clickhouse的data目录下包含如下目录：

```
[root@brfs-stress-01 201403_10_10_0]# ll /data01/clickhouse/data
total 4
drwxr-x--- 2 clickhouse clickhouse  6 Aug 28 19:33 cores
drwxr-x--- 5 clickhouse clickhouse 48 Aug 31 14:25 data
drwxr-x--- 2 clickhouse clickhouse  6 Aug 28 19:33 dictionaries_lib
drwxr-x--- 2 clickhouse clickhouse  6 Aug 28 19:33 flags
drwxr-x--- 5 clickhouse clickhouse 85 Aug 31 14:25 metadata
drwxr-x--- 2 clickhouse clickhouse  6 Aug 28 19:33 metadata_dropped
drwxr-x--- 2 clickhouse clickhouse 39 Aug 28 19:33 preprocessed_configs
-rw-r----- 1 clickhouse clickhouse 59 Aug 28 19:33 status
```

其中与表有关的主要目录有两个：`data`和`metadata`

`metadata`保存了创建数据库及表的sql信息。

`data`目录下会为每个数据库创建一个目录：

```
[root@brfs-stress-01 201403_10_10_0]# ll /data01/clickhouse/data/data
total 0
drwxr-x--- 2 clickhouse clickhouse   6 Aug 28 19:33 default
drwxr-x--- 7 clickhouse clickhouse 108 Aug 28 19:34 system
drwxr-x--- 4 clickhouse clickhouse  36 Aug 31 14:33 tutorial
```

通过客户端查看数据库的信息，可知有三个database，而每个与database对应的目录与database具有相同的名称。进入到database对应的目录后，其内部会为每个表创建一个对应的目录：

```
[root@brfs-stress-01 201403_10_10_0]# ll /data01/clickhouse/data/data/tutorial/
total 0
drwxr-x--- 4 clickhouse clickhouse  66 Aug 31 15:57 hits_v1
drwxr-x--- 8 clickhouse clickhouse 143 Aug 31 16:00 visits_v1
```

通过查询可知，`tutorial`数据库中保存有两个表：`hits_v1`和`visits_v1`。

再进入到表目录中，存储的是按时间进行分区的表数据信息，其中每个分区对应一个目录，目录名为时间信息，至于是什么的时间这个目前还不是很清楚，观察到一个情况是最初入数据时，分区目录很多，基本包含一个月的数据，但通过`OPTIMIZE TABLE tutorial.hits_v1 FINAL`对表进行优化后，数据合并为一个分区了，所以看不出这个分区名称表示的时间是指什么的时间。例如，优化前的目录结构如下：

```
[root@brfs-stress-01 hits_v1]# ls
201403_10_10_0  201403_12_12_0  201403_13_18_1  201403_1_6_1    201403_18_18_0
201403_20_20_0  201403_22_22_0  201403_25_25_0  201403_27_27_0  201403_30_30_0
201403_4_4_0  201403_7_12_1  201403_9_9_0
201403_1_1_0    201403_1_31_2   201403_14_14_0  201403_16_16_0  201403_19_19_0
201403_21_21_0  201403_23_23_0  201403_25_30_1  201403_28_28_0  201403_31_31_0
201403_5_5_0  201403_7_7_0   detached
201403_11_11_0  201403_13_13_0  201403_15_15_0  201403_17_17_0  201403_19_24_1
201403_2_2_0    201403_24_24_0  201403_26_26_0  201403_29_29_0  201403_3_3_0
201403_6_6_0  201403_8_8_0   format_version.txt
```

合并之后，变为：

```
[root@brfs-stress-01 hits_v1]# ls
201403_1_31_2  detached  format_version.txt
```

进入到表目录后，其内部保存有很多文件，可以分为如下几类：

主键文件：

```
-rw-r----- 1 clickhouse clickhouse      378 Aug 31 15:42 primary.idx
```

这个文件保存了主键的值，这个文件的内容总是在内存中常驻。

ClickHouse中的主键是稀疏的，这意味着文件中的每个主键索引只是标志若干行的开始位置，每个主键之间间隔的行数称为索引粒度(index granularity)，默认为8192。主键决定了每列数据的存储顺序，它们是按照主键的字典(lexicographically)顺序排列的。

