32 CODEC(Gorilla, LZ4),
    year UInt32 CODEC(T64, ZSTD)
) ENGINE = ...
```

#4 Production Ready

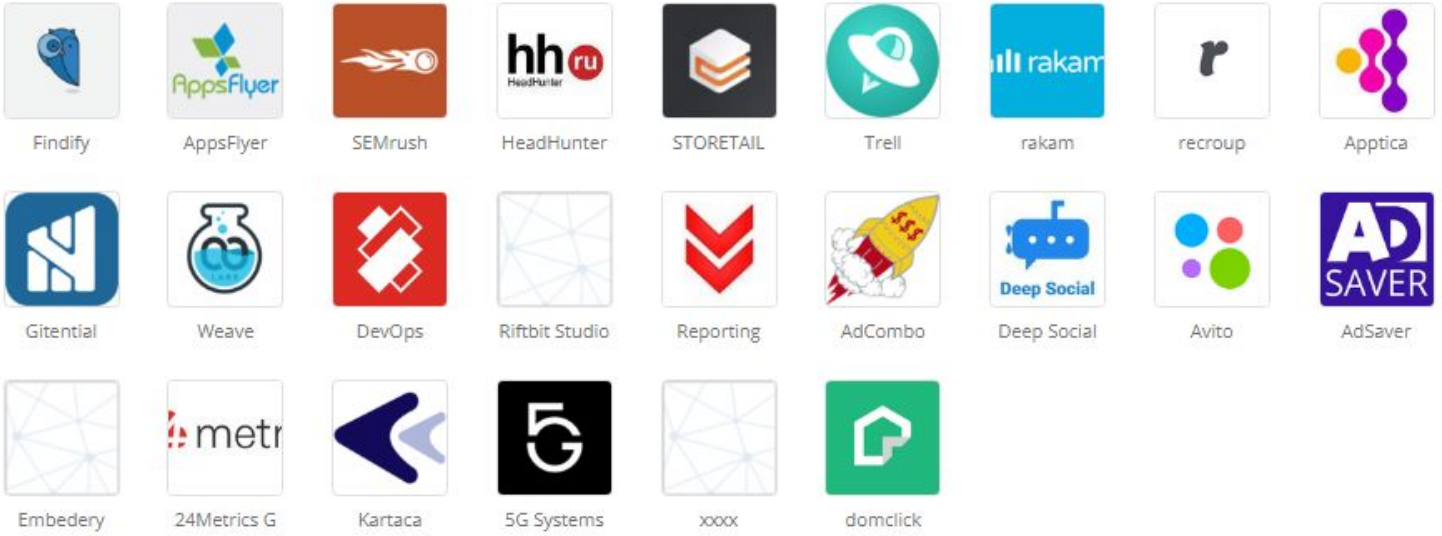
✓ *Fault tolerant*



Multi-master replication

#4 Production Ready

✓ *Used by several companies worldwide*



This list is taken from stackshare.io/clickhouse

#4 Production Ready

✓ *Already being used in Yandex Metrica**

A cluster of **600+** servers
More than **30 trillion** rows
20 billion events per day
17 PB data(**2 PB** compressed)

Just a reminder**: The unixtimestamp is number of seconds that have elapsed since the 1 January 1970, which is around **1.5 Billion

** Yandex Metrica is one of the World's largest analytics database*

#5 Integration

✓ *Connects to any other JDBC database so you can query them just like a ClickHouse table*


```
SELECT *
FROM jdbc('mysql://localhost:3306/?user=root&password=root', 'schema', 'table')
```

*This means **you can combine any JDBC database** with ClickHouse*

#5 Integration

✓ *Even can connect to another ClickHouse without any configuration*

```
SELECT * FROM remote('ch2', 'db_name', 'table_name', 'user', 'password')
```

✓ *Connects to REST services!*

```
SELECT * FROM url('http://127.0.0.1:12345/', CSV, 'column1 String, column2 UInt32')
```

#5 Integration

✓ *Clickhouse integrates into anything**

... and much more

* *almost*

#6 Awesome Table Capabilities

- ✓ *Enums*
- ✓ *Partitioning*
- ✓ *Arrays*
- ✓ *Nested columns*
- ✓ *TTL tables*
- ✓ *Materialized views*
- ✓ *Live views*

#6 Awesome Table Capabilities

Enums

```
CREATE TABLE enum_table(
    `day` Enum8('SUNDAY' = 0, 'MONDAY' = 1, 'TUESDAY' = 2,
        'WEDNESDAY' = 3, 'THURSDAY' = 4, 'FRIDAY' = 5, 'SATURDAY' = 6)
)
ENGINE = ...
```

