Databend

minibend

从零开始打造查询引擎

https://github.com/psiace/databend-internals

Databend
Internals

@PsiACE

Databend

第二弹

CONTENT

Databend

PART 01

前情提要

回顾：minibend ，数据库 和 Rust

MINIBEND

上期要点汇总之 minibend

Databend

minibend

- 从零开始、使用 Rust 构建的查询引擎
- 参考 Databend 设计，致力于降低数据库内核开发门槛

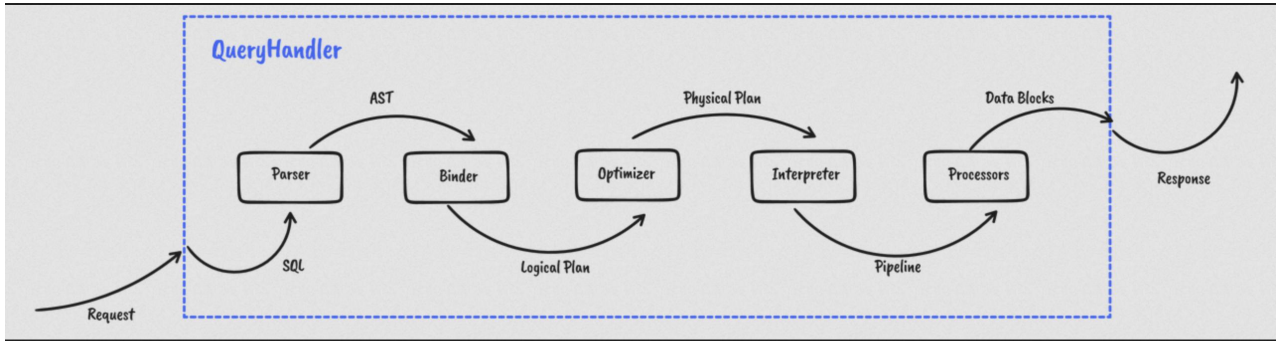
数据库

上期要点汇总之 数据库

Databend

数据库相关的基础知识

- 从存储到云存储
- 不同索引可以对不同查询场景有加速作用
- 查询执行的基本过程
- 查询优化的基本概念
- MPP 和分布式的异同



Rust

上期要点汇总之 Rust

Databend

```rust
// This is a comment, and is ignored by the compiler
// You can test this code by clicking the "Run" button over there ->
// or if you prefer to use your keyboard, you can use the "Ctrl + Enter" shortcut

// This code is editable, feel free to hack it!
// You can always return to the original code by clicking the "Reset" button ->

// This is the main function
fn main() {
    // Statements here are executed when the compiled binary is called

    // Print text to the console
    println!("Hello World!");
}
```

Rust

- 函数和类型
- 模式匹配与控制流
- 内存安全与指针

Databend

PART 02

类型系统和 Arrow

行、列、以及其表示

类型系统

why & how

Databend

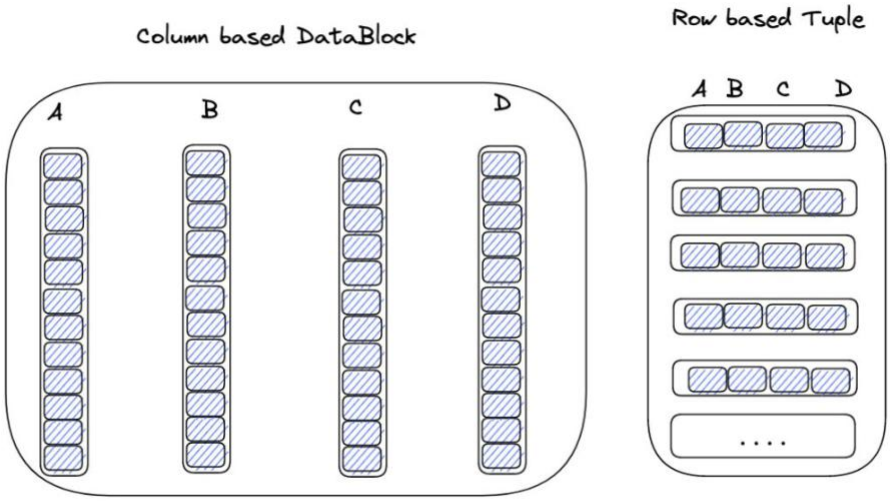
Primitive	Container	Top Type	Bottom Type
`Boolean`	`Array<T>`	`Any`	`Hole`
`String`	`Nullable<T>`		
`Date`			
`Timestamp`			
`Object`			
`Variant`			
`Variadic`			
`UInt8`			
`UInt16`			
`UInt32`			
`UInt64`			
`Int8`			
`Int16`			
`Int32`			
`Int64`			
`Float32`			
`Float64`			

数据在查询引擎中是如何表示的?