用于存储用户数据的列文件：

```
<column_name>.bin
<column_name>.mrk2
```

.bin文件是数据文件，其内部包含若干个压缩快(Compressed Block)，根据数据的平均大小，每个块大概保存64K到1M的原始数据(未压缩数据)。每个块中包含的值按主键的顺序依次排列。

.mrk2文件存储每个主键对应的行在数据文件中的偏移量标致。每个标致由两部分组成：行所在的压缩块位于数据文件中的偏移量和行在解压后的数据块中的偏移量。这个文件的内容会被缓存到内存中。

有个别字段包含如下形式的文件：

```
<column_name>.size0.bin
<column_name>.size0.mrk2
```

其他的就是如下形式的文件：

```
[root@brfs-stress-01 201403_10_10_0]# ll | grep -v ".bin" | grep -v ".mrk2"
total 46964
-rw-r----- 1 clickhouse clickhouse    14078 Aug 31 15:42 checksums.txt
-rw-r----- 1 clickhouse clickhouse     4518 Aug 31 15:42 columns.txt
-rw-r----- 1 clickhouse clickhouse        6 Aug 31 15:42 count.txt
-rw-r----- 1 clickhouse clickhouse        4 Aug 31 15:42 minmax_StartDate.idx
-rw-r----- 1 clickhouse clickhouse        4 Aug 31 15:42 partition.dat
```

目前，`columns.txt`保存当前分区中的字段信息，`count.txt`保存当前分区的数据条数。其他的尚不清楚。

ClickHouse为啥这么快

根据官方介绍，ClickHouse快的原因有三个：

1、宏观上使用了一些高效计算的设计方法，比如列式存储、能加载到内存中的索引、数据压缩、向量化计算、计算资源的可拓展性等；

2、细节上ClickHouse没有使用通用实现的算法，而是根据数据的具体情况为其选择并实现特定的算法，这种方式在牺牲算法通用性的情况下可以充分提升算法的效率。可以参考一个例子：https://habr.com/en/company/yandex/blog/457612/；

3、开发团队一直关注这最新的设计理念及算法实现，并不断尝试改进。

ClickHouse赠送了几个构建高效软件的建议：

- 设计系统是一定要重视底层的实现细节
- 要考虑硬件能提供的能力
- 应该根据实际需求来选择数据结构及其抽象
- 特殊情况特殊处理
- 不断进步，尝试更新更好的算法
- 可以根据运行时数据动态选择合适的算法
- 一定要在真实的数据集上进行基准测试
- 使用CI对性能做回归分析测试
- 监控系统的每一个角落，不要漏掉任何细节

ClickHouse的副本管理

副本有两种实现方式：

- Replicated*MergeTree系列表
- 分布式表

分布式表也可以依赖Replicated系列表实现副本。

Replicated

Replicated类型的表有如下几类：

- ReplicatedMergeTree

- ReplicatedSummingMergeTree
- ReplicatedReplacingMergeTree
- ReplicatedAggregatingMergeTree
- ReplicatedCollapsingMergeTree
- ReplicatedVersionedCollapsingMergeTree
- ReplicatedGraphiteMergeTree

副本以table为单位，对于Replicated系列的表需要使用zookeeper保存副本的状态信息，这类table在zookeeper上会有一个对应的路径，具有相同路径的table互为副本。如果没有Zookeeper配置，Replicated系列的表将会是只读状态。

Zookeeper的配置方式如下：

```
<zookeeper>
    <node index="1">
        <host>example1</host>
        <port>2181</port>
    </node>
    <node index="2">
        <host>example2</host>
        <port>2181</port>
    </node>
    <node index="3">
        <host>example3</host>
        <port>2181</port>
    </node>
</zookeeper>
```

通过INSERT语句向表里写入数据时，数据被组织为一系列的block，每个block最多包含max_insert_block_size（默认1048576）行。每个block的写入会向Zookeeper中写入大概10个数据信息。所以向Replicated表中写入数据的延迟比非Replicated表要略微高一些。所以文档中建议通过批量的方式写入数据，并且每秒不要超过一次INSERT（这里不是太懂？！）。每个block的写入具有原子性。并且ClickHouse自带避免重复写入的机制，也就是说，如果连续写入相同的block（具有相同数据的block，大小，顺序，内容都要一致），只有一个会写入成功。

一个副本组里的每个table都可以作为主节点，这意味着每个副本都可以作为数据写入的对象节点。然后这些节点之间通过异步的方式相互同步数据（只同步插入的原始数据），达到数据的一致性。

