Read from object storage 100 times faster

Kseniia Sumarokova, ClickHouse Core SWE

ClickHouse: Compute and Storage

Compute – blazing fast.

Storage – depends...

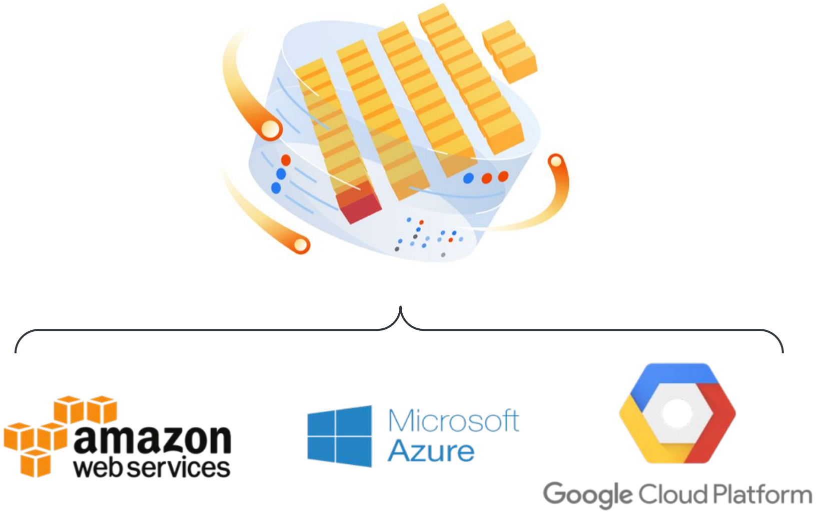
- Local SSD, HDD, NBS
- HTTP server with static files
- Object storages:
    - AWS S3
    - MinIO
    - Google Cloud Storage
    - R2 (Cloudflare)
    - Azure Blob Storage

ClickHouse: Compute and Storage

Compute – blazing fast.

Storage – depends...

- Local SSD, HDD, NBS
- Remote HTTP server
- **Object storages:**
  - AWS S3
  - MinIO
  - Google Cloud Storage
  - R2 (Cloudflare)
  - Azure Blob Storage



What is Object Storage?

- A distributed storage
- Usually provides HTTP API
- Stores blobs with some path (URL)
- Blobs are immutable
- Storage has flat structure
- Very different from traditional filesystems:
  - No seeks
  - No hardlinks
  - No appends
  - etc

Object Storage: Pros and Cons

+ Infinite data storage

+ Highly reliable

+ Scales automatically

+ Storage is cheap

+ High throughput

– Big latency

– Requests (read, writes) are not free

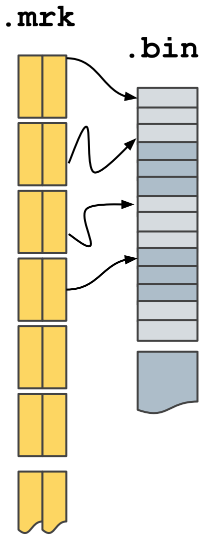
– API is limited compared to FS

Towards Object Storage support in MergeTree engine

- ✓ Design FS abstraction
- ✓ Make MergeTree work with this abstraction
- ✓ Implement the abstraction for Object Storages
- ? Optimize… as much as possible
  - ○ Use benefits of the API
  - ○ Utilize high throughput
  - ○ Reduce high latency

Reading data in MergeTree: the basics

- Column oriented storage
- Index is sparse (fits in memory)
- Columns are compressed
- Marks allow to read compressed data according to index
- Marks are located in .mrk file for each column
- Data is located in .bin file
- Granule is a range of rows between two marks
- Read data by granules
- Read different granules in parallel