```
INSERT INTO enum_table VALUES('SATURDAY'),('SUNDAY')
```

```
SELECT *
FROM enum_table
```

day
SATURDAY
SUNDAY

```
SELECT CASE day
  WHEN 0 THEN 'SUNDAY'
  WHEN 1 THEN 'MONDAY'
  WHEN 2 THEN 'TUESDAY'
  WHEN 3 THEN 'WEDNESDAY'
  WHEN 4 THEN 'THURSDAY'
  WHEN 5 THEN 'FRIDAY'
  WHEN 6 THEN 'SATURDAY'
END FROM enum_table;
```

#6 Awesome Table Capabilities

Partitioning

- *Each partition is stored separately in order to simplify manipulations of this data*
- *Each partition can be detached, attached or dropped instantly*

```
CREATE TABLE visits(
    VisitDate Date,
    Hour UInt8,
    ClientID UUID
) ENGINE = MergeTree()
PARTITION BY toYYYYMM(VisitDate)
ORDER BY Hour;
```

```
PARTITION BY (toMonday(StartDate), EventType)
```

```
ALTER TABLE visits DETACH PARTITION 201910;

ALTER TABLE visits ATTACH PARTITION 201911;

ALTER TABLE visits DROP PARTITION 201911;
```

Partitions	VisitDate	Hour	ClientID
Part_1	2019-10-11	14	e0d6e0ff
	2019-10-12	16	d2af7d50
	2019-10-28	17	1aef1ff5
Part_2	2019-11-01	18	0f4def0b
	2019-11-17	19	46638b95
	2019-11-21	21	d6e0af7d
	2019-11-23	23	38b9ef1f

#6 Awesome Table Capabilities

Arrays

- *Columns can be array of any type*
- *Great flexibility*

```
CREATE TABLE timeseries(
    entity String,
    ts UInt64,
    m Array(String),
    v Array(Float32),
    d Date DEFAULT toDate(ts)
) ENGINE = MergeTree(d, ts, 8192)
```

entity	ts	m	v
cpu1	1574018595	[temp,load]	[78, 0.85]
cpu2	1574018674	[load]	[0.44]
cpu7	1574019333	[ghz, temp]	[3.1, 0.4]
cpu4	1574019501	[load, ghz]	[0.9, 3.8]

#6 Awesome Table Capabilities

Nested Data

```
CREATE TABLE customers(
    custId UInt32,
    name String,
    Orders Nested(
        date Date,
        items UInt16,
        price Float32)
) ENGINE = TinyLog()
```

custId	name	Orders		
		date	*items*	*price*
123	Joe Lee	2019-01-23	3	€ 34.56
		date	*items*	*price*
234	Kate Hall	2019-01-24	2	€ 23.45
		2019-01-25	4	€ 21.09
		date	*items*	*price*
456	Ann Cook	2019-01-26	5	€ 18.91

```
INSERT INTO customers VALUES (123, 'Joe Lee', ['2019-01-23'], [3], [34.56])
```

#6 Awesome Table Capabilities

TTL Tables

Automatically deletes rows based on a conditions

```
CREATE TABLE traffic2(
    datetime DateTime,
    custId UInt32
) ENGINE = MergeTree
PARTITION BY toYYYYMM(datetime)
ORDER BY (datetime, custId)
TTL datetime + INTERVAL 6 MONTH;
```

Deletes data after 6 months

#6 Awesome Table Capabilities

Materialized Views

Automatically aggregates data on inserts

```
SELECT *
FROM CCEvents
```

dt	custId	name	ccNumber	posId	purchase
2019-10-11 21:06:49	987	Kate	XXXX-XXXX-XXXX-XXXX	557	98.9
2019-10-17 12:00:01	123	Joe	XXXX-XXXX-XXXX-XXXX	345	145.3
2019-10-18 12:00:01	123	Joe	XXXX-XXXX-XXXX-XXXX	124	17.4
2019-10-19 12:00:01	123	Joe	XXXX-XXXX-XXXX-XXXX	548	51
2019-11-01 21:06:49	987	Kate	XXXX-XXXX-XXXX-XXXX	345	61.1
2019-11-01 21:06:49	987	Kate	XXXX-XXXX-XXXX-XXXX	345	61.1
2019-11-13 12:00:01	123	Joe	XXXX-XXXX-XXXX-XXXX	852	45.3
2019-11-14 21:06:49	123	Joe	XXXX-XXXX-XXXX-XXXX	499	245.6
2019-11-15 21:06:49	123	Joe	XXXX-XXXX-XXXX-XXXX	756	5.8
2019-11-21 21:06:49	987	Kate	XXXX-XXXX-XXXX-XXXX	345	83.4
2019-12-07 21:06:49	987	Kate	XXXX-XXXX-XXXX-XXXX	345	20.2
2019-12-27 21:06:49	987	Kate	XXXX-XXXX-XXXX-XXXX	345	21.3

#6 Awesome Table Capabilities

Materialized Views

Automatically aggregates data on inserts