Replicated系列引擎都有三个参数：

- **zookeeper_path**

  表在Zookeeper上的路径，互为副本的表有相同的路径。

- **replica_name**

  一个副本组内的每个副本有一个自己的replica_name，用于区分这些副本。

- **other_parameters**

  用于特殊类型引擎的参数，比如ReplacingMergeTree的version参数。

分布式表副本

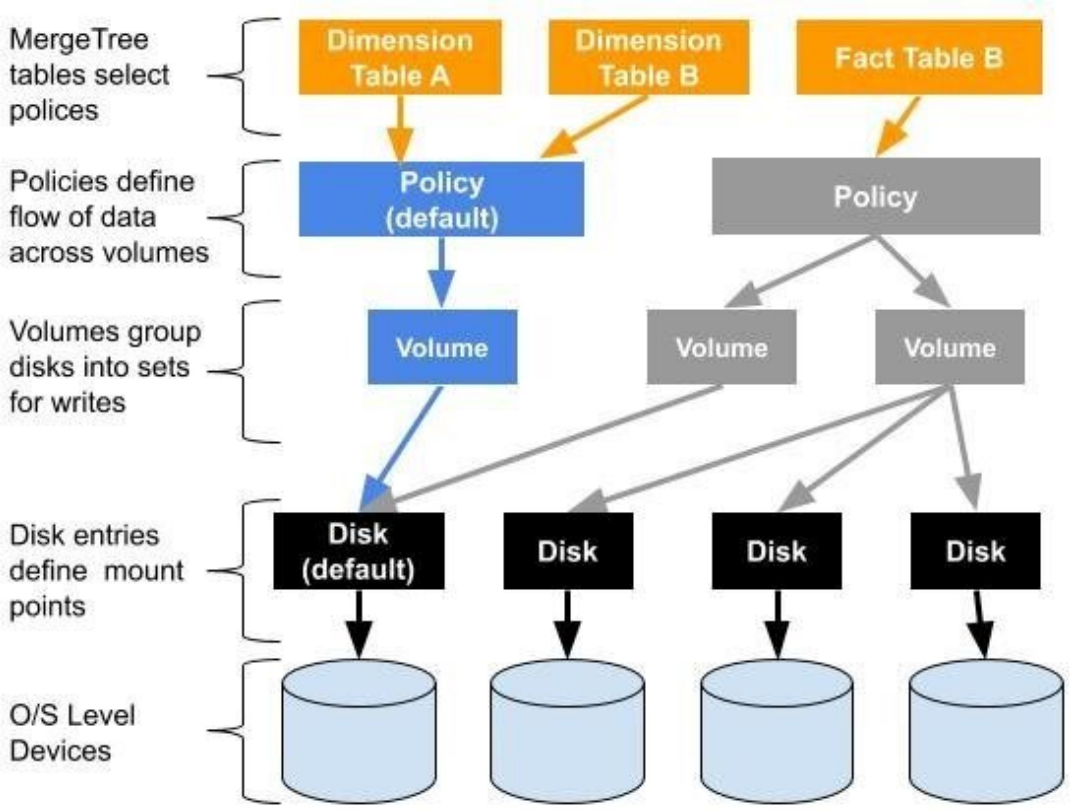
分布式表依赖集群，集群中又可以设置副本。分布式表中的副本只是个声明，这些副本既可以对应Replicated类型的表，也可以对应普通的MergeTree表，这种对应关系会决定数据怎么写入到表中。对于Replicated的表，数据会在副本表之间自动同步，因此副本的`internal_replication`的参数设置为true；对于普通MergeTree表，因为没有自动同步机制，所以要求分布式表自己负责对各个副本的写入工作，通过设置`internal_replication`为false表明这种情况。

ClickHouse的多磁盘存储机制

多磁盘存储的主要目的是为不同类型的数据提供不同的存储方案，实现存储空间的层级化（tiered storage），比如冷热数据分离。（这里的存储分离是面向单个ClickHouse节点，而非集群）

此外，多磁盘存储可以有效的解决单机存储扩容的问题。

这里引用一张别人制作的图说明多磁盘的逻辑概念及结构：



从图中可以了解到，每个物理磁盘会对应文件系统中的一个mount point，多个磁盘组成一个volume，而多个volume又属于一个policy，每个表对应一个policy。