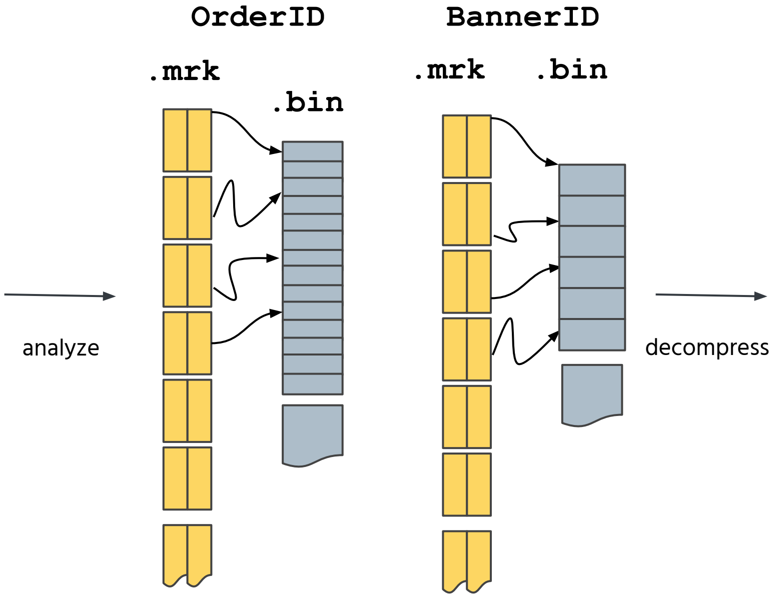
ClickHouse

Reading data in MergeTree

primary.idx

OrderID	BannerID
0	10000
8192	18192
16384	26384
1998848	2008848

Sparse index



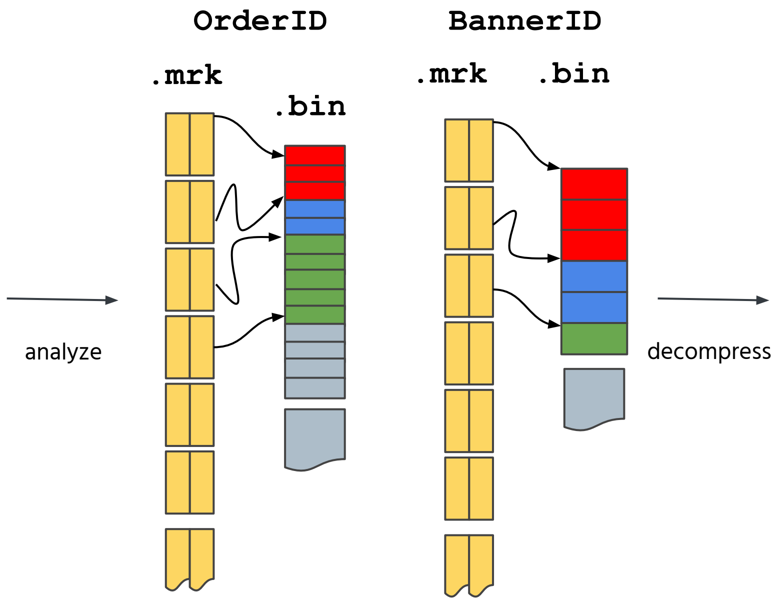
OrderID	BannerID
3861813205	n#O=n)SNW%-(?c|G%[hN
1356609667	ui5vptJlh+mKS]1}elL&
3265981873	[D/,i5G\\6F5hP=i]<q|
2267935489	Hk.Qo78?jM9,,\-M,Rj^
1150892606	g&(]A&@2U/b-|BU,c;d%
2836790991	lfGmuuauNz"HeOjH1unN
2749022404	7AZ&5LY`X\ydCAhT9lJh
2393185453	\T`6R-?q}:6CtQJ2f9rS
1211724375	`=6P&_ieD^|1:|<?s?(M
3109046926	Y&B1I\N'@(A.4/a"Y'Ia

