ClickHouse 資料庫引擎源碼

About Me

Pomin Wu

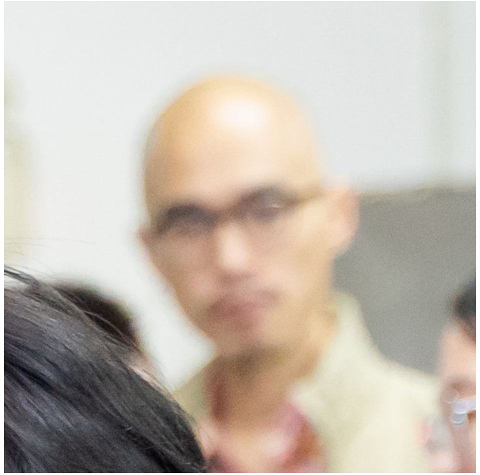
pm5@g0v.social

喜歡算數學、寫程式、練習人造語言。

覺得宅太久的時候會去攀岩。

參與 g0v 零時政府等公民黑客社群，是 g0v 國際交流小組共同發起人。

目前於質子科技擔任資深工程師。



We are hiring

Taipei Office

- System Administrator
- System & Reliability Engineer
- Spam and Abuse Analyst
- Front-End / Back-End / Full-stack
- Machine Learning Engineer
- Technical Support Specialist

Proton

Proton Mail

Proton Calendar

Proton Drive

Proton VPN

Proton Bridge

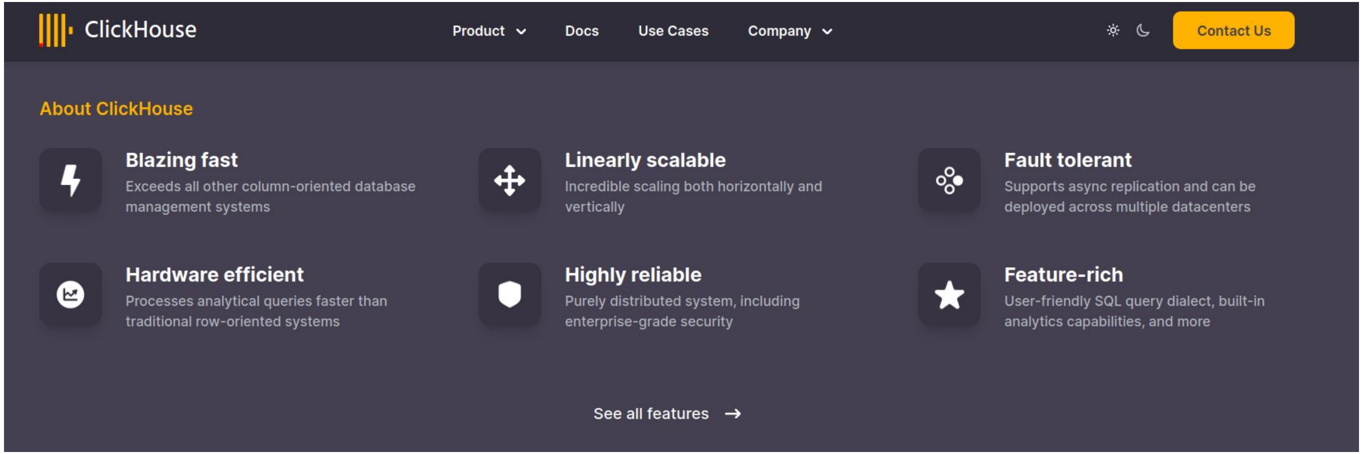
ClickHouse

https://clickhouse.com/

開源 OLAP 資料庫

Column-oriented

還蠻快



ClickHouse

一個中等大小的例子

- data_table 保存一些 log 紀錄
- timestamp 是 log 的時間(精確到秒)
- 整個 data_table 大約 2.6TB
- timestamp 沒有建 index

好處

- 不用建 index。
- 改變查詢條件，對查詢速度影響相對小。

```
SELECT
  toStartOfMinute(timestamp) AS ts,
  count(*)
FROM data_table
WHERE timestamp BETWEEN ...
GROUP BY ts
```

```
... rows in set. Elapsed: 1.684
sec. Processed 9.59 billion rows,
38.37 GB (5.70   billion rows/s.,
22.78 GB/s.)
```