单数据源和多数据源的情况下怎么考虑类型系统?

行存还是列存

数据存储时候的模型




```
select pow(A, B),  C + 2 from table where C > 0
```

对于 OLAP 系统，往往处理大量数据，更需要关注数据的吞吐量和执行效率，采用列式存储具有天然的优势。

- 只需要读取需要的列，无需经 IO 读取其余列，从而避免不必要的 IO 开销。
- 同一列的数据中往往存在大量的重复项，压缩率会非常高，进一步节约 IO 资源。
- 利用向量化处理和 SIMD 指令进行优化，提高性能。

Arrow

通用、跨语言、高性能的列式数据内存格式规范

Databend

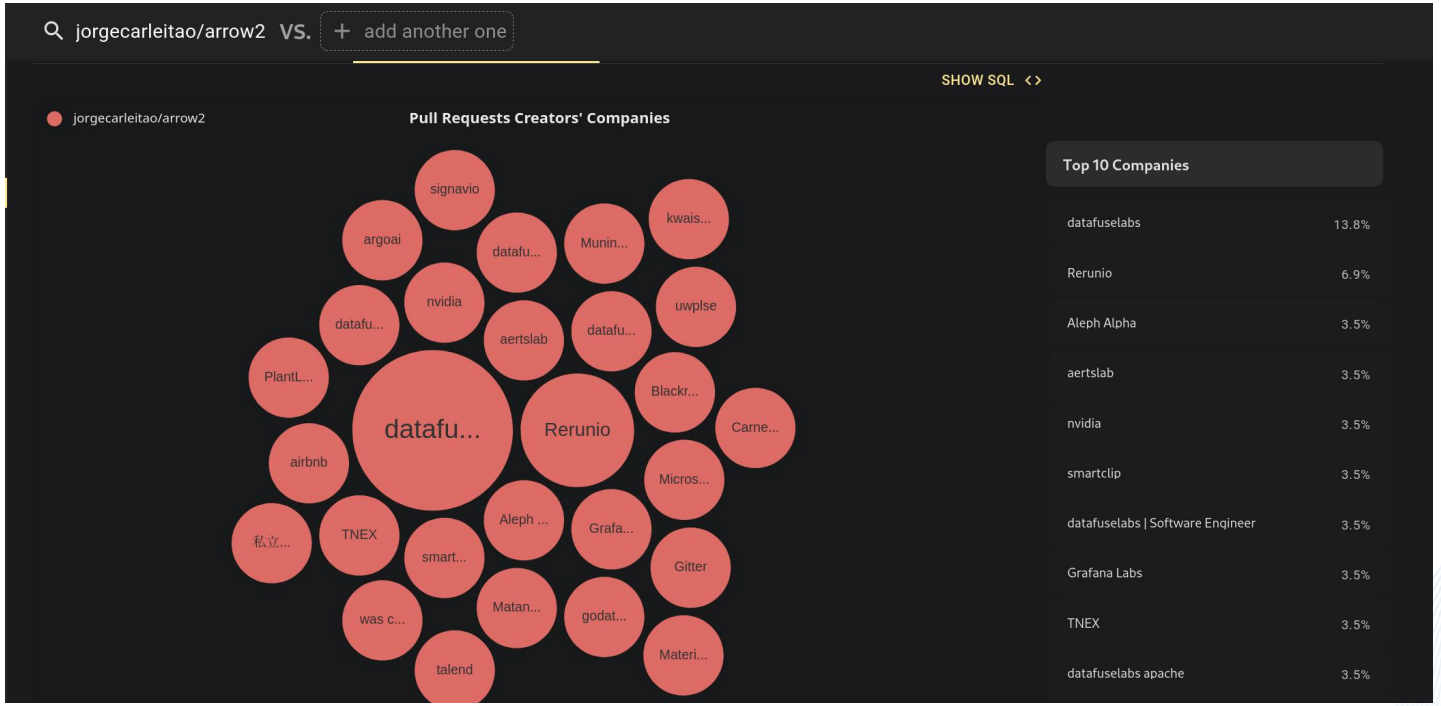
- 现代硬件的向量化执行能力
- 大数据分析系统和应用程序之间的互操作性

Databend -> minibend

一些异同和有意思的事情

Databend

- Databend 是面向海量数据设计的云数仓，面向分析型工作负载进行设计，采用列式存储，使用 Apache Arrow 作为内存格式规范，并在此基础上设计开发类型系统。minibend 在这一点上将会与 Databend 保持一致。