```
SELECT
    name,
    toYYYYMM(dt) AS month,
    sum(purchase)
FROM CCEvents
GROUP BY
    name,
    month
```

name	month	sum(purchase)
Joe	201911	296.70000553131104
Kate	201910	98.9000015258789
Kate	201911	205.5999984741211
Joe	201910	213.7000026702881
Kate	201912	41.5

5 rows in set. Elapsed: 0.010 sec.

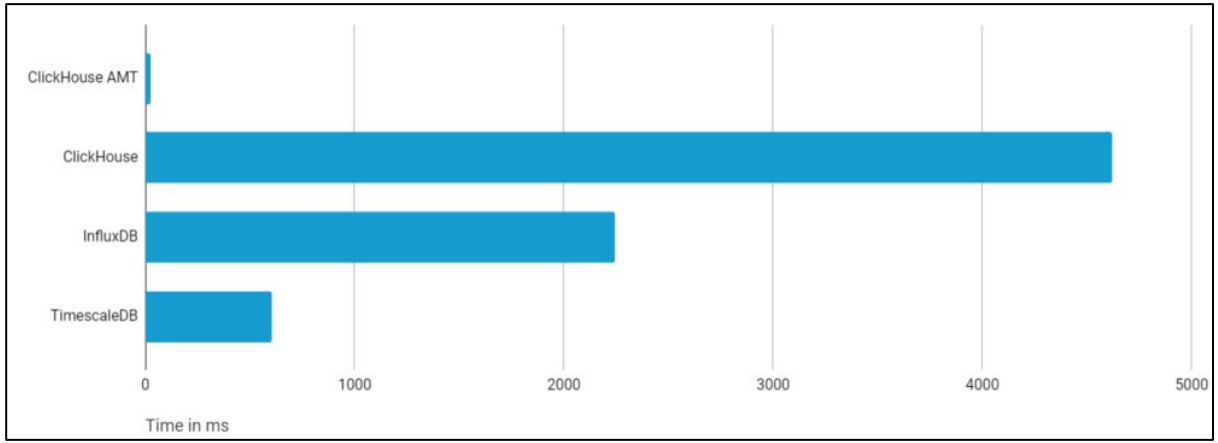
```
CREATE MATERIALIZED VIEW CCMonthlySums
ENGINE = SummingMergeTree
ORDER BY (name, month)
POPULATE AS
SELECT
    name,
    toYYYYMM(dt) AS month,
    sum(purchase)
FROM CCEvents
GROUP BY
    name,
    month
```

#6 Awesome Table Capabilities

Materialized Views

Automatically aggregates data on inserts

```
SELECT *
FROM CCMonthlySums

┌─name──┬─month──┬─sum(purchase)─┐
│ Joe   │ 201910 │ 213.7000026702881 │
│ Joe   │ 201911 │ 296.70000553131104 │
│ Kate  │ 201910 │ 98.9000015258789 │
│ Kate  │ 201911 │ 205.5999984741211 │
│ Kate  │ 201912 │ 41.5 │
└───────┴────────┴──────────────┘

5 rows in set. Elapsed: 0.010 sec.
```



*A benchmark performed by **Altinity** shows how fast materialized views are**

*A quote from conclusion of the benchmark: "**Using ClickHouse AggregatingMergeTree technique we decreased response time of the last point query from 4.5s to 20ms -- this is more than a 200x improvement**"*

** ClickHouse Continues to Crush Time Series, https://www.altinity.com/blog/clickhouse-continues-to-crush-time-series*

#6 Awesome Table Capabilities

Live Views

When the conditions are met, you get a instant notification