参考：

https://altinity.com/blog/2019/11/27/amplifying-clickhouse-capacity-with-multi-volume-storage-part-1

https://altinity.com/blog/2019/11/29/amplifying-clickhouse-capacity-with-multi-volume-storage-part-2

分布式表

分布式表本身不存储任何数据，它只是把多个本地表关联在一起，提供一个统一操作的接口。

分布式表的engine名为`Distributed`，其包含几个参数：

- 集群名（集群的信息在服务的配置文件中设置）
- 本地表的数据库名
- 本地表的表名
- 用于分区的Key（可选）
- 用于异步发送数据的临时文件存储策略（可选）

由以上配置参数可以看出，分布式表关联的所有本地表必须拥有相同的数据库名及表名。

另外，可以向分布式表中写入数据，其会通过第四个参数sharding key把数据分发到不同的shard中。

同时分布式表把数据发送到本地表的方式分为两种：`sync`和`async`，可以通过配置项`insert_distributed_sync`设置发送模式，0为异步发送，1为同步发送，默认为0。如果设置为同步发送方式，那么写入操作会等到数据已经写入到所有分片后才返回，这里需要注意的是，每条数据只会写入到一个分片中，但通常写入是批量写入，所以一批里的数据可能会分别属于不同的分片，另外，如果分片有多个副本的情况下，则根据配置项`internal_replication`的值判断一个分片什么时候写入成功，如果为true，则只要分片的一个副本写入成功即可；如果为false，则需要写入分片的所有副本后此分片才算写入成功。具体可查阅官方文档中关于`internal_replication`的说明。

对于异步写入，分布式表会把数据临时保存到本地磁盘，最后一个参数用于设置数据保存到磁盘中的策略名，这个参数的作用和`MergeTree`中的`storage_policy`配置一样。

对分布式表的查询和大部分分布式数据库的机制一样，是一个分发合并的过程，先让本地表执行部分查询，然后再把本地表查询的中间结果进行合并汇总后产生最终的查询结果。

集群配置

集群配置的信息必须通过服务的配置文件进行设置，而且，最好为集群涉及到的每个节点都进行集群配置。集群配置可以动态生效，也就是说，对配置文件配置集群信息后不用重启服务也能看到信息的更新。

当前可用的集群信息可以通过系统表`system.clusters`查看。

这里是官方文档中的一个配置示例

```
<remote_servers>
    <logs>
        <shard>
            <!-- Optional. Shard weight when writing data. Default: 1. -->
            <weight>1</weight>
            <!-- Optional. Whether to write data to just one of the replicas.
Default: false (write data to all replicas). -->
            <internal_replication>false</internal_replication>
            <replica>
                <!-- Optional. Priority of the replica for load balancing (see
also load_balancing setting). Default: 1 (less value has more priority). -->
                <priority>1</priority>
                <host>example01-01-1</host>
                <port>9000</port>
            </replica>
            <replica>
```

```
                <host>example01-01-2</host>
                <port>9000</port>
            </replica>
        </shard>
        <shard>
            <weight>2</weight>
            <internal_replication>false</internal_replication>
            <replica>
                <host>example01-02-1</host>
                <port>9000</port>
            </replica>
            <replica>
                <host>example01-02-2</host>
                <secure>1</secure>
                <port>9440</port>
            </replica>
        </shard>
    </logs>
</remote_servers>
```

从配置中可以看到，一个集群中可以包含很多个shard，而每个shard又可以包含多个replica，并且每个shard中的replica个数可以各不相同，最后每个replica对应一个本地表。有一个细节需要注意，之前的内容说分布式表关联的本地表都有相同的数据库名及表名，而每个节点不能创建两个同库同名的表，所以配置中的每个replica都会对应一个唯一的节点。

shard的配置中有几个个重要的参数：

- `internal_replication`

  定义分片中副本之间的同步方式。如果为true，则对分布式表的写入只会写入shard的一个replica，replica之间的数据备份通过本地表自身的机制进行完成；如果为false，则分布式表负责写入shard的所有replica，这种方式没有数据一致性的保证。

- `weight`

  不同的数据被分到不同的shard，weight信息表示shard需要存储的数据的比重，比如有两个分片，一个weight为3，一个weight为5，则前者会存储总数据量的3/8，而后者为5/8，这个值只是设置一个趋势，就想概率一样，只有在数据量足够多的情况下才会显示出这种比例关系。

  sharding key用来决定每行数据应该存储到那个shard。它可以为任意表达式，唯一的要求是其返回值必须为整数。如果想随机分配，可以使用`rand()`函数设置sharding key；也可以根据某些列的值进行划分。