ClickHouse

GitHub - ClickHouse/ClickHouse
https://github.com/ClickHouse/ClickHouse

Overview of ClickHouse Architecture
https://clickhouse.com/docs/en/development/architecture/

GitHub - ClickHouse/clickhouse-presentations
https://github.com/ClickHouse/clickhouse-presentations

Altinity, Inc. - YouTube
https://www.youtube.com/c/AltinityLtd

Storages

```
$ tree -P '*.h' -L 1 src/Storages/
src/Storages/
├── ...
├── IStorage.h
├── ...
├── StorageJoin.h
├── StorageLog.h
├── ...
├── StorageMaterializedMySQL.h
├── StorageMaterializedView.h
├── StorageMemory.h
├── ...
├── StorageMongoDB.h
├── ...
├── StorageMySQL.h
├── StorageNull.h
├── StoragePostgreSQL.h
├── ...
├── StorageS3.h
├── ...
```

IStorage

Storage engines in ClickHouse has an interface **IStorage** defined in **`src/Storages/IStorage.h`**.

The most important methods are **`read`** and **`write`**.

`write` returns a **`SinkToStoragePtr`**, which points to a **`SinkToStorage`**.

Storages extend **`SinkToStorage`** to implement their own sink to handle writing data.

More on “what’s a sink” later.

```
// src/Storages/IStorage.h
   virtual void read(
       QueryPlan & query_plan,
       const Names & /*column_names*/,
       const StorageSnapshotPtr & /*storage_snapshot*/,
       SelectQueryInfo & /*query_info*/,
       ContextPtr /*context*/,
       QueryProcessingStage::Enum /*processed_stage*/,
       size_t /*max_block_size*/,
       unsigned /*num_streams*/);

   virtual SinkToStoragePtr write(
       const ASTPtr & /*query*/,
       const StorageMetadataPtr & /*metadata_snapshot*/,
       ContextPtr /*context*/)
   {
       throw Exception("Method write is not supported by
storage " + getName(), ErrorCodes::NOT_IMPLEMENTED);
   }
```

IStorage

(Public) **`read`** calls (private) **`read`** to create a **`Pipe`**, which is used to create a **`ReadFromStorageStep`**, which is then added to **`query_plan`**.

```
// src/Storages/IStorage.cpp
void IStorage::read(QueryPlan & query_plan, const Names & column_names,
const StorageSnapshotPtr & storage_snapshot, SelectQueryInfo &
query_info, ContextPtr context, QueryProcessingStage::Enum
processed_stage, size_t max_block_size, unsigned num_streams)
{
    auto pipe = read(column_names, storage_snapshot, query_info, context,
processed_stage, max_block_size, num_streams);
    readFromPipe(query_plan, std::move(pipe), column_names,
storage_snapshot, query_info, context, getName());
}

void IStorage::readFromPipe(QueryPlan & query_plan, Pipe pipe, const
Names & column_names, const StorageSnapshotPtr & storage_snapshot,
SelectQueryInfo & query_info, ContextPtr context, std::string
storage_name)
{
    // ...
    {
        auto read_step =
std::make_unique<ReadFromStorageStep>(std::move(pipe), storage_name,
query_info.storage_limits);
        query_plan.addStep(std::move(read_step));
    }
}
```

IStorage

One of the interesting things about ClickHouse is that it supports many kinds of storage engine.

Merge tree family is the most common ones, but also **`StorageMemory`**, **`StorageFile`**, **`StorageMySQL`**, **`StoragePostgreSQL`**, **`StorageSQLite`**, **`StorageDistributed`**, etc.

This probably shows that ClickHouse has a lot of optimizations that are applicable regardless of the underlying storage engine implementation.

StorageMemory

(Now public) **StorageMemory::read** produces a **Pipe**. This **Pipe** is composed of **num_streams** of **MemorySource**.

IStorage::write returns a point to an **SinkToStorage**. The one returned from **StorageMemory::write** is **MemorySink**.

