Migrating from Redshift to ClickHouse

iFunny is a fun picture and GIF app that lets users to pass the time looking at memes, comics, funny pictures, cat GIFs, etc. Plus, users can even upload their own content and share it. The iFunny app has been using Redshift for quite some time as a database for events in backend services and mobile apps. We went with them because in the beginning there really weren't any alternatives comparable in terms of cost and convenience.

However, the public release of ClickHouse was a real game changer. We studied it inside and out, compared the cost and possible architecture, and this summer finally decided to try it out and see if we could use it. This article is all about the challenge Redshift had been helping us solve and how we migrated this solution to ClickHouse.

CHALLENGE

iFunny required a service similar to Yandex.Metrika, but designed exclusively for internal use. Let me explain why.

External clients—including mobile apps, websites, or internal backend services—create events. It is very difficult to explain to these clients that the event handling service is currently unavailable, and that they should "try again in 15 minutes or maybe an hour." We have a lot of clients, they're constantly sending event messages and do not want to wait at all.

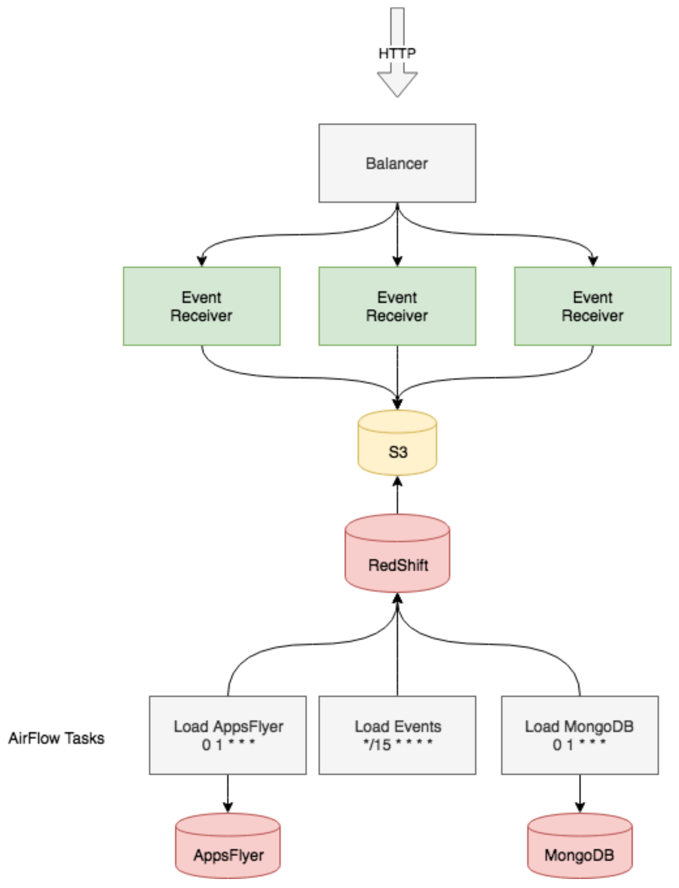
But we also have internal services and users that are rather tolerant in this respect: they can work fine even when the analytics service is unavailable. The majority of product metrics and A/B testing results only need to be checked once daily, or maybe even less. That is why requirements for reading are rather low. In the event of a disaster or update, database can be unavailable or lose consistency for several hours, or in the worst case, days.

If we dig into the numbers: we need to accept about 5 billion events per day (300 GB of compressed data), and at the same time maintain the data for three months in hot storage available for SQL queries. We also need to keep data in cold storage for a couple years or more while being able to return to hot storage in just a few days.

In general, our data are sets of time-ordered events. There are about three hundred types of events, each with its own set of properties. We also have some data from external sources that need to be synchronized with the analytics database: for example, a collection of app installations from MongoDB or the external AppsFlyer service.

In the end, we need about 40 TB of disk space for the database and around 250 TB more for cold storage.

REDSHIFT-POWERED SOLUTION



Okay, so we have mobile clients and backend services the events should be received from. An HTTP service receives the data, carries out basic validation, groups the events into files by minute of reception, saves them on a local disk, and then compresses and sends them to an S3 bucket. The availability of this service depends on the availability of application servers and AWS S3. Applications are stateless, which makes them easy to balance, scale, and replace. S3 is a rather simple file storage service with a good reputation and high availability, so we know we can rely on it.