```
SELECT *
FROM CCEvents
```

dt	custId	name	ccNumber	posId	purchase
2019-10-11 21:06:49	987	Kate	XXXX-XXXX-XXXX-XXXX	557	98.9
2019-10-17 12:00:01	123	Joe	XXXX-XXXX-XXXX-XXXX	345	145.3
2019-10-18 12:00:01	123	Joe	XXXX-XXXX-XXXX-XXXX	124	17.4
2019-10-19 12:00:01	123	Joe	XXXX-XXXX-XXXX-XXXX	548	51
2019-11-01 21:06:49	987	Kate	XXXX-XXXX-XXXX-XXXX	345	61.1
2019-11-01 21:06:49	987	Kate	XXXX-XXXX-XXXX-XXXX	345	61.1
2019-11-13 12:00:01	123	Joe	XXXX-XXXX-XXXX-XXXX	852	45.3
2019-11-14 21:06:49	123	Joe	XXXX-XXXX-XXXX-XXXX	499	245.6
2019-11-15 21:06:49	123	Joe	XXXX-XXXX-XXXX-XXXX	756	5.8
2019-11-21 21:06:49	987	Kate	XXXX-XXXX-XXXX-XXXX	345	83.4
2019-12-07 21:06:49	987	Kate	XXXX-XXXX-XXXX-XXXX	345	20.2
2019-12-27 21:06:49	987	Kate	XXXX-XXXX-XXXX-XXXX	345	21.3

#6 Awesome Table Capabilities

Live Views

*We want to track **Kate's spendings**, so we create a Live View*

```
SELECT
    name,
    toYYYYMM(dt) AS month,
    sum(purchase) AS spending
FROM CCEvents
WHERE (name = 'Kate') AND (month = 201912)
GROUP BY
    name,
    month
```

name	month	spending
Kate	201912	41.5

```
CREATE LIVE VIEW kate_over_spending AS
SELECT
    name,
    toYYYYMM(dt) AS month,
    sum(purchase) AS spending
FROM CCEvents
WHERE (name = 'Kate') AND (month = 201912)
GROUP BY
    name,
    month
```

#6 Awesome Table Capabilities

Live View Tables

When the conditions are met, you get a instant notification

```
WATCH kate_over_spending

┌─name─┬─month──┬─spending─┬─_version─┐
│ Kate │ 201912 │     41.5 │        1 │
└──────┴────────┴──────────┴──────────┘
↘ Progress: 1.00 rows, 33.00 B (0.01 rows/s., 0.17 B/s.)
```

```
INSERT INTO CCEvents VALUES('2019-12-28 13:08:01', 987, 'Kate', 'XXXX-XXXX-XXXX-XXXX', 345, 30);
```

```
WATCH kate_over_spending

┌─name─┬─month──┬─spending─┬─_version─┐
│ Kate │ 201912 │     41.5 │        1 │
└──────┴────────┴──────────┴──────────┘
┌─name─┬─month──┬─spending─┬─_version─┐
│ Kate │ 201912 │     71.5 │        2 │
└──────┴────────┴──────────┴──────────┘
← Progress: 2.00 rows, 66.00 B (0.01 rows/s., 0.20 B/s.)
```

#7 Better SQL+

- ✓ *Formats*
- ✓ *Great functions:*
  - *countIf, sumIf, arrayJoin, groupArray, geoLocation, JSON*
- ✓ *Lambda functions*
- ✓ *Resolving expression names*
- ✓ *system.numbers*
- ✓ *Order by condition*

Format	INSERT	SELECT
TabSeparated	✓	✓
CSV	✓	✓
Values	✓	✓
Vertical	✗	✓
JSON	✗	✓
TSKV	✓	✓
Protobuf	✓	✓
RowBinary	✓	✓
Null	✗	✓
XML	✗	✓
CapnProto	✓	✗

#7 Better SQL+

✓ *Conditional aggregates*

```
SELECT
    name,
    sum(purchase) AS spending,
    count(*) AS event_count
FROM CCEvents
GROUP BY name
```

name	spending	event_count
Kate	376	7
Joe	510.4000082015991	6

vs

```
SELECT
    sumIf(purchase, name = 'Kate') AS kate_spending,
    countIf(name, name = 'Joe') AS joe_event_count
FROM CCEvents
```

kate_spending	joe_event_count
376	6

#7 Better SQL+

✓ *Array functions*

arrayJoin *converts arrays to rows while* ***groupArray*** *does the opposite*

```
SELECT arrayJoin([1, 2, 3]) AS arr
```

```
┌─arr─┐
│   1 │
│   2 │
│   3 │
└─────┘
```

```
SELECT groupArray(arr)
FROM (SELECT arrayJoin([1, 2, 3]) AS arr)
```

```
┌─groupArray(arr)─┐
│ [1,2,3]         │
└─────────────────┘
```

```
SELECT
    arrayJoin([1, 2, 3]) AS nums,
    arrayJoin(['a', 'b']) AS chars
```