MemorySource and **MemorySink** are the places where data read and write actually happens.

```
// src/Storages/StorageMemory.cpp
Pipe StorageMemory::read(const Names & column_names, const StorageSnapshotPtr &
storage_snapshot, SelectQueryInfo & /*query_info*/, ContextPtr /*context*/,
QueryProcessingStage::Enum /*processed_stage*/, size_t /*max_block_size*/,
unsigned num_streams)
{
    // ...
    size_t size = current_data->size();

    if (num_streams > size)
        num_streams = size;

    Pipes pipes;

    auto parallel_execution_index = std::make_shared<std::atomic<size_t>>(0);

    for (size_t stream = 0; stream < num_streams; ++stream)
    {
        pipes.emplace_back(std::make_shared<MemorySource>(column_names,
storage_snapshot, current_data, parallel_execution_index));
    }

    return Pipe::unitePipes(std::move(pipes));
}

SinkToStoragePtr StorageMemory::write(const ASTPtr & /*query*/, const
StorageMetadataPtr & metadata_snapshot, ContextPtr context)
{
    return std::make_shared<MemorySink>(*this, metadata_snapshot, context);
}
```

StorageMemory

Many of the storages of ClickHouse implement their own sources and sinks.

There are **`MergeTreeSequentialSource`** and **`MergeTreeSink`**, **`MemorySource`** and **`MemorySink`**, **`KafkaSource`** and **`KafkaSink`**, etc.

We will soon explain what sources and sinks are for ClickHouse.

StorageMemory

The main methods of concern in the case of memory storage are **`MemorySource::generate`** and **`MemorySink::consume`**.

`MemorySource::generate` creates a **`Block`** out of the indexed data, get a **`Columns`** out of it, and store that in the **`Chunk`** to be returned.

```
// src/Storages/StorageMemory.cpp
Chunk MemorySource::generate() override
{
    // ...
    const Block & src = (*data)[current_index];

    Columns columns;
    size_t num_columns = column_names_and_types.size();
    columns.reserve(num_columns);

    auto name_and_type = column_names_and_types.begin();
    for (size_t i = 0; i < num_columns; ++i)
    {
        columns.emplace_back(tryGetColumnFromBlock(src,
*name_and_type));
        ++name_and_type;
    }

    fillMissingColumns(columns, src.rows(), column_names_and_types,
/*metadata_snapshot=*/ nullptr);
    assert(std::all_of(columns.begin(), columns.end(), [](const
auto & column) { return column != nullptr; }));

    return Chunk(std::move(columns), src.rows());
}
```

StorageMemory

`MemorySink::consume` stores the **`Chunk`** input in **`new_blocks`**.

```
// src/Storages/StorageMemory.cpp
void MemorySink::consume (Chunk chunk) override
{
    auto block = getHeader().cloneWithColumns(chunk.getColumns());
    storage_snapshot->metadata->check(block, true);
    if (!storage_snapshot->object_columns.empty())
    {
        auto extended_storage_columns = storage_snapshot->getColumns(
GetColumnsOptions(GetColumnsOptions::AllPhysical).withExtendedObjects());

        convertObjectsToTuples(block, extended_storage_columns);
    }

    if (storage.compress)
    {
        Block compressed_block;
        for (const auto & elem : block)
            compressed_block.insert({ elem.column->compress(), elem.type,
elem.name });

        new_blocks.emplace_back(compressed_block);
    }
    else
    {
        new_blocks.emplace_back(block);
    }
}
```

StorageMemory

And in **`MemorySink::onFinish`**, acquires a lock and writes data to **`storage`**.

```
// src/Storages/StorageMemory.cpp
void MemorySink::onFinish() override
{
    size_t inserted_bytes = 0;
    size_t inserted_rows = 0;

    for (const auto & block : new_blocks)
    {
        inserted_bytes += block.allocatedBytes();
        inserted_rows += block.rows();
    }

    std::lock_guard lock(storage.mutex);

    auto new_data =
std::make_unique<Blocks>(*(storage.data.get()));
    new_data->insert(new_data->end(), new_blocks.begin(),
new_blocks.end());

    storage.data.set(std::move(new_data));
    storage.total_size_bytes.fetch_add(inserted_bytes,
std::memory_order_relaxed);
    storage.total_size_rows.fetch_add(inserted_rows,
std::memory_order_relaxed);
}
```

Wrap up

Chunk is the unit of data processing in ClickHouse.

Storage engines implement their sources and sinks, to read and write data in chunks.

Processors

```
$ tree -L 1 -P '*.h'  src/Processors
src/Processors
├── Chunk.h
├── ...
├── Executors
├── ForkProcessor.h
├── Formats
├── IAccumulatingTransform.h
├── ...
├── IProcessor.h
├── ISimpleTransform.h
├── ISink.h
├── ISource.h
├── ...
├── Port.h
├── QueryPlan
├── ...
├── Sinks
├── Sources
├── ...
├── Transforms
```

IProcessor

An **IProcessor** has input and output ports.

