ClickHouse over Object Storage

Pavel Kovalenko, Senior Software Engineer, Yandex.Cloud

ClickHouse MergeTree

Important storage internal details

- All data divided into chunks named as “parts”
- Each part contains column data in multiple or one (compact form) files
- Parts are immutable, written once, most files are not modified
- Merges and mutations spawn new parts (MVCC)
- CH is designed to have continuous read/write I/O profile (no random access)
- CH is designed to read only a subset of column data (select * is bad practice)

ClickHouse cluster

Usability issues

- Data is tightly coupled with hosts/shards
- Storage and execution engine is the same thing (works only with POSIX FS)
- Data is limited by local disks capacity
- Scaling is not easy operation
- You can’t just redeploy a node in case of disk failure
- Need to have expertise to maintain stateful deployment*

* Databases and Kubernetes (RU): https://www.youtube.com/watch?v=BnegHj53pW4

ClickHouse cluster

How to improve

- Decouple execution engine from storage (virtual file system)
- Store parts data into a elastic object storage with high availability and strong durability guarantees (S3, HDFS)
- Store metadata (file/directory hierarchy, names, sizes, permissions) into a transactional KV store or RDB (PostgreSQL, Zookeeper, YDB)
- Local disks are used for caching and storing temporary data

ClickHouse over Object Storage

Benefits

- Unlimited capacity
- Data integrity moved to object storage responsibility
- Disk space can be used more efficiently (hot data in disk cache, cold data in object storage)
- No need to have replicas only for HA
- No need to manually transfer data between replicas
- Node can be quickly redeployed from scratch
- Open doors to elasticity and auto-scaling

01

Virtual File System

Virtual File System

How?

- Virtualize all I/O operations with files (file read/write/remove, directory create/iterate, renaming, seeks, hardlinks)
- Disk as abstraction layer
- Integration with existing storage policies
- Compatibility with current behavior (DiskLocal)
- Possibility to various implementations: S3, HDFS, Memory, etc
- Work is already done for MergeTree and *Log engines

S3 Object Storage

Why?

- Yandex has own S3-compatible Object Storage
- A lot of other cloud implementations AWS, GCP, Azure
- A relatively simple API
- Support range queries (seeks)
- C++ integration out of the box (AWS SDK)

Disk S3

How it’s implemented now?

- Metadata storage is local FS yet
- FS layout is preserved. Part’s files hierarchy and naming are same as in local disk storage but files contain only metadata
- Real data is saved to S3 object with random name
- Metadata files contain a list of S3 object names, size of all S3 objects and references count (hardlinks)
- Returns R/W BufferFromFileBase to transparent read/write as to regular files (with append & seek support).
- Append is needed only for Log engines

Disk S3

Metadata file layout

```
1                      # Metadata file version
3   1044               # Number of objects, Total size of objects
44  data/qrlj...zcv
868 data/nvjb...ffk # Object size, Object S3 path
132 data/asit...fet
1                      # References count
```

Similar layout can be represented in KV / RDB metadata storage

Disk S3

How to use?

```xml
<yandex>
    <storage_configuration>
        <disks>
            <s3>
                <type>s3</type>
                <endpoint>https://s3.yandexcloud.net/jokserfn/data/</endpoint>
                <access_key_id>***</access_key_id>
                <secret_access_key>***</secret_access_key>
            </s3>
        </disks>
        <policies>
            <s3>
                <volumes>
                    <main>
                        <disk>s3</disk>
                    </main>
                </volumes>
            </s3>
        </policies>
    </storage_configuration>
</yandex>
```

Disk S3

How to use?

```sql
CREATE TABLE my_table (
  dt DateTime,
  id Int64,
  data String
) ENGINE=MergeTree()
PARTITION BY dt
ORDER BY (dt, id)
SETTINGS storage_policy='s3'
```

Disk S3

How to use?

```
<yandex>
    <storage_configuration>
        <disks>
            <s3>
                <type>s3</type>
                <endpoint>https://s3.yandexcloud.net/jokserfn/data/</endpoint>
                <access_key_id>***</access_key_id>
                <secret_access_key>***</secret_access_key>
            </s3>
   <ssd>
       <type>local</type>
                <path>/data/</path>
   </ssd>
        </disks>
        <policies>
            <s3_cold>
                <volumes>
                    <main>
                        <disk>ssd</disk>
                    </main>
           <external>
                        <disk>s3</disk>
                    </external>
                </volumes>
            </s3_cold>
        </policies>
    </storage_configuration>
</yandex>
```