```
┌─nums─┬─chars─┐
│    1 │ a     │
│    1 │ b     │
│    2 │ a     │
│    2 │ b     │
│    3 │ a     │
│    3 │ b     │
└──────┴───────┘
```

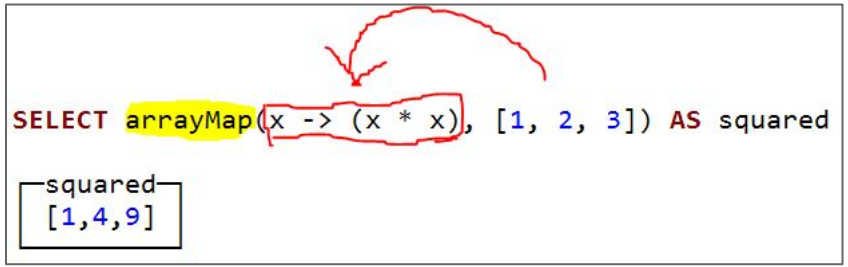
Some other array functions:

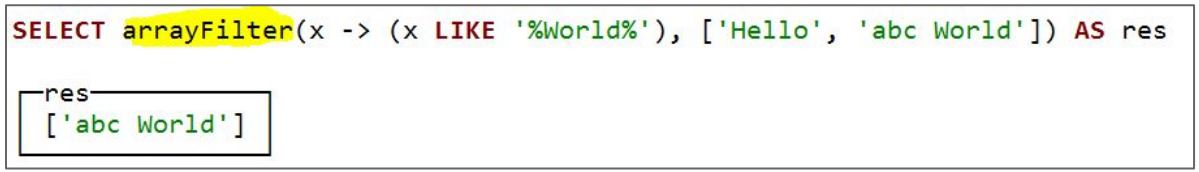
- **indexOf**
- **arrayDistinct**
- **arrayReverse**
- **arrayConcat**
- **has**
- **hasAll**

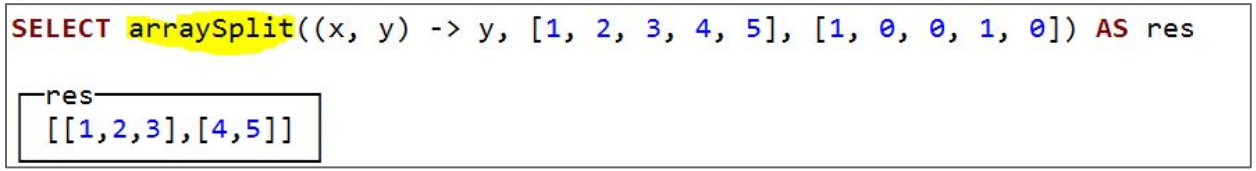
... and more

#7 Better SQL+

✓ *Lambda functions*







Some other functions accepting lambdas:

- **arraySum**
- **arrayCount**
- **arrayAll**
- **arraySort**
- **arrayFill**

... and more

#7 Better SQL+

✓ *Real example for lambda functions: calculating* ***euclidean distance***

$$d(p,q) = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2}$$

$$d(p,q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + (p_3 - q_3)^2}$$

```
SELECT *
FROM embeddings
```

name	e
Joe	[0.93,0.89,0.93,...,0.11,0.58,0.47,0.81,0.95,0.4,0.4,0.76,0.8]
Joe	[0.16,0.22,0.32,...,0.17,0.9,0.91,0.3,0.69,0.77,0.63,0.52,0.5]

```
SELECT
    name,
    sqrt(arraySum(arrayMap((a, b) -> pow(a - b, 2), e, t))) AS distance
FROM (
    SELECT
        name,
        e,
        (SELECT e AS target FROM embeddings WHERE name = 'Kate' ) AS t
    FROM embeddings
)
```

name	distance
Joe	2.5301778700047444
Kate	0

This works much more faster that anything else because of ***vectorized execution***

#7 Better SQL+

✓ *Resolving expression names*

```sql
SELECT
    name,
    toYYYYMM(dt) AS month,
    sum(purchase) AS spending
FROM CCEvents
WHERE (name = 'Kate') AND (toYYYYMM(dt) = 201912)
GROUP BY
    name,
    toYYYYMM(dt)
HAVING sum(purchase) > 100
```

vs

```sql
SELECT
    name,
    toYYYYMM(dt) AS month,
    sum(purchase) AS spending
FROM CCEvents
WHERE (name = 'Kate') AND (month = 201912)
GROUP BY
    name,
    month
HAVING spending > 100
```

#8 REST Capabilities

✓ *REST server with* **HTTP PORT 8123**
✓ *REST client with* **SELECT FROM URL**