Then we need to somehow get the data to Redshift. This part is relatively easy: the recommended method to upload data to Redshift is its built-in S3 importer. Every 10 minutes, a script is executed that connects to Redshift and asks it to request data at the prefix

```
s3://events-bucket/main/year=2018/month=10/day=
14/10_3*
```

We use Apache Airflow to track task uploading status because we can repeat the operation in the event of an error and it has an easy-to-read task execution log, which is crucial when the number of tasks is high. If we encounter any issues, we can repeat the upload for specific time intervals or upload cold data from S3 storage dating back up to a year.

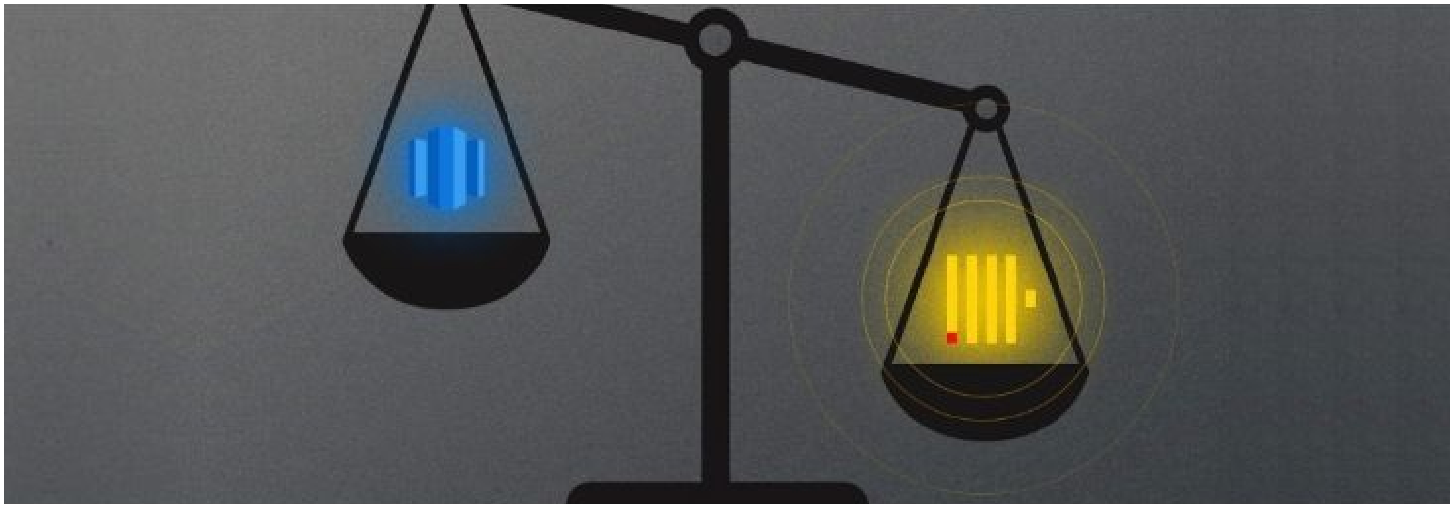
Airflow also has scheduled scripts that connect to the database and periodically upload data from external storage or make event aggregations with operations like

```
INSERT INTO ... SELECT …
```

Redshift has poor availability guarantees. AWS can stop a cluster for updates or scheduled maintenance once a week for up to half an hour (the time slot is specified in settings). When one node goes offline, the cluster also becomes unavailable until the host comes back online. This usually takes about 15 minutes and happens approximately twice a year. It isn't a problem for the current system, as we knew from the start the database would be periodically unavailable.

We used 4 ds2.8xlarge instances (36 CPU, 16 ТБ HDD) for Redshift for 64 TB of total disk space.

The final issue is backup. The backup schedule can be set in the cluster settings, and it runs very smoothly.



REASONS TO MOVE TO CLICKHOUSE

Of course, nobody would ever think about migration to ClickHouse if they never had any issues. And we're no exception.