Databend

PART 03

Data Source

与数据一起工作

Data Souce

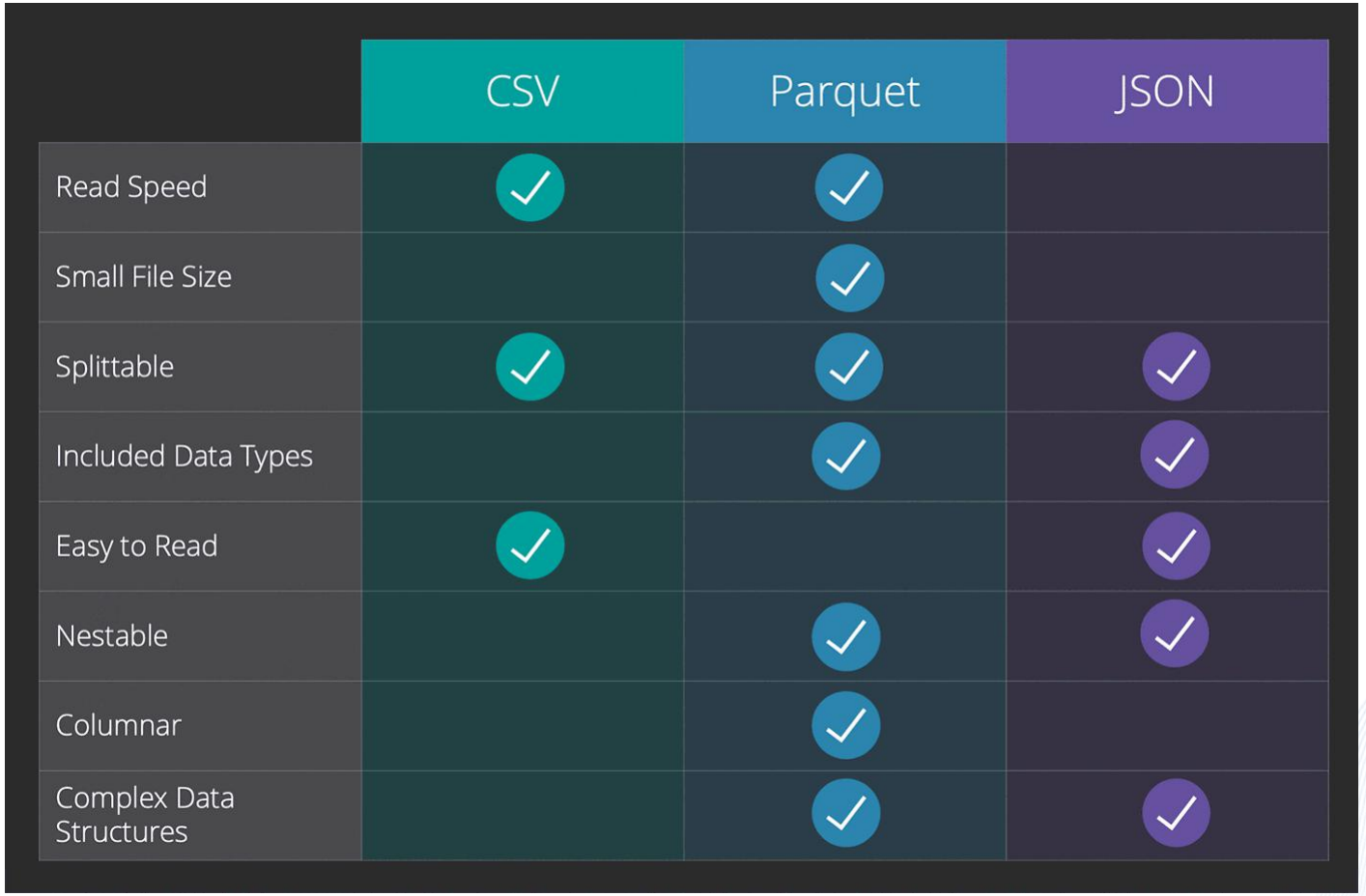
各种各样的数据源

Databend

数据源可以以不同的形态出现。

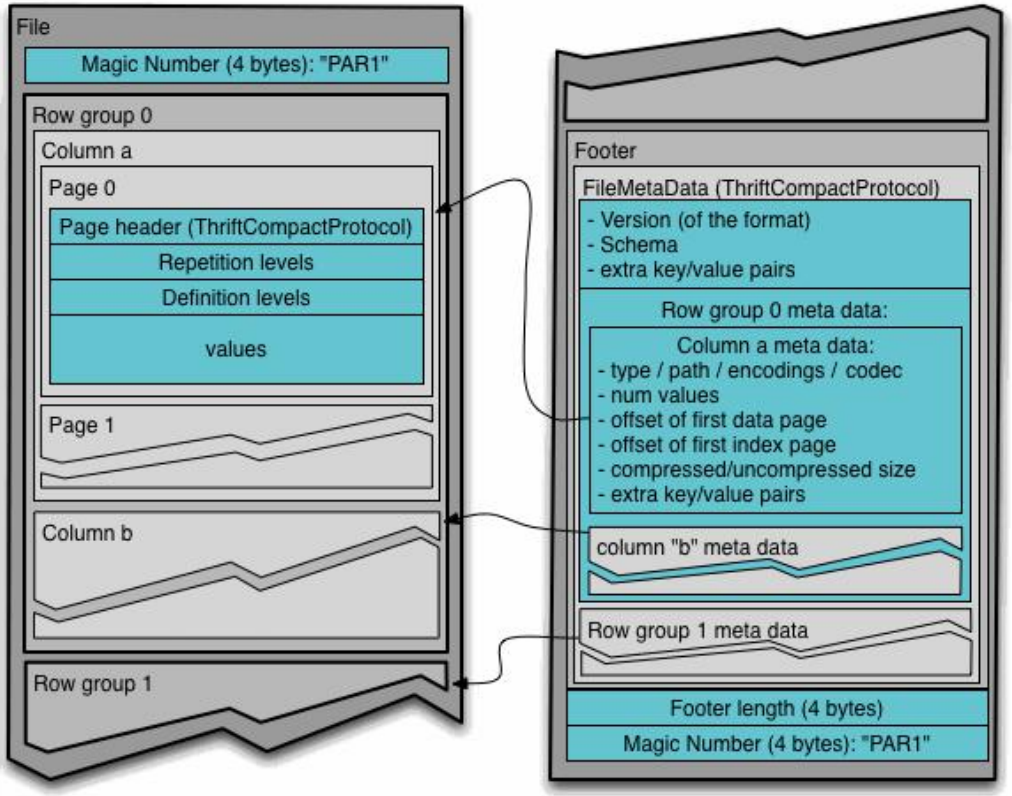
查询引擎需要定义一套统一的接口，并确保能够返回符合预期的数据。

- schema
- data



Parquet

开源的、面向列的数据文件格式



高效的数据压缩和编码方案。

受到 Google Dremel 格式启发。

存储模型主要由行组（Row Group）、列块（Column Chuck）、页（Page）组成。

Databend -> minibend

一些异同和有意思的事情

- Databend 的底层存储格式为 Parquet ，过去其他格式的数据需要通过 Streaming Load 或者 Copy Into 等方式转换到 Databend 支持的 Parquet 格式。而在近期的设计和实现中，Databend 开始逐步实现对位于本地/远端的文件进行查询的支持。