```
$ echo 'SELECT 1 FORMAT JSON' | curl 'http://localhost:8123/' --data-binary @-
```

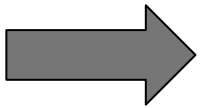

```
{
  "meta": [
    {"name": "1", "type": "UInt8"}
  ],
  "data": [
    {"1": 1}
  ],
  "rows": 1,
  "statistics": {
    "elapsed": 0.000067311,
    "rows_read": 1,
    "bytes_read": 1
  }
}
```

#8 REST Capabilities

✓ *REST server*

***Secret sauce**: HTTP interface + FORMAT JSON*

```
SELECT
    name,
    toYYYYMM(dt) AS month,
    sum(purchase) AS spending
FROM CCEvents
WHERE (name = 'Kate') AND
      (month = 201912)
GROUP BY
    name,
    month
FORMAT JSON
```



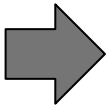

```
{
  "meta": [
    {"name": "name", "type": "String"},
    {"name": "month", "type": "UInt32"},
    {"name": "spending", "type": "Float64"}
  ],
  "data": [
    {"name": "Kate", "month": 201912, "spending": 71.5}
  ],
  "rows": 1,
  "statistics": {
    "elapsed": 0.002663858,
    "rows_read": 13,
    "bytes_read": 147
  }
}
```

#8 REST Capabilities

✓ *REST client*

```
from http.server
import BaseHTTPRequestHandler, HTTPServer

class CSVHTTPServer(BaseHTTPRequestHandler):
  def do_GET(self):
    self.send response(200)
    self.send header('Content-type', 'text/csv')
    self.end_headers()

    self.wfile.write(bytes('Hello,1\nWorld,2\n', "utf-8"))

if __name__ == "__main__":
  server_address = ('127.0.0.1', 12345)

HTTPServer(server_address, CSVHTTPServer).serve_forever()
```

Sample REST Server running at http://myserver.com



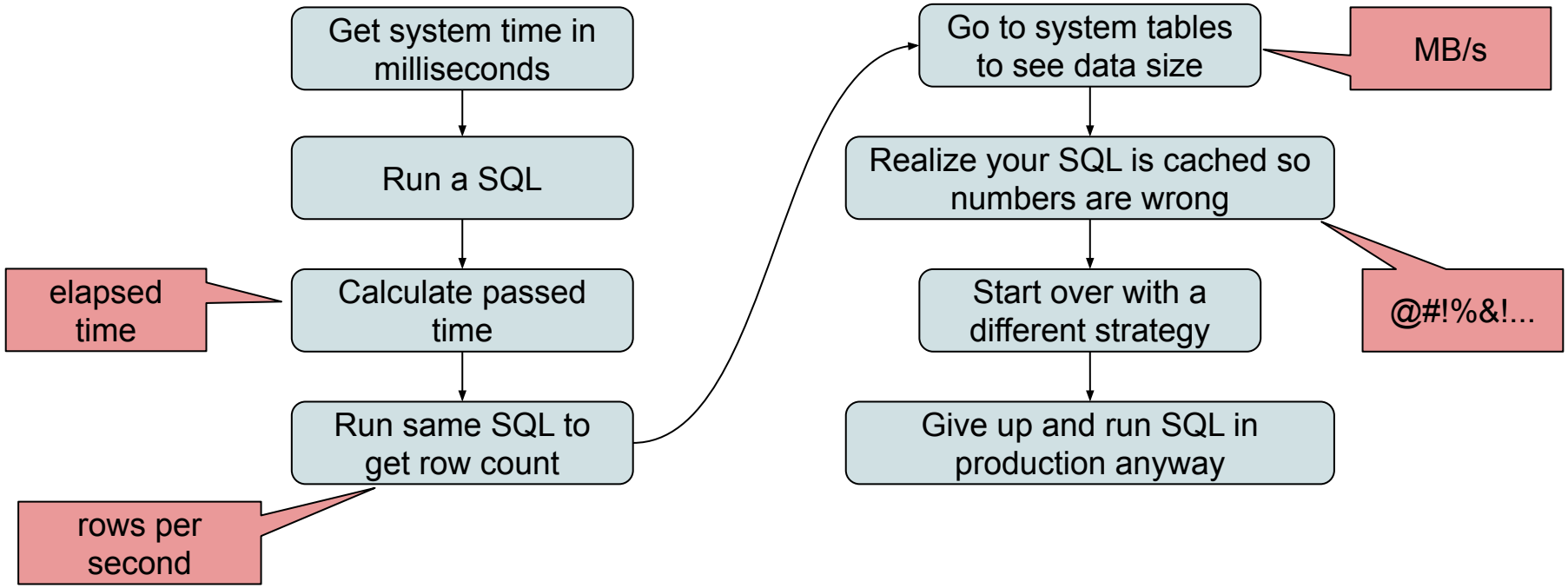
```
CREATE TABLE rest_server (word String, value UInt64)
ENGINE=URL('http://myserver.com:12345/', CSV);