Disk S3

How to use?

```sql
CREATE TABLE my_table (
  dt DateTime,
  id Int64,
  data String
) ENGINE=MergeTree()
PARTITION BY dt
ORDER BY (dt, id)
TTL dt + INTERVAL 1 MONTH TO DISK 's3‘
SETTINGS storage_policy='s3_cold'
```

02

Performance benchmark

CH over S3 benchmark

Preparation

- Yandex has S3-compatible Object Storage in cloud
- One CH instance (4 CPU, 16 Gb RAM)
- Benchmark against network-hdd (2Tb), linear read throughput ~ 94 MB/sec
- S3 per one connection read/write throughput ~ 55 MB/sec
- Benchmark data is small part of Yandex.Metrica (used in stateful tests)
- Hits (133 columns, 7.3 Gb)
- Visits (181 columns, 2.5 Gb)

CH over S3 benchmark

Insert benchmark

- Part compact form is used to have less files (setting min_bytes_for_wide_part)

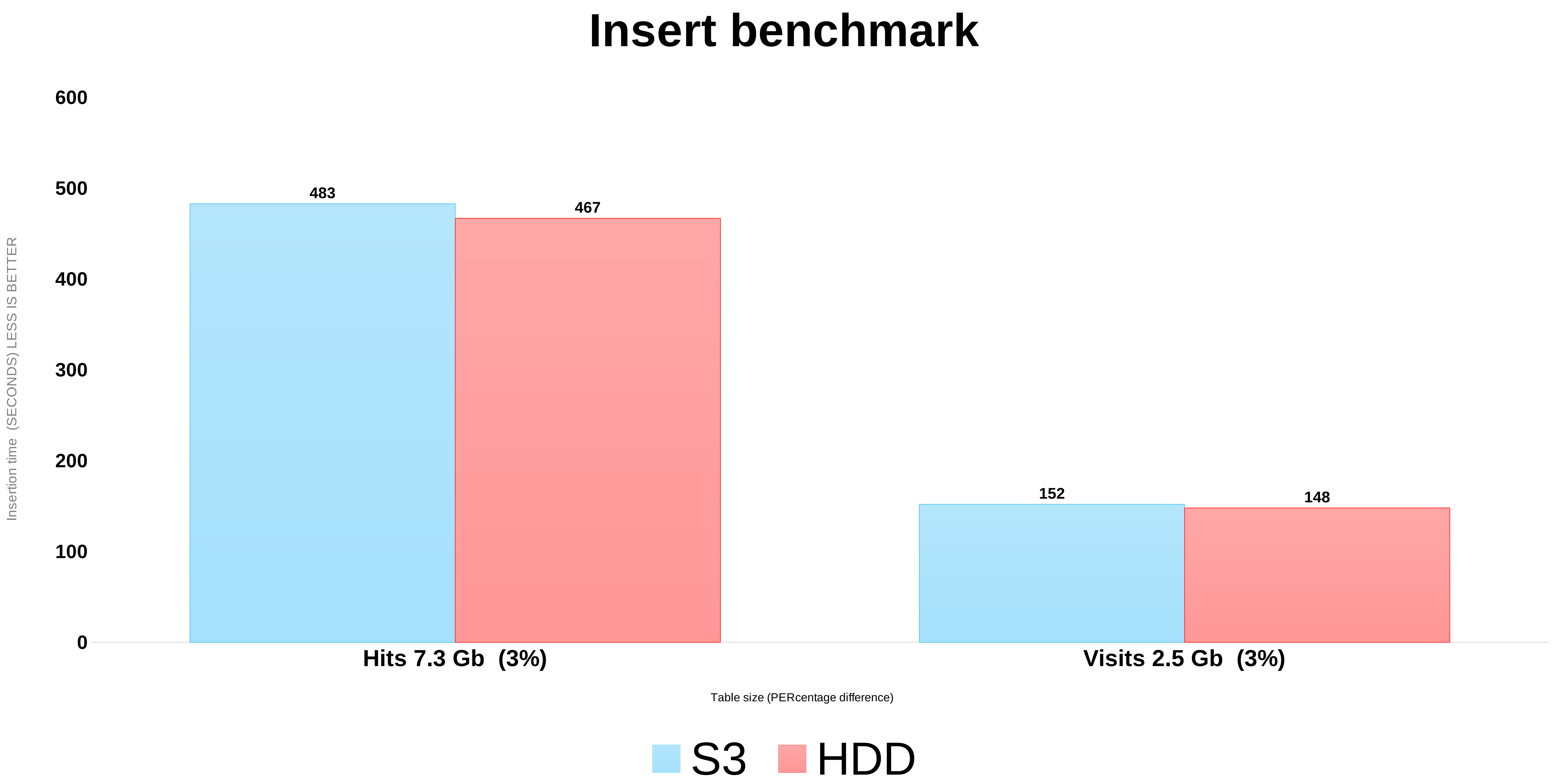
```
time (cat hits_v1.tsv | clickhouse-client --query "INSERT INTO hits_v1 FORMAT TSV")

time (cat visits_v1.tsv | clickhouse-client --query "INSERT INTO visits_v1 FORMAT TSV")
```