Sources and sinks and transforms are all **IProcessor**s.

It can read from input ports, write to output ports, and transform the data with **work**.

prepare is not thread-safe. **work** is thread-safe.

Sources, transforms, and sinks is a common pattern in data flow or stream processing systems.

```
// src/Processorts/IProcessor.h
class IProcessor
{
protected:
    InputPorts inputs;
    OutputPorts outputs;
public:
    IProcessor() = default;

    IProcessor(InputPorts inputs_, OutputPorts outputs_)
        : inputs(std::move(inputs_)), outputs(std::move(outputs_))
    {
        for (auto & port : inputs)
            port.processor = this;
        for (auto & port : outputs)
            port.processor = this;
    }
    virtual Status prepare();
    virtual void work();
    virtual int schedule();
    virtual Processors expandPipeline();
    // ...
};
```

Port

An **`InputPort`** is `connect` to an **`OutputPort`**.

A connected pair of ports act like a shared lock over the shared **`Port::State`** between the ports.

```
// src/Processors/Pipe.h
Chunk InputPort::pull(bool set_not_needed = false);
void OutputPort::push(Chunk chunk);

void connect(OutputPort & output, InputPort & input)
{
    if (input.state)
        throw Exception(ErrorCodes::LOGICAL_ERROR, "Port is already
connected, (header: [{}])", input.header.dumpStructure());

    if (output.state)
        throw Exception(ErrorCodes::LOGICAL_ERROR, "Port is already
connected, (header: [{}])", output.header.dumpStructure());

    auto out_name = output.getProcessor().getName();
    auto in_name = input.getProcessor().getName();

    assertCompatibleHeader(output.getHeader(), input.getHeader(),
fmt::format(" function connect between {} and {}", out_name, in_name));

    input.output_port = &output;
    output.input_port = &input;
    input.state = std::make_shared<Port::State>();
    output.state = input.state;
}
```

Block

Let's take a quick look at **Chunk** and **Block**.

***column** is where the data is actually stored.

***type** is an **IDataType** and all data types are defined in **src/DataTypes**.

So a **Block** is basically a vector of columns all having their data, type, and name.

```
// src/Core/Block.h
class Block
{
private:
    using Container = ColumnsWithTypeAndName;
    using IndexByName = std::unordered_map<String,
size_t>;

    Container data;
    IndexByName index_by_name;
// ...
};

struct ColumnWithTypeAndName
{
    ColumnPtr column;
    DataTypePtr type;
    String name;
    // ...
};
```

Chunk

A **Chunk** is like a **Block** without data type info.

```
// src/Processors/Block.h
class Chunk
{
// ...
private:
    Columns columns;
    UInt64 num_rows = 0;
    ChunkInfoPtr chunk_info;
};

using Columns = std::vector<ColumnPtr>;
```

ISimpleTransform

`ISimpleTransform` are the simplest transforms.

They have one input port, one output port.

All transforms are in **`src/Processors/Transforms/`**.

Examples are **`LimitTransform`** which implements LIMIT, **`ExtremesTransform`** which implements EXTREMES in SQL.

```cpp
// src/Processors/ISimpleTransform.cpp
void ISimpleTransform::work()
{
    if (input_data.exception)
    {
        /// Skip transform in case of exception.
        output_data = std::move(input_data);
        has_input = false;
        has_output = true;
        return;
    }

    try
    {
        transform(input_data.chunk, output_data.chunk);
    }
    catch (DB::Exception &)
    {
        output_data.exception = std::current_exception();
        has_output = true;
        has_input = false;
        return;
    }

    has_input = !needInputData();

    if (!skip_empty_chunks || output_data.chunk)
        has_output = true;

    if (has_output && !output_data.chunk && getOutputPort().getHeader())
        /// Support invariant that chunks must have the same number of columns as header.
        output_data.chunk = Chunk(getOutputPort().getHeader().cloneEmpty().getColumns(),
0);
}
```

Explain pipeline

You can see how your query is translated to transforms with **`EXPLAIN PIPELINE`**.

```
:) EXPLAIN PIPELINE SELECT 1 LIMIT 10

Query id: 1f586ac5-ba6d-4cc8-abc4-9dae2bff6998

┌─explain─────────────────────────────────┐
│ (Expression)
│ ExpressionTransform
│   (SettingQuotaAndLimits)
│     (Limit)
│     Limit
│     (ReadFromStorage)
│     SourceFromSingleChunk 0 → 1
└─────────────────────────────────────────┘
```

ExtremesTransform

Since we are working with **`Chunk`** here, with **`ISimpleTransform::prepare`**, it can actually do a lot more than “map”.