SELECT * FROM rest_server;
```

word	value
Hello	1
World	2

Use it as local table

#9 CLI

Almost every programmer did something like this...



#9 CLI

✓ *CLI provides a lot of useful information*

```
SELECT
    table_name,
    requester,
    countIf(hit, hit = 1) AS hits,
    countIf(miss, miss = 1) AS misses,
    count() AS total,
    round((hits * 100) / total, 2) AS efficiency
FROM cachelog
GROUP BY
    table_name,
    requester
ORDER BY total DESC
LIMIT 10
```

table_name	requester	hits	misses	total	efficiency
isverenintratoken		18456146	38468	18494614	99.79
hd_v2		177647	7493254	7670901	2.32
hd_v2		20689	658119	678808	3.05
hd_v2test		77284	22	77306	99.97
isverentokentest		75086	137	75223	99.82
hd_v2		33013	41517	74530	44.29
hd_v2		9154	40209	49363	18.54
hd_v2		2998	45498	48496	6.18
hd_4aturksa		5026	8732	13758	36.53
hd_v2		9896	648	10544	93.85

elapsed time

rows per second

MB/s

```
10 rows in set. Elapsed: 0.711 sec. Processed 28.63 million rows, 174.83 MB (40.28 million rows/s., 246.02 MB/s.)