Reading data in MergeTree: in Parallel

primary.idx

OrderID	BannerID
0	10000
8192	18192
16384	26384
1998848	2008848

Sparse index

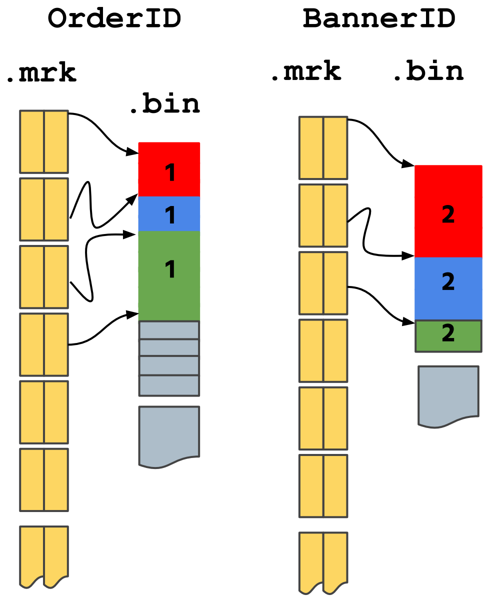


OrderID	BannerID
3861813205	n#O=n)SNW%-(?c|G%[hN
1356609667	ui5vptJlh+mKS]1}elL&
3265981873	[D/,i5G\\6F5hP=i]<q|
2267935489	Hk.Qo78?jM9,,\-M,Rj^
1150892606	g&(]A&@2U/b-|BU,c;d%
2836790991	lfGmuuauNz"HeOjH1unN
2749022404	7AZ&5LY`X\ydCAhT9lJh
2393185453	\T`6R-?q}:6CtQJ2f9rS
1211724375	`=6P&_ieD^|1:|<?s?(M
3109046926	Y&B1I\N'@(A.4/a"Y'Ia

ClickHouse

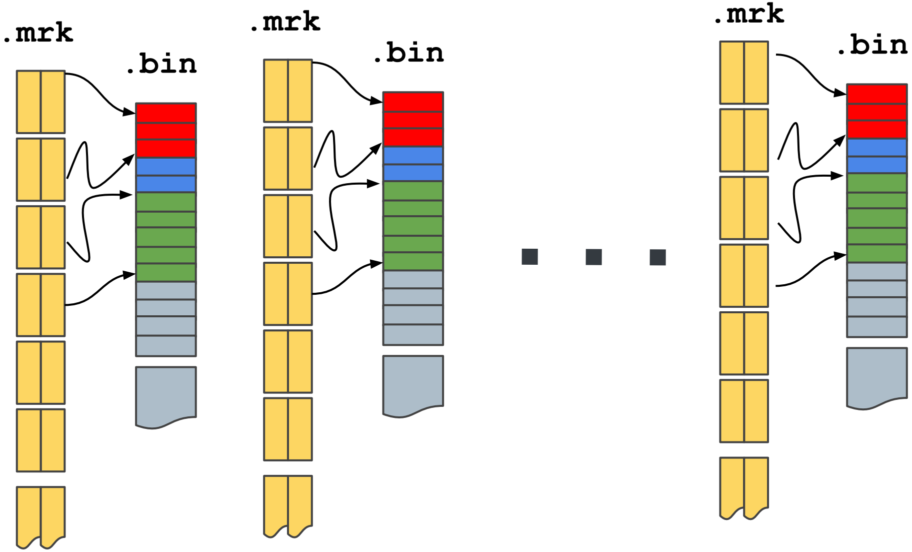
Reading data in MergeTree

- Each thread reads a non-intersecting set of granule(s).
- Each granule (in the picture: red, blue or green) is a range of rows between two marks and includes multiple columns
- Columns in a granule are read **one by one**



What if the number of columns is huge?

- Huge number of columns + big latency results in too slow query execution



Optimization #1: Prefetch all the columns

- In each query thread for each column in granules make asynchronous request to read the data
- Optimization #2: Using Object Storage range requests if supported

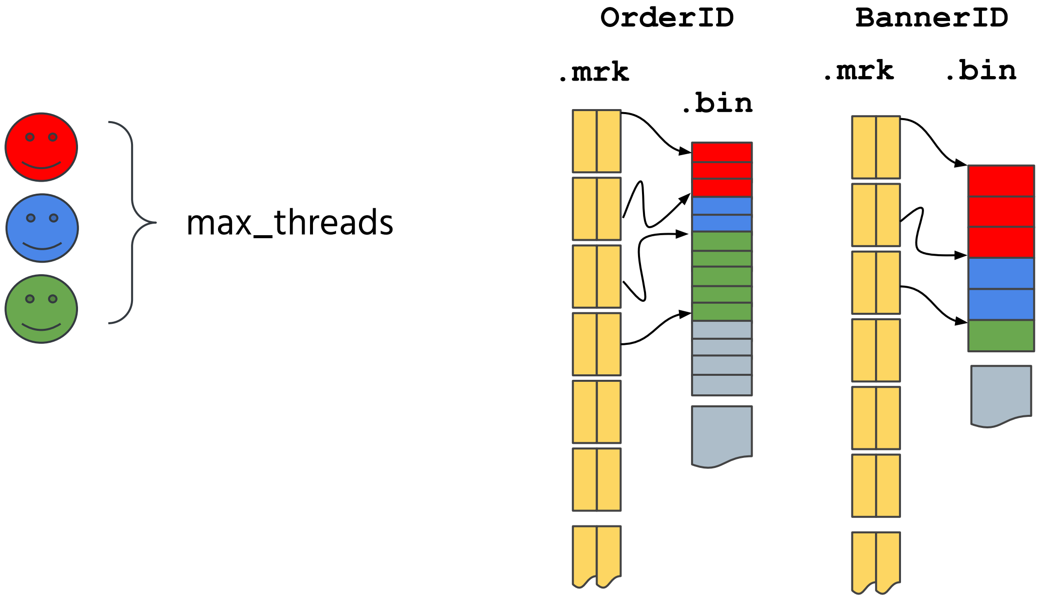
What can be improved:

- Marks are still loaded synchronously
- Only max_threads workers can read the granules and prefetch the data

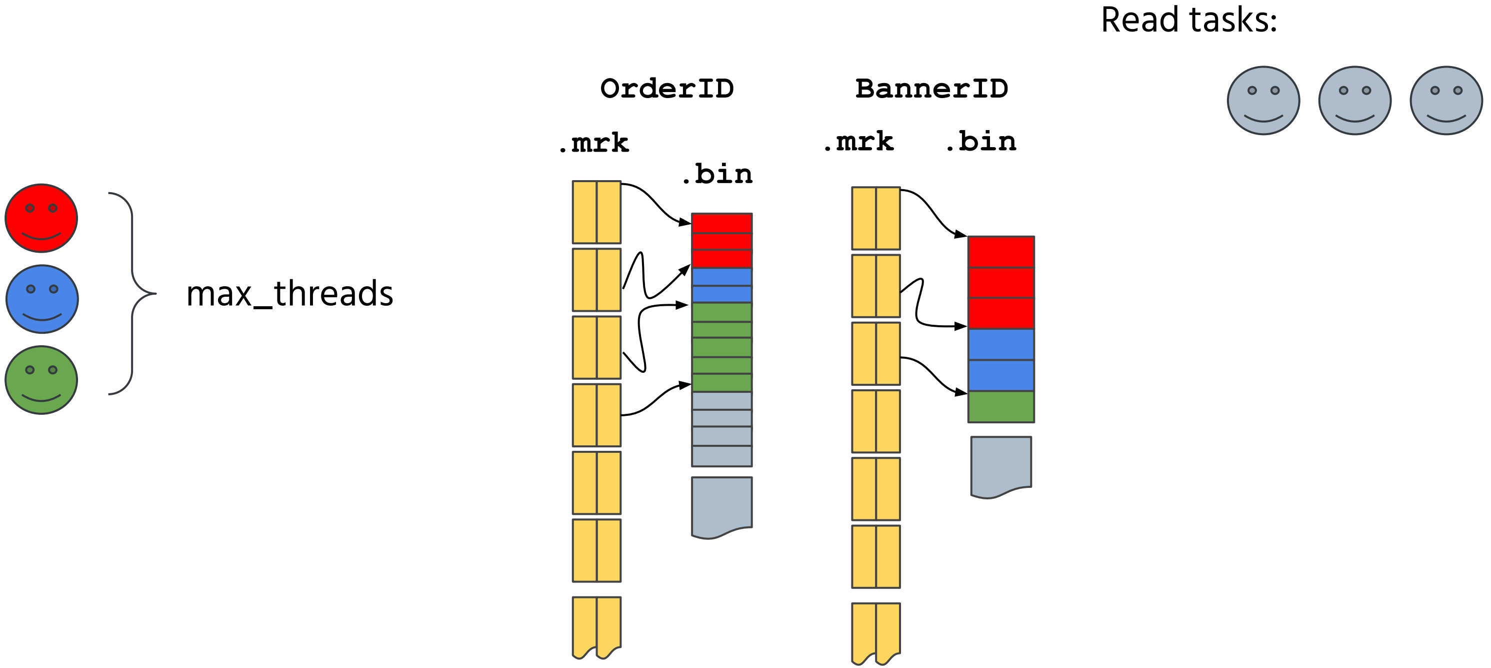
Optimization #3: Asynchronous Loading of Marks

- Mark cache already exists but size is limited
- Preload all marks for query into cache on query start
- Use separate pool for marks preload