When we compare the storage schemas in ClickHouse with the MergeTree engine and Redshift, we find their ideology is quite similar. Both databases are column-based, they excel at handling a large number of columns and are very good at compressing data on disks (in Redshift, you can even configure the compression types for each specific column). Data are even stored identically: they are sorted by the primary key, so only specific blocks are read and there is no need to keep specific indices in memory, which is crucial when working with big data.

As usual, the devil is in the details.

Data are sorted on the disk and deleted in Redshift when you run the following command:

```
VACUUM <tablename>
```

One day, one table

The vacuum process works with all data in the table. But when data for all three months is in a single table, the process requires an outrageous amount of time. However, we needed to run it at least once per day because we had old data deleted and new data added. We had to create a special table for every day and consolidate them with views, which not only complicated the view rotation and support, but also slowed down query execution. The “explain” operator showed that all tables were scanned upon query execution, and even though scanning one table takes less than a second, every query takes at least a full minute to execute when we have 90 such tables. This is far from convenient.

Duplicates

The second issue was duplicates. Whenever data are transferred online, two things can happen: data are lost, or duplicates are created. We could not afford losing messages, so we just came to terms with the fact that a small fraction of events will be duplicated. Daily duplicates can be deleted by creating a new table, inserting data from the old table into it with rows with identical IDs deleted by a window function, deleting the old table, and renaming the new one. We needed to keep in mind that the view derived from the daily tables and we had to delete it when the tables were being renamed. We also had to keep an eye on locks, otherwise we could receive a query that would lock the view or one of the tables, which would extend the process for a long time.

Monitoring and maintenance

There are no queries in Redshift that take less than a couple of seconds. Even if you just want to add a user or browse the list of active queries, you need to wait for a couple dozen seconds. Of course, we got used to waiting, and a delay of this length is acceptable for this class of databases, but eventually we started losing too much time.

Cost

According to our calculations, deploying ClickHouse on AWS instances with the same resources was exactly half as expensive. But that makes sense: Redshift is an out-of-the-box database. You just click a few times in the AWS console, connect to it with any PostgreSQL client, and AWS does the rest for you. But was it really worth the money? We already had the infrastructure and presumably knew how to back up, monitor and configure a large number of our internal services. So why not support ClickHouse?

MIGRATION

At the beginning, we deployed a small ClickHouse installation: just a single machine. We used built-in tools to regularly import data from S3. As a result, we were able to test our assumptions about ClickHouse performance and capabilities.

After spending a couple weeks testing a small data copy, we understood that some issues needed to be solved before fully replacing Redshift with Clickhouse:

- What types of instances and disks should we use for deployment?
- Do we need replication?
- How do we install, configure and launch it?
- How do we monitor?
- What schema should we use?
- How do we send data from S3?
- How do we rewrite all queries from standard to non-standard SQL?

INSTANCES AND DISK TYPES. We decided to use the current installation of Redshift as a reference to determine the number of CPUs, disks and amount of memory we needed. We had several options, including i3 instances with local NVMe disks, but in the end we opted for r5.4xlarge and an 8T ST1 EBS storage for each instance. According to our estimates, this would give us performance comparable with Redshift at half the price. With EBS disks, we get simple backup and recovery via disk snapshots, which is almost the same as in Redshift.

REPLICATION. As we were using our existing Redshift configuration as a reference, we decided not to use replication. Another benefit of this is that we did not need to learn ZooKeeper, a service we don't have in our infrastructure yet (but it's great we can now perform replication when we need to).

INSTALLATION. This is the simplest part. Just a small Ansible role that installs ready-to-use RPM packages and makes the same configuration on every host is enough.

MONITORING. We use Prometheus with Telegraph and Grafana to monitor all our services. So we just installed the Telegraph agents to our ClickHouse hosts and prepared a Grafana dashboard to show the current workload on server CPUs, memory, and disks. We used a Grafana plugin to display the current queries to the cluster, the status of the import from S3, and other useful things. The result was much better and more informative than the AWS console dashboard, and it worked quicker too!