clickhouse :)
```

#9 CLI

✓ *CLI provides a lot of useful information*

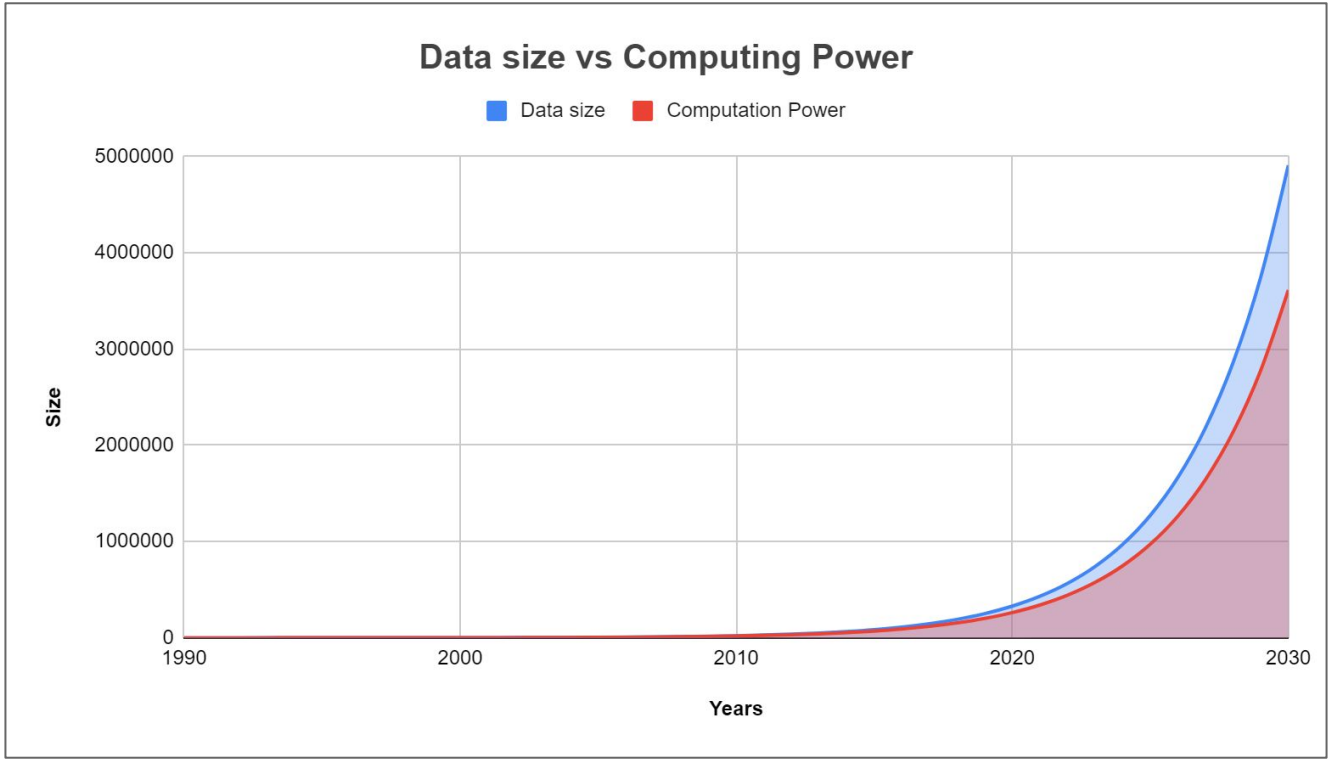
```
SELECT
    rand() % 100 AS percent,
    bar(percent, 1, 200, 100) AS bar
FROM system.numbers
LIMIT 10
```

#10 Great Ecosystem

- ✓ *Open Source*
- ✓ *Free of charge!*
- ✓ *Great community support*
- ✓ *Constantly Improving*

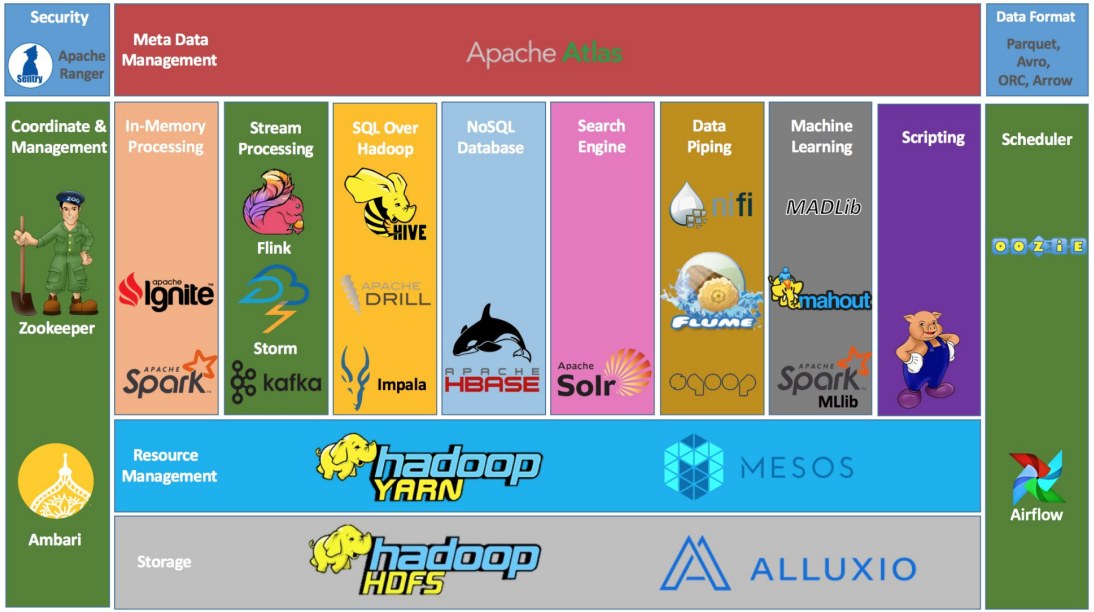
#11 Close the gap effectively

Data is growing faster than computing power*, so **we need to close the gap effectively**



* https://www.newsweek.com/2014/08/15/computers-need-be-more-human-brains-262504.html

#11 Close the gap effectively



Big Data Ecosystem

vs

ClickHouse

*"Because it is easier to **learn**/**use** **one tool** that does **most of things well enough** than using **lots of tools** which are designed to do **one thing good**"*

Honorable Mentions

- ✓ *Machine learning*
- ✓ *Data sampling*
- ✓ *Rich text functions(search, replace, RegEx, ngram)*
- ✓ *UUID, domain name and other helper functions*
- ✓ *External dictionaries*
- ✓ *Geo location*
- ✓ *clickhouse-local*
- ✓ *Distributed DDLs*

How come ClickHouse born in Yandex and become so good?

How come ClickHouse born in Yandex and become so good?

How come ClickHouse born in Yandex and become so good?

How come ClickHouse born in Yandex and become so good?

*ClickHouse is developed in Yandex Metrica by the developers that will actually use the product**

* *That's my opinion*

Thank You!

Contacts:
ramazanpolat@gmail.com

https://www.linkedin.com/in/ramazanpolat/

You can get presentations from:
https://github.com/ClickHouse/clickhouse-presentations