- `priority`

  一个shard中的replica之间保存着相同的数据，查询时只要读取其中一个，`priority`参数用来设置哪个replica会被优先读取，这个参数的值越小，优先级越高。对于副本的选择策略，可以通过配置项`load_balancing`设置，其可被设置的值参考：

  https://clickhouse.tech/docs/en/operations/settings/settings/#settings-load_balancing

  默认为Random。

向分布式表写入数据有两种方式：

1. 直接向分布式表关联的本地表进行写入。这种方式对用户来说最灵活，可以自行定义数据怎么分布，怎么写入，而且每个本地表的写入不会相互影响。但需要用户做更多的事。

2. 向分布式表写入数据。分布式表根据上述参数把数据分发到各个shard存储。数据写入默认是异步方式，接收到的数据会被立即写入本地磁盘上，然后在后台以异步的方式发送到远程表中。异步发送的时间区间有两个配置项控制：`distributed_directory_monitor_sleep_time_ms`和`distributed_directory_monitor_max_sleep_time_ms`，分布式表以第一个参数的时间间隔发送数据，默认为100ms，当发送失败时，这个时间间隔会以指数方式增长，最大不超过第二个参数指定的时间间隔，默认为30s。

   分布式表默认每次发送一个文件，可以通过把`distributed_directory_monitor_batch_inserts`设置为1，打开批量发送的机制，提高发送效率。用于发送数据的线程数通过`background_distributed_schedule_pool_size`设置，默认为16。

   分布式表待发送的数据会存储到分布式表的数据目录下：`/var/lib/clickhouse/data/<database>/<table>/`，这里以ClickHouse默认的配置举例，其中`<database>`和`<table>`分别为分布式表的数据库及表名。

   如果在写入分布式表过程中，如果服务异常终止，则可能会导致写入数据的丢失。（这里有几点不太明白，只要有集群中的节点异常就会丢失数据吗，还是只是写入时分布式表引擎运行的节点异常才会导致数据丢失；其次不太理解数据丢失的原因，因为接收到的数据已经被写入到本地磁盘，难道是在磁盘设备损坏的情况下丢失，还是说丢失的是位于缓存中还没写入磁盘的部分）

磁盘配置

MergeTree系列的引擎支持多磁盘存储，一个表的数据可以分布到多个磁盘上，并且已存储的数据可也以根据条件从一个磁盘转移到另一个磁盘。数据转移的单位是part，也就是说一个part的数据必须都位于同一个磁盘上。

这种机制可以用于冷热数据的存储及转移，比如，热数据存储于ssd，冷数据存储于普通磁盘。

系统中可用的磁盘通过`system.disks`查看。可以在配置文件中添加新的磁盘信息：

```
<storage_configuration>
    <disks>
        <disk_name_1> <!-- disk name -->
            <path>/mnt/fast_ssd/clickhouse/</path>
        </disk_name_1>
        <disk_name_2>
            <path>/mnt/hdd1/clickhouse/</path>
            <keep_free_space_bytes>10485760</keep_free_space_bytes>
        </disk_name_2>
        <disk_name_3>
            <path>/mnt/hdd2/clickhouse/</path>
            <keep_free_space_bytes>10485760</keep_free_space_bytes>
        </disk_name_3>

        ...
    </disks>

    ...
</storage_configuration>
```

其中每个磁盘与一个路径相对应，路径必须以`/`结尾。

`keep_free_space_bytes`声明磁盘中被保留的空余空间。

存储策略

存储策略也是在配置文件中设置。每个策略包含若干个volume，每个volume包含若干个磁盘。

```
<storage_configuration>
    ...
    <policies>
        <policy_name_1>
            <volumes>
                <volume_name_1>
                    <disk>disk_name_from_disks_configuration</disk>

 <max_data_part_size_bytes>1073741824</max_data_part_size_bytes>
                </volume_name_1>
                <volume_name_2>
                    <!-- configuration -->
                </volume_name_2>
                <!-- more volumes -->
            </volumes>
            <move_factor>0.2</move_factor>
        </policy_name_1>
        <policy_name_2>
            <!-- configuration -->
        </policy_name_2>

        <!-- more policies -->
    </policies>
    ...
</storage_configuration>
```