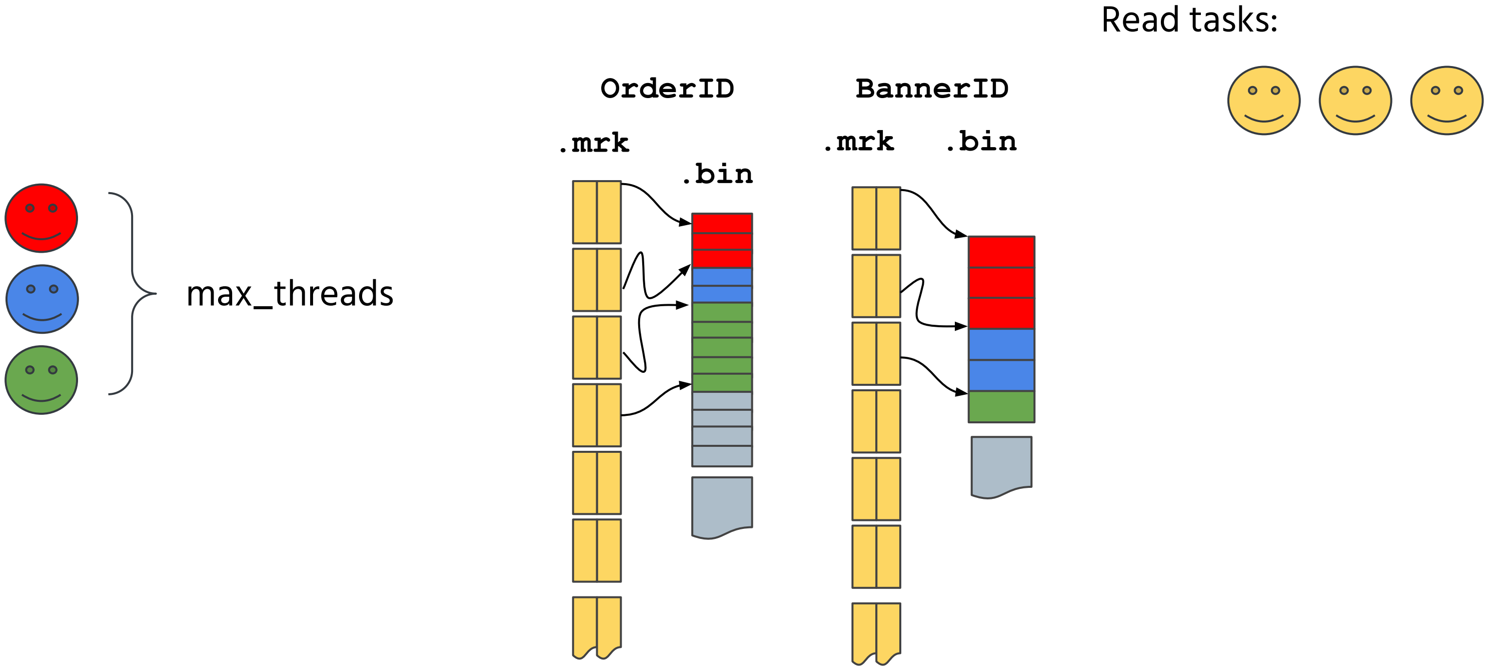
Dealing with limited number of query threads