Select benchmark

- **OPTIMIZE FINAL** is performed on all tables before run selects
- Page cache is dropped before each query execution
- Query performed several times, best result is used

CH over S3 benchmark



CH over S3 benchmark

Select queries

#1
```sql
SELECT
    SearchEngineID AS k1,
    AdvEngineID AS k2, count() AS c
FROM local.hits_v1
GROUP BY k1, k2
ORDER BY c DESC, k1, k2
LIMIT 10
```

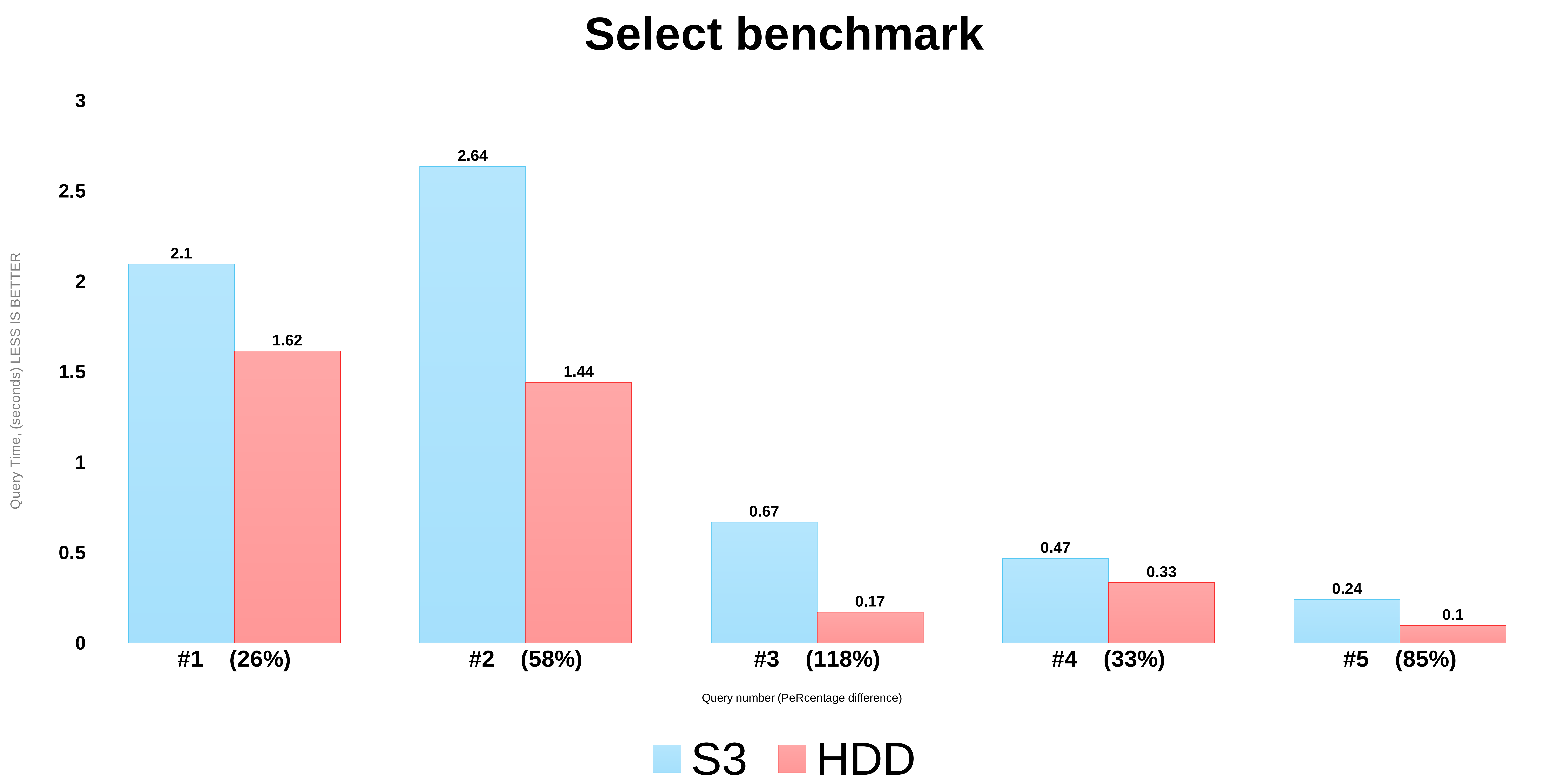
#2
```sql
SELECT EventDate, count() AS hits, any(visits)
FROM local.hits_v1 ANY LEFT JOIN
(
    SELECT
        StartDate AS EventDate,
        sum(Sign) AS visits
    FROM local.visits_v1
    GROUP BY EventDate
) USING EventDate
GROUP BY EventDate
ORDER BY hits DESC
LIMIT 10
```