- minibend 将会考虑优先从查询本地现有数据文件开始进行支持。首先是支持 Parquet 作为数据源，但为了方便浏览数据和审计查询结果，或许对 CSV 格式的支持应该提上日程。

```
> export CONFIG_FILE=tests/local/config/databend-local.toml
> cargo run --bin=databend-local -- --sql="SELECT * FROM tbl1" --table=tbl1=/path/to

exec local query: SELECT * FROM tbl1
+-----------------------------+---------------------+------+
| title                       | author              | date |
+-----------------------------+---------------------+------+
| Transaction Processing      | Jim Gray            | 1992 |
| Readings in Database Systems | Michael Stonebraker | 2004 |
| Transaction Processing      | Jim Gray            | 1992 |
| Readings in Database Systems | Michael Stonebraker | 2004 |
+-----------------------------+---------------------+------+
4 rows in set. Query took 0.015 seconds.
```

Databend

代码时间

Cargo.toml

初次见面的一些注意事项

Databend

```toml
[package]
name = "minibend"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[features]
default = []
simd = ["arrow2/simd"]

[dependencies]
arrow2 = { version = "0.15.0", features = ["io_parquet"] }
async-fn-stream = "0.2"
futures = "0.3"
thiserror = "1.0"
[dev-dependencies]
tokio = { version = "1.23", features = ["full"] }
```

- 使用 features 可以控制特性开关
- 区分一般依赖和开发依赖
- arrow2: Apache Arrow in Rust
- async-fn-stream: Stream trait 的实现
- futures: 定义实现异步控制流
- thiserror: 错误处理
- tokio: 异步运行时

lib.rs 与代码组织

如何更好管理 Rust 项目

Databend

```rust
#[allow(dead_code)]
mod catalog;
mod datablock;
mod error;
mod source;
```

```
✦ ❯ tree
.
├── Cargo.toml
├── README.md
├── src
│   ├── catalog.rs
│   ├── datablock.rs
│   ├── error.rs
│   ├── lib.rs
│   ├── source
│   │   └── parquet.rs
│   └── source.rs
└── tests
    └── source
        └── alltypes_plain.parquet

4 directories, 9 files
```

error.rs 与错误处理

thiserror 的初步使用

Databend

- 派生 Error 。
- 使用 #[error(...)] 为错误生成 Display 实现。
- 使用 #[from] 定义错误的 source 和 backtrace 。

- 例子: minibend io error: No such file or directory (os error 2)

```rust
use thiserror::Error;

#[derive(Error, Debug)]
pub enum Error {
    #[error("minibend io error: {0}")]
    IO(#[from] std::io::Error),
    #[error("minibend arrow2 error: {0}")]
    Arrow(#[from] arrow2::error::Error),
    #[error("minibend no such table error: {0}")]
    NoSuchTable(String),
}

impl From<String> for Error {
    fn from(v: String) -> Self {
        Self::NoSuchTable(v)
    }


pub type Result<T> = std::result::Result<T, Error>;
```

datablock.rs 和 source.rs

实现具体数据源之前

Databend

- 定义 DataBlock，实质上就是 RecordBatch ，也是 Chunk
- TableRef 可以看作 DataSource 的一个引用
- DataSource 定义了我们之前提到的需要关心的两个操作

```rust
use std::sync::Arc;

use arrow2::datatypes::Schema;

use crate::datablock::DataBlockStream;

pub mod parquet;

pub type TableRef = Arc<dyn DataSource>;

pub trait DataSource {
    /// Returns the schema of the underlying data
    fn schema(&self) -> Schema;

    /// Returns a stream of DataBlocks
    fn scan(&self, projection: Option<Vec<String>>) -> DataBlockStream;
}
```

```rust
use std::pin::Pin;

use arrow2::array::Array;
use arrow2::chunk::Chunk;
use futures::Stream;

use crate::error::Result;

pub type DataBlock = Chunk<Box<dyn Array>>;
pub type DataBlockStream = Pin<Box<dyn Stream<Item = Result<DataBlock>> + Send + Sync + 'static>>;
```