Dealing with limited number of query threads

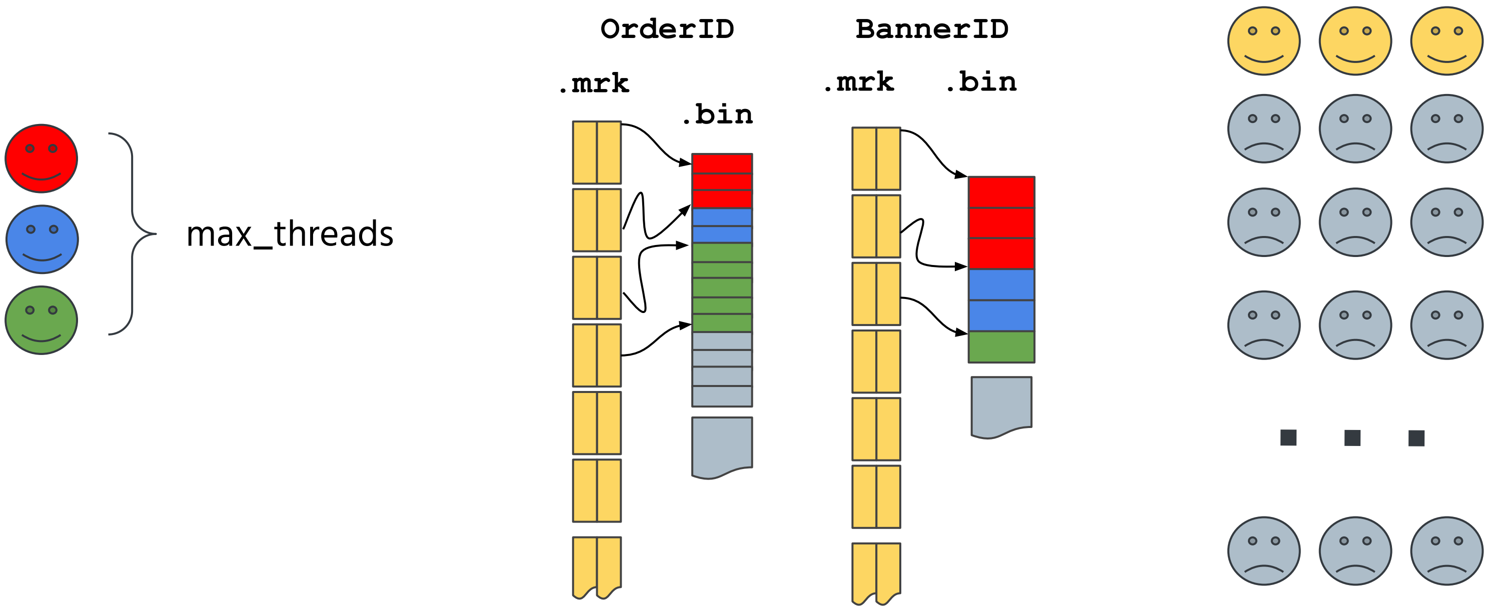


Dealing with limited number of query threads



Dealing with limited number of query threads

Read tasks:



max_threads

OrderID

.mrk

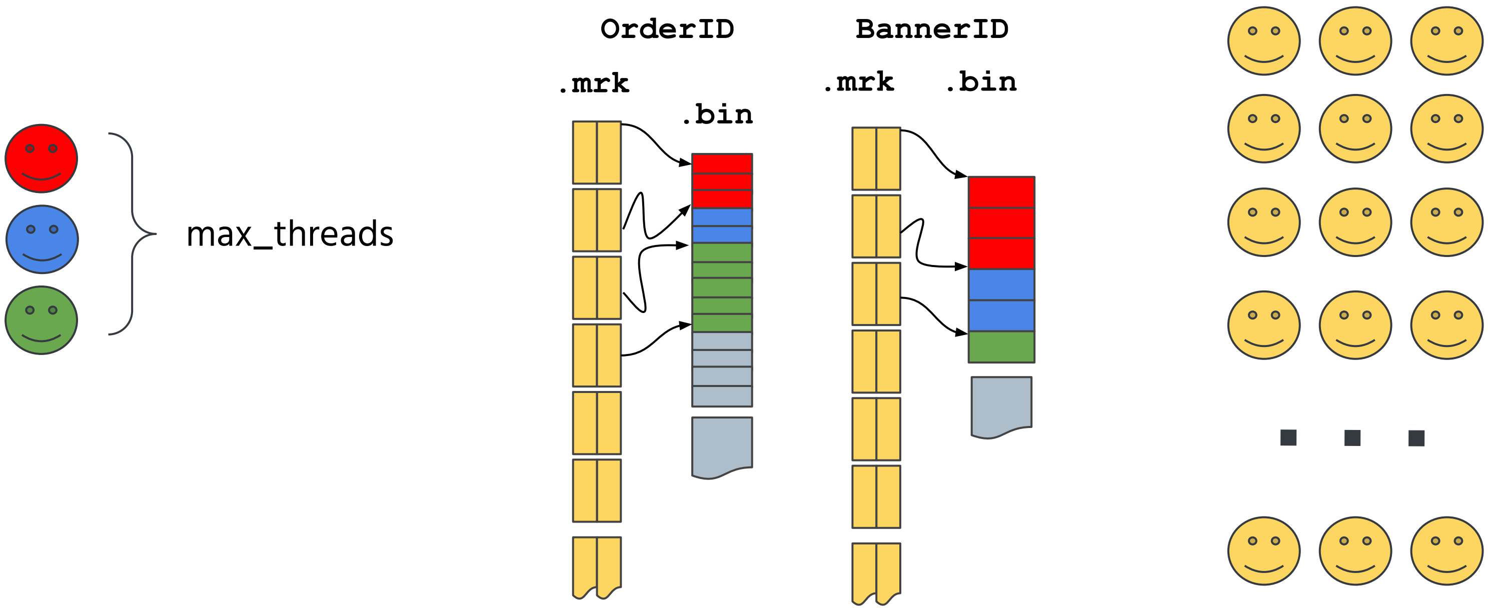
.bin

BannerID

.mrk

.bin

Start prefetching for all read tasks



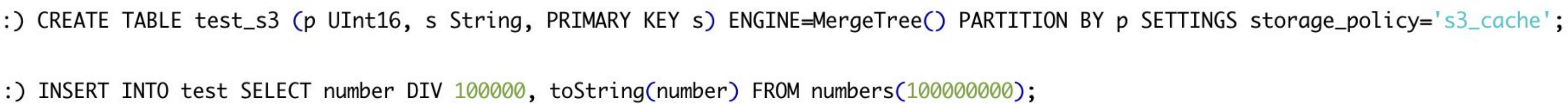
Optimization #4: Prefetch all the tasks

- Read task consists of:
  - Part to read
  - Marks ranges within part to read
- Can prefetch them independently from query threads:
  - Start asynchronously with the start of query
  - Use single threadpool per server with a fixed number of threads
  - Additionally limit the number of concurrent prefetches (avoid OOM)

Optimization #5: Caching on local disk

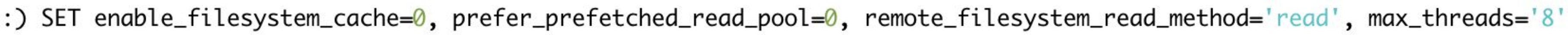
- Cache has limited size
- Cache consists of file segments - limited sized segments of files
- Removal from cache follows eviction policy
- Caching on reads, writes, background merges
- Files are immutable, so cached file segments validation is not required.
- On file deletion, corresponding cache files are also removed.

Performance comparison

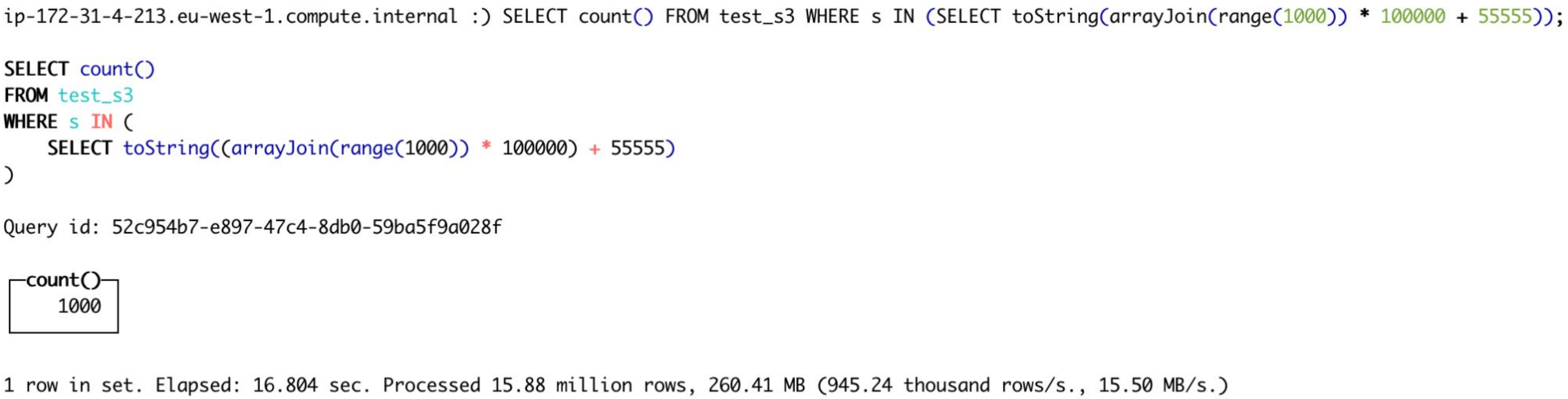


```
:) CREATE TABLE test_s3 (p UInt16, s String, PRIMARY KEY s) ENGINE=MergeTree() PARTITION BY p SETTINGS storage_policy='s3_cache';

:) INSERT INTO test SELECT number DIV 100000, toString(number) FROM numbers(100000000);
```