SCHEMA. One of our main mistakes in Redshift was storing only the main event fields in separate columns while concatenating all other rarely-used fields in one large column named properties. Indeed, we were able to flexibly edit fields at early stages when we had no understanding which events we'd collect and what properties they'd have (plus they might change 5 times a day). But on the other hand, queries to the large properties column were taking more and more time. In ClickHouse, we decided to do everything right from the very start by taking all the columns we could and assigning them the optimal type. As a result, we were left with a table with approximately two thousand columns.

The next stage was selecting the right engine for storage and partitioning.

As for partitioning, we decided not to reinvent the wheel: we just copied our approach from Redshift creating a partition every day, but now all the partitions were stored in a single table, which considerably accelerated queries and simplified maintenance. We chose the ReplacingMergeTree engine for storage, as we can delete duplicates from a specific partition by just running the OPTIMIZE... FINAL command. Moreover, with the daily partitioning model, we work with data for just one day, which in the event of errors or disasters is much quicker than dealing with a month's worth of data.

DELIVERY OF DATA FROM S3 TO CLICKHOUSE. This was one of the longest processes because we were unable to use the built-in ClickHouse tools for uploading. S3 stores data in JSON, so each field has to be extracted via its jsonpath (as we did in Redshift), and sometimes we even had to make transformations: for example, converting the message UUID from a standard form like `DD96C92F-3F4D-44C6-BCD3-E25EB26389E9` into bytes and putting it into FixedString(16).

We wanted to have a special service similar to the COPY command in Redshift. But we were unable to find an out-of-the-box solution, so we had to make it on our own. Describing this solution could be a topic for another article, so to make a long story short, it is an HTTP service deployed on each host with ClickHouse. Any host can be communicated with. In the query parameters, we specify the S3 prefix, from which the files are taken, the jsonpath list to transform from JSON to a set of columns, and the set of transformations for each column. The server receiving the query starts scanning the files from S3 and sets parcing tasks to other hosts. It was important for us to store all rows that could not be imported into a separate ClickHouse table along with an error message. This is very helpful in investigating issues and bugs in the event handling service and in the clients that generate these events. When we deployed the importer directly on the database hosts, we utilized resources that were usually idle because they do not receive complex queries round-the-clock. We also have the option of moving the importer service to standalone hosts if the number of queries increases.

Importing data from external sources was not a challenge for us. We just changed the destination from Redshift to ClickHouse in the scripts we already had.

There was also an option to connect MongoDB as a dictionary instead of doing daily copies. Unfortunately, this did not suit us because a dictionary must be always stored in memory, and the sizes of most MongoDB collections make this impossible. However, we still made use of dictionaries because they are very convenient in connecting GeoIP databases from MaxMind and are very useful in queries. To do this, we use the ip_trie layout and CSV files provided by the service. For example, the configuration of the geoip_asn_blocks_ipv4 dictionary looks like this:

```
<dictionaries>
        <dictionary>
                <name>geoip_asn_blocks_ipv4</name>
                <source>
                        <file>
                                <path>GeoLite2-ASN-Blocks-IPv4.csv</path>
                                <format>CSVWithNames</format>
                        </file>
                </source>
                <lifetime>300</lifetime>
                <layout>
                        <ip_trie />
                </layout>
                <structure>
                        <key>
                                <attribute>
                                        <name>prefix</name>
                                        <type>String</type>
                                </attribute>
                        </key>
                        <attribute>
                                <name>autonomous_system_number</name>
                                <type>UInt32</type>
                                <null_value>0</null_value>
                        </attribute>
                        <attribute>
                                <name>autonomous_system_organization</name>
                                <type>String</type>
                                <null_value>?</null_value>
                        </attribute>
                </structure>
        </dictionary>
</dictionaries>
```