For example, in **`ExtremesTransform`**

1. Adds a new port to store the **`extremes`** upon creation,
2. Calculate extremes (a **`Chunk`**) in **`ExtremesTransform::transform`**, and
3. Use **`ExtremesTransform::prepare`** to push the extremes to the new port.

```
// src/Processors/Transforms/ExtremesTransform.cpp
ExtremesTransform::ExtremesTransform(const Block & header)
    : ISimpleTransform(header, header, true)
{
    /// Port for Extremes.
    outputs.emplace_back(outputs.front().getHeader(), this);
}

IProcessor::Status ExtremesTransform::prepare()
{
    if (!finished_transform)
    {
        auto status = ISimpleTransform::prepare();

        if (status != Status::Finished)
            return status;

        finished_transform = true;
    }

    auto & totals_output = getExtremesPort();

    /// Check can output.
    if (totals_output.isFinished())
        return Status::Finished;

    if (!totals_output.canPush())
        return Status::PortFull;

    if (!extremes && !extremes_columns.empty())
        return Status::Ready;

    if (extremes)
        totals_output.push(std::move(extremes));

    totals_output.finish();
    return Status::Finished;
}
```

ExtremesTransform

For example, **`ExtremesTransform`**

1. Adds a new port to store the **`extremes`** upon creation,
2. Calculate extremes (a **`Chunk`**) in **`ExtremesTransform::transform`**, and
3. Use **`ExtremesTransform::prepare`** to push the extremes to the new port.

```
// src/Processors/Transforms/ExtremesTransform.cpp
void ExtremesTransform ::work()
{
    if (finished_transform )
    {
        if (!extremes  && !extremes_columns .empty())
            extremes .setColumns (std::move(extremes_columns ), 2);
    }
    else
        ISimpleTransform ::work();
}

void ExtremesTransform ::transform (DB::Chunk & chunk)
{
    // ...
    {
        for (size_t i = 0; i < num_columns; ++ i)
        {
            if (isColumnConst (*extremes_columns [i]))
                continue ;

            Field min_value  = (*extremes_columns [i])[0];
            Field max_value  = (*extremes_columns [i])[1];

            Field cur_min_value ;
            Field cur_max_value ;

            columns [i]->getExtremes (cur_min_value , cur_max_value );

            if (cur_min_value  < min_value )
                min_value  = cur_min_value ;
            if (cur_max_value  > max_value )
                max_value  = cur_max_value ;

            MutableColumnPtr  new_extremes  = extremes_columns [i]->cloneEmpty ();

            new_extremes ->insert (min_value );
            new_extremes ->insert (max_value );

            extremes_columns [i] = std::move(new_extremes );
        }
    }
}
```

Wrap up

The pipeline consists of processors, connected with ports.

Processor and port methods work with chunks.

QueryPipeline

```
$ tree -L 1 -P '*.h' src/QueryPipeline
src/QueryPipeline
├── BlockIO.h
├── Chain.h
├── ...
├── Pipe.h
├── ProfileInfo.h
├── QueryPipelineBuilder.h
├── QueryPipeline.h
├── ...
```

QueryPlan

All of these needs to be tied together, in a **`QueryPipeline`** in ClickHouse.

Recall in **`IStorage::read`** where a **`Pipe`** to source is added to **`query_plan`** with **`QueryPlan::addStep`**.

The step being added is a **`ReadFromStorageStep`**, which is a **`IQueryPlanStep`**.

Steps are added to the **`QueryPlan`**, and **`QueryPlan::buildQueryPipeline`** will build a **`QueryPipeline`** using a **`QueryPipelineBuilder`**.

```
// src/Processors/QueryPlan/QueryPlan.h
class QueryPlan
{
public:
    // ...
    QueryPipelineBuilderPtr buildQueryPipeline(
        const QueryPlanOptimizationSettings &
optimization_settings,
        const BuildQueryPipelineSettings &
build_pipeline_settings);

    struct Node
    {
        QueryPlanStepPtr step;
        std::vector<Node *> children = {};
    };

    using Nodes = std::list<Node>;

private:
    QueryPlanResourceHolder resources;
    Nodes nodes;
    Node * root = nullptr;

    // ...
};
```

Explain plan

There are many of these steps in **`src/Processors/QueryPlan`**.