#3
```sql
SELECT
    StartDate,
    TraficSourceID IN (0) ? 'type_in' : 'other' AS traf_type,
    sum(Sign)
FROM local.visits_v1
WHERE CounterID = 842440
GROUP BY StartDate, traf_type ORDER BY StartDate, traf_type
```

#4
```sql
SELECT CounterID, count() AS c
FROM local.hits_v1
GROUP BY CounterID
ORDER BY c DESC
LIMIT 10
```

#5
```sql
SELECT count()
FROM local.hits_v1
WHERE AdvEngineID != 0
```

CH over S3 benchmark



CH over S3 benchmark

Results and discovered issues

- Overall drop without any optimizations is 20-120%
- S3 has high latency 100-200ms even on small requests
- S3 insertion/selection times linearly depended on the number of files
- Page cache is not working for S3. Marks cache improves latency.
- Seek works not optimally (download all file with specified range instead of chunks)
- Best I/O scenario for S3 is consecutive scan of large files
- Caching and writing files to S3 in parallel should really help

03

Future plans

Future plans

Shared metadata storage

- Transactional engine to perform consistent changes in data parts
- First write to object storage then commit metadata
- GC objects in case of failures
- Reference counters for hard links implementation and sharing parts between replicas
- PostgreSQL or Zookeeper as choice

Future plans

Disk caching

- Store parts content on local disks for better latency
- Mark and index files should be cached first
- Strong consistency (client receives ack if part is uploaded to object storage)
- Eventual consistency (first write to cache then asynchronously replicate to object storage)
- Read-ahead caching (load some files to cache in background ahead of time)

Future plans

Virtual sharding

- Divide all data onto logical partitions (range or key based)
- Distribute ownership of partitions across nodes (consistent / rendezvous hashing)
- During cluster changes re-distribute ownership between nodes
- No data shuffling is needed (all data is already in object storage)
- Possibility to scale on-the-fly if load is too heavy
- Get rid of Distributed/Replicated tables

04

What about other databases?

ClickHouse rivals

Druid & Pinot

- “Deep storage” concept
- Data backup and transferring between cluster nodes
- Prefetch only (Druid), VFS layer + Disk caching (Pinot)
- Data storage engines: S3, HDFS, Azure, GCP
- Metadata storage engines: PostgreSQL, MySQL, Zookeeper

ClickHouse rivals

Impala

- Stateless execution engine
- Main data storage is HDFS (support S3 as well)
- HDFS caching for acceleration (keeping HDFS blocks in memory on data nodes)
- Metadata storage engines: PostgreSQL and MySQL

Kudu

- No HDFS or S3 integration. Storage oriented system
- Tight integration with Impala

ClickHouse rivals

Snowflake

- SaaS solution
- Data storage is S3
- Query execution is decoupled with storage (virtual warehouse)
- Using disk caching
- Metadata in transactional K/V

Design document: http://pages.cs.wisc.edu/~remzi/Classes/739/Spring2004/Papers/p215-dageville-snowflake.pdf

05

Conclusions

ClickHouse over Object Storage

Conclusions

- Acceptable performance even without any optimizations
- More efficient work with S3 can improve throughput
- Disk caching can significantly improve latency
- Many databases already use S3 as main storage, it’s time for ClickHouse to catch up
- S3 is not only option. HDFS, GCP, Azure can be used as well
- Reduces costs of maintenance
- Elastic deployment

Q&A

Yandex Cloud

Thank you!

Pavel Kovalenko

Senior Software Engineer, Yandex.Cloud

jokserfn@gmail.com

@jokserfn