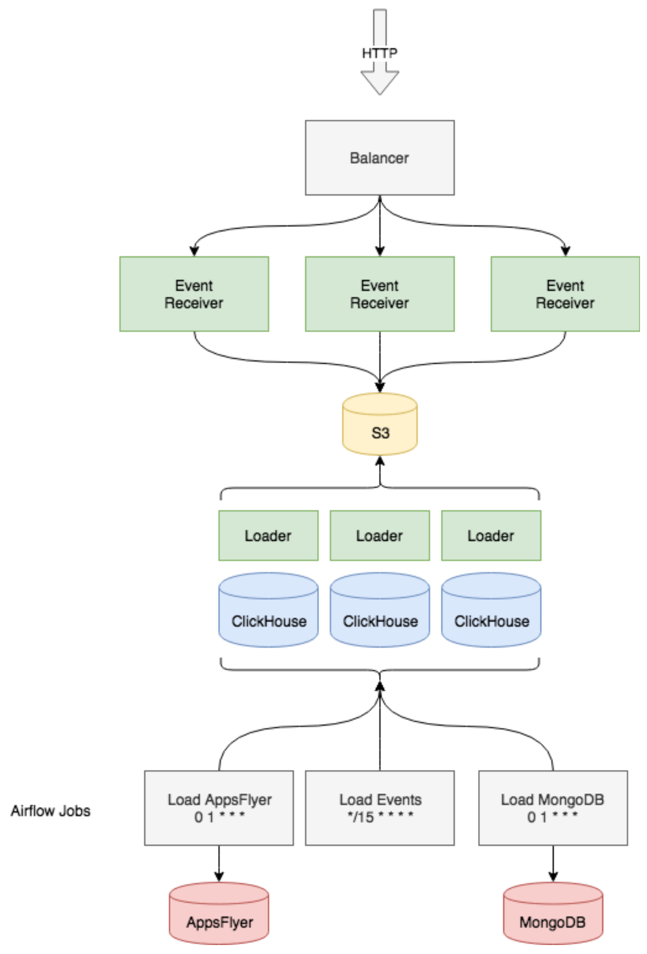
Just put this config at `/etc/clickhouse-server/geoip_asn_blocks_ipv4_dictionary.xml`, and you can make requests to the dictionary to get a provider name by IP address:

```
SELECT dictGetString('geoip_asn_blocks_ipv4', 'autonomous_system_organization',
tuple(IPv4StringToNum('192.168.1.1')));
```

Changing the data schema. As previously mentioned, we decided to abstain from replication because we can afford unavailability in the event of a disaster or scheduled maintenance, and a data copy already stored in S3 can be moved to ClickHouse in a reasonable amount of time. We do not need to deploy ZooKeeper if we don't use replication, but the absence of ZooKeeper makes it impossible to use the ON CLUSTER expression in DDL queries. We solved this issue with a small Python script that connects to any ClickHouse host (there are just 8 of them now) and executes the specified SQL query.

Incomplete SQL support in ClickHouse. We were converting queries from the Redshift to ClickHouse syntax at the same time we were developing the importer, and this task was mostly accomplished by an analyst team. It might seem strange, but the case was not in JOIN, but in window functions. It took us several days to understand how to implement them with arrays and lambda functions. We were lucky this issue is often addressed in the numerous articles on ClickHouse, for example, on events.yandex.ru/lib/talks/5420. At the time, our data were recorded in two places at the same time, Redshift and the new ClickHouse database, so we could compare the results when transferring queries. Nonetheless, it was rather difficult to compare performance because we removed one large properties column, and most queries addressed only the columns they actually needed. Of course, performance growth was easy to notice in such cases. As for queries that did not address the properties column, they had either the same or slightly higher performance.

As result, the database schema looked like this:



RESULTS

In the end, we gained the following:

- One table instead of 90
- Execution of service queries in milliseconds
- Cost halved
- Simple deletion of event duplicates

There were also some cons, but we were ready for them:

- We have to recover clusters on our own in the event of disasters
- Schema changes must be carried out on each host separately
- We have to update the system on our own

We can't make a straight comparison about query speed because the data schema changed so much. But many queries sped up simply because less data are read from the disk. Truth be told, we should have made this change in Redshift, but we decided to combine it with our migration to ClickHouse.

It took us three months to prepare and carry out the migration. Two specialists started on the project at the beginning of July and wrapped it up in September. On September 27, we shut down Redshift, and since that time we have been working exclusively on ClickHouse. In other words, just about three months have passed. Not a very long time, but we have not yet encountered a data loss or critical bug that would make the entire cluster go offline. We're looking forward to upgrades to new versions!