You can see how your query is translated to transforms with **`EXPLAIN PLAN`**.

```
:) EXPLAIN PLAN SELECT 1 LIMIT 10

EXPLAIN
SELECT 1
LIMIT 10

Query id: b1f7c731-4d60-49cf-8fb6-2c8ba9e54143

┌─explain──────────────────────────────────────────────────┐
│ Expression ((Projection + Before ORDER BY))              │
│   SettingQuotaAndLimits (Set limits and quota after reading from storage)│
│       Limit (preliminary LIMIT (without OFFSET))         │
│       ReadFromStorage (SystemOne)                        │
└──────────────────────────────────────────────────────────┘
```

QueryPipeline

`QueryPipeline` consists of all the **`IProcessors`** connected with **`Ports`** that we just saw.

```
// src/QueryPipeline/QueryPipeline.h
class QueryPipeline
{
public:
    // ...
    bool initialized() const { return !processors.empty(); }
    /// When initialized, exactly one of the following is true.
    /// Use PullingPipelineExecutor or PullingAsyncPipelineExecutor.
    bool pulling() const { return output != nullptr; }
    /// Use PushingPipelineExecutor or PushingAsyncPipelineExecutor.
    bool pushing() const { return input != nullptr; }
    /// Use PipelineExecutor. Call execute() to build one.
    bool completed() const { return initialized() && !pulling() && !pushing(); }
private:
    // ...
    Processors processors;

    InputPort * input = nullptr;

    OutputPort * output = nullptr;
    OutputPort * totals = nullptr;
    OutputPort * extremes = nullptr;

    friend class PushingPipelineExecutor;
    friend class PullingPipelineExecutor;
    friend class PushingAsyncPipelineExecutor;
    friend class PullingAsyncPipelineExecutor;
    friend class CompletedPipelineExecutor;
    friend class QueryPipelineBuilder;
};
```

Executors

```
$ tree -L 1 -P '*.h'  src/Processors/Executor
src/Processors/Executors
├── CompletedPipelineExecutor.h
├── ExecutingGraph.h
├── ExecutionThreadContext.h
├── ExecutorTasks.h
├── IReadProgressCallback.h
├── PipelineExecutor.h
├── PollingQueue.h
├── PullingAsyncPipelineExecutor.h
├── PullingPipelineExecutor.h
├── PushingAsyncPipelineExecutor.h
├── PushingPipelineExecutor.h
├── StreamingFormatExecutor.h
├── TasksQueue.h
├── ThreadsQueue.h
├── traverse.h
└── UpgradableLock.h
```

Executor

`QueryPipeline` is used by executors.

As you can see there are many executors.

Synchronous executors like **`PushingPipelineExecutor`** and **`PullingPipelineExecutor`** all use **`PipelineExecutor::executeStep`**.

```
// src/Processors/Executors/PushingPipelineExecutor.h
class PushingPipelineExecutor
{
public:
    explicit PushingPipelineExecutor(QueryPipeline & pipeline_);
    void push(Chunk chunk);
    void push(Block block);
    // ...
private:
    QueryPipeline & pipeline;
    PipelineExecutorPtr executor;
    // ...
};

void PushingPipelineExecutor::push(Chunk chunk)
{
    if (!started)
        start();

    pushing_source->setData(std::move(chunk));

    if (!executor->executeStep(&input_wait_flag))
        throw Exception(ErrorCodes::LOGICAL_ERROR,
                        "Pipeline for PushingPipelineExecutor was
finished before all data was inserted");
}
```

Executor

Asynchronous executors like **`PushingAsyncPipelineExecutor`** and **`PullingAsyncPipelineExecutor`** work differently.

```
// src/Processors/Executors/PullingPipelineExecutor.h
class PullingPipelineExecutor
{
public:
    explicit PullingPipelineExecutor (QueryPipeline & pipeline_);
    bool pull(Chunk & chunk);
    bool pull(Block & block);
    // ...
private:
    QueryPipeline & pipeline;
    PipelineExecutorPtr executor;
    // ...
};


bool PullingPipelineExecutor ::pull(Chunk & chunk)
{
    if (!executor)
    {
        executor = std::make_shared<PipelineExecutor>(pipeline.processors,
pipeline.process_list_element);
        executor->setReadProgressCallback (pipeline.getReadProgressCallback ());
    }

    if (!executor->checkTimeLimitSoft ())
        return false;

    if (!executor->executeStep (&has_data_flag))
        return false;

    chunk = pulling_format->getChunk ();
    return true;
}
```

Executor

PipelineExecutor build an **ExecutingGraph** and runs the graph in steps.