Performance comparison



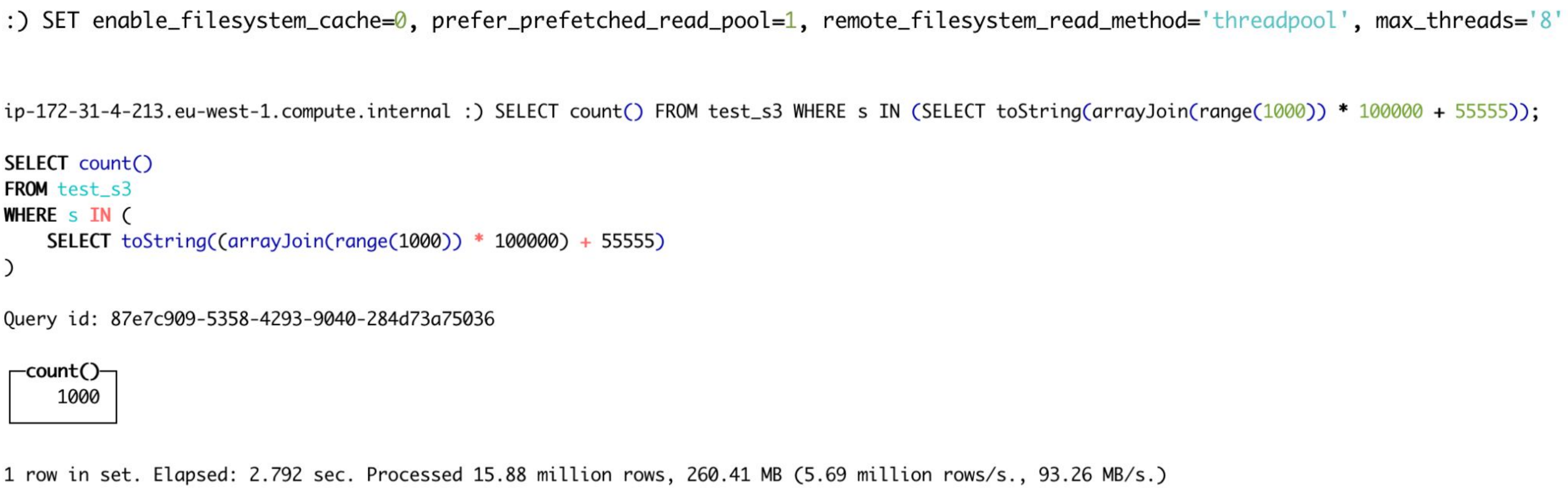
```
:) SET enable_filesystem_cache=0, prefer_prefetched_read_pool=0, remote_filesystem_read_method='read', max_threads='8'
```



```
ip-172-31-4-213.eu-west-1.compute.internal :) SELECT count() FROM test_s3 WHERE s IN (SELECT toString(arrayJoin(range(1000)) * 100000 + 55555));

SELECT count()
FROM test_s3
WHERE s IN (
    SELECT toString((arrayJoin(range(1000)) * 100000) + 55555)
)

Query id: 52c954b7-e897-47c4-8db0-59ba5f9a028f

┌─count()─┐
│    1000 │
└─────────┘

1 row in set. Elapsed: 16.804 sec. Processed 15.88 million rows, 260.41 MB (945.24 thousand rows/s., 15.50 MB/s.)
```

ClickHouse

Performance comparison



```
:) SET enable_filesystem_cache=0, prefer_prefetched_read_pool=1, remote_filesystem_read_method='threadpool', max_threads='8'


ip-172-31-4-213.eu-west-1.compute.internal :) SELECT count() FROM test_s3 WHERE s IN (SELECT toString(arrayJoin(range(1000)) * 100000 + 55555));

SELECT count()
FROM test_s3
WHERE s IN (
    SELECT toString((arrayJoin(range(1000)) * 100000) + 55555)
)

Query id: 87e7c909-5358-4293-9040-284d73a75036

┌count()─┐
│   1000 │
└────────┘

1 row in set. Elapsed: 2.792 sec. Processed 15.88 million rows, 260.41 MB (5.69 million rows/s., 93.26 MB/s.)
```

Thanks for attention!

Questions?