其中，`disk_name_from_disks_configuration`是磁盘配置中声明的磁盘名称。`max_data_part_size_bytes`表示当前volume的磁盘中存储的单个part的最大大小。`move_factor`表示volume的空闲空间比例，当volume的空闲空间少于这个比例，则自动把当前volume的数据移动到下一个volume。所以volume的声明顺序很重要。

创建表的时候，通过`SETTINGS storage_policy=<storage_policy_name>`指定表的存储策略。如果没有指定存储策略，默认会使用系统中的`default`策略，这个策略只包含一个volume，并且这个volume只包含一个磁盘，即配置文件中`<path>`标签指定的存储路径。一旦表创建完毕，存储策略就不能再更改了。

一个part存储到哪个磁盘的选择方法：

1、选择第一个有足够磁盘空间及符合大小限制的volume，即（unreserved_space > current_part_size）并且（max_data_part_size_bytes > current_part_size）；

2、在volume中选择足够空间大小的磁盘（unreserved_space - keep_free_space_bytes > current_part_size）。

可以通过`system.part_log`查看part的移动记录。

AggregateFunction

它是ClickHouse中的一个特殊字段类型，用于保存聚合的中间结果，实现增量式聚合。其声明的格式为：

```
AggregateFunction(name, types_of_arguments…)
```

其中`name`为聚合函数的名称（包括聚合函数使用到的参数），`types_of_arguments`为被执行聚合函数的字段类型。

这个字段内部保存的是聚合的状态，所以对这种字段的写入是通过`<function>State`函数实现的，其中`<function>`为聚合函数名。

例如，创建一个包含如下字段的表：

```
CREATE TABLE t
(
    column1 AggregateFunction(uniq, UInt64),
    column2 AggregateFunction(anyIf, String, UInt8),
    column3 AggregateFunction(quantiles(0.5, 0.9), UInt64),
    ...
) ENGINE = ...
```

那么，写入数据使用如下方式：

```
INSERT INTO t (uniqState(value1), anyIfState(value2, cond), quantilesState(0.5, 0.9)(value3))
```

对于查询数据来说，如果直接SELECT这种类型的字段，返回的是表示聚合函数中间状态的二进制格式的数据，这些数据可以之后直接通过INSERT再次写入对应的字段中。

如果想查询聚合函数的结果，需要通过GROUP BY分组，然后对AggregateFunction字段调用`<function>Merge`函数。其中`<function>`为聚合函数名。

例如查询上述表t中的column1字段，语句如下：

```
SELECT uniqMerge(column1) FROM t GROUP BY ...
```

SimpleAggregateFunction

这个是AggregateFunction类型的简易版，它不像AggregateFunction存储聚合的状态信息，而是直接存储局部聚合的结果。所以它支持的聚合函数有限，主要是可以通过局部聚合结果获取最终聚合结果的聚合函数，比如`sum`，`max`，`min`等。

如下是SimpleAggregateFunction支持的所有聚合函数：

- `any`
- `anyLast`
- `min`
- `max`
- `sum`
- `sumWithOverflow`
- `groupBitAnd`
- `groupBitOr`

- `groupBitXor`
- `groupArrayArray`
- `groupUniqArrayArray`
- `sumMap`
- `minMap`
- `maxMap`

因为这种类型是直接存储聚合后的结果，所以其保存的值类型和原始字段的类型相同，这样写入和查询时就不再使用`-Merge`/`-State`函数。

Architecture

代码实现中存在两类优化查询效率的方法：向量化执行（Vectorized query execution）和动态代码生成（Runtime code generation）。具体描述可参考：

http://15721.courses.cs.cmu.edu/spring2016/papers/p5-sompolski.pdf

表数据在内存中通过Block进行标识，每个Block包含3部分(IColumn, IDataType, column name)，其中IColumn保存表数据，IDataType表明数据类型，column name是相应的列名。Block中的数据是不可变的，对Block中的数据进行计算时只会增加和删除列。

Block中只包含表数据的一部分，表的所有数据以Block流（IBlockInputStream和IBlockOutputStream）表示。每种类型的流可以对Block进行某种计算，其实原理和迭代器一样。流通过pull的方式获取Block，这种方式驱动着Block在不同的Stream类型之间流动，多个Stream组成一个流水线。