The graph nodes are **IProcessors** with execution status and locks, and the edges are the connected **Ports**.

```
// src/Processors/Executors/ExecutingGraph.h
class ExecutingGraph
{
public:
   struct Edge
   {
       Edge(uint64_t to_, bool backward_,
            uint64_t input_port_number_, uint64_t output_port_number_,
            std::vector<void *> * update_list)
           : to(to_), backward(backward_)
           , input_port_number(input_port_number_), output_port_number(output_port_number_)
       {
           update_info.update_list = update_list;
           update_info.id = this;
       }

       /// Processor id this edge points to.
       /// It is processor with output_port for direct edge or processor with input_port for
backward.
       uint64_t to = std::numeric_limits<uint64_t>::max();
       bool backward;
       /// Port numbers. They are same for direct and backward edges.
       uint64_t input_port_number;
       uint64_t output_port_number;

       /// Edge version is increased when port's state is changed (e.g. when data is pushed). See
Port.h for details.
       /// To compare version with prev_version we can decide if neighbour processor need to be
prepared.
       Port::UpdateInfo update_info;
   };
   // ...
};
```

ExecutingGraph

`PipelineExecutor` build an **`ExecutingGraph`** and runs the graph in steps.

The graph nodes are **`IProcessor`**s with execution status and locks, and the edges are the connected **`Port`**s.

```cpp
// src/Processors/Executors/ExecutingGraph.h
class ExecutingGraph
{
public:
    struct Node
    {
        /// Processor and it's position in graph.
        IProcessor * processor = nullptr;
        uint64_t processors_id = 0;

        /// Direct edges are for output ports, back edges are for input ports.
        Edges direct_edges;
        Edges back_edges;

        /// Current status. It is accessed concurrently, using mutex.
        ExecStatus status = ExecStatus::Idle;
        std::mutex status_mutex;

        /// Exception which happened after processor execution.
        std::exception_ptr exception;

        IProcessor::Status last_processor_status = IProcessor::Status::NeedData;

        IProcessor::PortNumbers updated_input_ports;
        IProcessor::PortNumbers updated_output_ports;

        Port::UpdateInfo::UpdateList post_updated_input_ports;
        Port::UpdateInfo::UpdateList post_updated_output_ports;

        /// Counters for profiling.
        uint64_t num_executed_jobs = 0;
        uint64_t execution_time_ns = 0;
        uint64_t preparation_time_ns = 0;

        Node(IProcessor * processor_, uint64_t processor_id)
            : processor(processor_), processors_id(processor_id)
        {
        }
    };
    // ...
};
```

PipelineExecutor

`PipelineExecutor` executes the graph in multithreads.

```cpp
// src/Processors/Executors/PipelineExecutor.cpp
void PipelineExecutor ::executeStepImpl (size_t thread_num , std::atomic_bool  * yield_flag )
{
    auto & context  = tasks .getThreadContext (thread_num );
    bool yield = false;

    while (!tasks .isFinished () && ! yield)
    {
        while (!tasks .isFinished () && ! context .hasTask ())
            tasks .tryGetTask (context );

        while (context .hasTask () && ! yield)
        {
            if (tasks .isFinished ())
                break;

            if (!context .executeTask ())
                cancel ();

            if (tasks .isFinished ())
                break;

            if (!checkTimeLimitSoft ())
                break;

            /// Try to execute neighbour processor.
            {
                Queue queue ;
                Queue async_queue ;

                /// Prepare processor after execution.
                if (!graph ->updateNode (context .getProcessorID (), queue, async_queue ))
                    finish ();

                /// Push other tasks to global queue.
                tasks .pushTasks (queue, async_queue , context );
            }

            /// We have executed single processor. Check if we need to yield execution.
            if (yield_flag  && *yield_flag )
                yield = true;
        }
    }
}
```

Wrap up

`QueryPlan` consists of `IQueryPlanStep`s.

`Interpreter` build a `QueryPipeline` from `QueryPlan`.

`PipelineExecutor` build an `ExecutingGraph` from `QueryPipeline`, then executes the steps.

Questions?