parquet.rs

如何读取 Parquet

```rust
use std::fs::File;
use std::sync::Arc;

use arrow2::array::Array;
use arrow2::chunk::Chunk;
use arrow2::datatypes::Schema;
use arrow2::io::parquet::read::*;
use async_fn_stream::fn_stream;

use crate::datablock::DataBlockStream;
use crate::error::Result;
use crate::source::DataSource;

use super::TableRef;

#[derive(Debug, Clone)]
pub struct ParquetTable {
    pub path: String,
}

impl ParquetTable {
    pub fn create(path: String) -> Result<TableRef> {
        Ok(Arc::new(Self { path }))
    }

    fn get_reader(&self) -> Result<FileReader<File>> {
        let mut file = File::open(self.path.clone())?;
        let metadata = read_metadata(&mut file)?;
        let schema = infer_schema(&metadata)?;
        let reader = FileReader::new(
          file, metadata.row_groups, schema, None, None, None
        );
        Ok(reader)
    }
}
```

```rust
impl DataSource for ParquetTable {
    fn schema(&self) -> Schema {
        let reader = self.get_reader().unwrap();
        reader.schema().clone()
    }

    fn scan(&self, projection: Option<Vec<String>>) -> DataBlockStream
{
        let reader = self.get_reader().unwrap();

        let indexes = projection.map(|projection| {
            projection
                .iter()
                .map(|p| {
                    self.schema()
                        .fields
                        .iter()
                        .enumerate()
                        .find(|(_idx, field)| field.name.eq(p))
                        .map(|(idx, _field)| idx)
                        .unwrap()
                })
                .collect::<Vec<_>>()
        });

        // need to consider only relevant columns
        let output = fn_stream(|emitter| async move {
            for maybe_chunk in reader {
                let chunk = maybe_chunk.unwrap();
                let result_chunk = match indexes {
                    Some(ref indexes) => {
                        let arrays = chunk.arrays();
                        let mut r: Vec<Box<dyn Array>> = Vec::new();
                        for idx in indexes {
                            let array = arrays.get(*idx).unwrap();
                            r.push(array.clone());
                        }
                        Chunk::new(r)
                    },
                    None => chunk,
                };
                // yield elements from stream via `emitter`
                emitter.emit(Ok(result_chunk)).await;
            }
        });

        Box::pin(output) as DataBlockStream
    }
}
```

Databend

Catalog

使用 Catalog 来管理表

Databend

```rust
use std::collections::HashMap;

use crate::error::{Error, Result};
use crate::source::parquet::ParquetTable;
use crate::source::TableRef;

#[derive(Default)]
pub struct Catalog {
    tables: HashMap<String, TableRef>,
}

impl Catalog {
    /// Add parquet table
    pub fn add_parquet_table(&mut self, table: &str, path: &str) -> Result<()> {
        let source = ParquetTable::create(path.into())?;
        self.tables.insert(table.into(), source);
        Ok(())
    }

    /// Get table
    pub fn get_table(&self, table: &str) -> Result<TableRef> {
        self.tables
            .get(table)
            .cloned()
            .ok_or_else(|| Error::NoSuchTable(format!("Unable to get table named: {}", table)))
    }
}
```

- 实现简单的增和查
- 使用 HashMap 维护映射关系

Unit Tests

简单聊聊单元测试

Databend

- 一般建议：对外暴露的可以考虑集成测试，内部使用单元测试
- 主要区别在于，编译时如何看待你的 crate
- 使用 tokio 测试异步函数

```rust
#[cfg(test)]
mod tests {
    use super::*;
    use crate::error::Result;
    use futures::StreamExt;

    #[tokio::test]
    async fn test_format_parquet() -> Result<()> {
        let test_file = format!("tests/source/alltypes_plain.parquet");
        let mut catalog = Catalog::default();
        catalog.add_parquet_table("parquet", &test_file)?;
        let table = catalog.get_table("parquet")?;

        let mut rbs = table.scan(None);
        let mut actual_row_count = 0;

        if let Some(rrb) = rbs.next().await {
            let rb = rrb?;
            assert_eq!(rb.columns().len(), 11); // all columns are expected
            actual_row_count += rb.columns().get(0).unwrap().len();
        }
        assert_eq!(actual_row_count, 8);

        Ok(())
    }
}
```

Databend

PART 04

前进四

回顾、展望和参考资料

回顾

本期要点汇总

Databend

在今天的内容中，我们简单介绍了类型系统和数据源的一些相关内容：

- 类型系统用于处理数据在查询引擎中的表示。
- 对于 OLAP 系统而言，基于列式存储会更能发挥现代硬件的能力。
- 数据源可以是多种多样的，文件、数据库、内存对象都可以作为数据源。
- Apache Arrow 和 Apache Parquet，前者是一套通用、跨语言、高性能的列式数据内存格式规范，后者是一种旨在实现最大空间效率的存储格式。

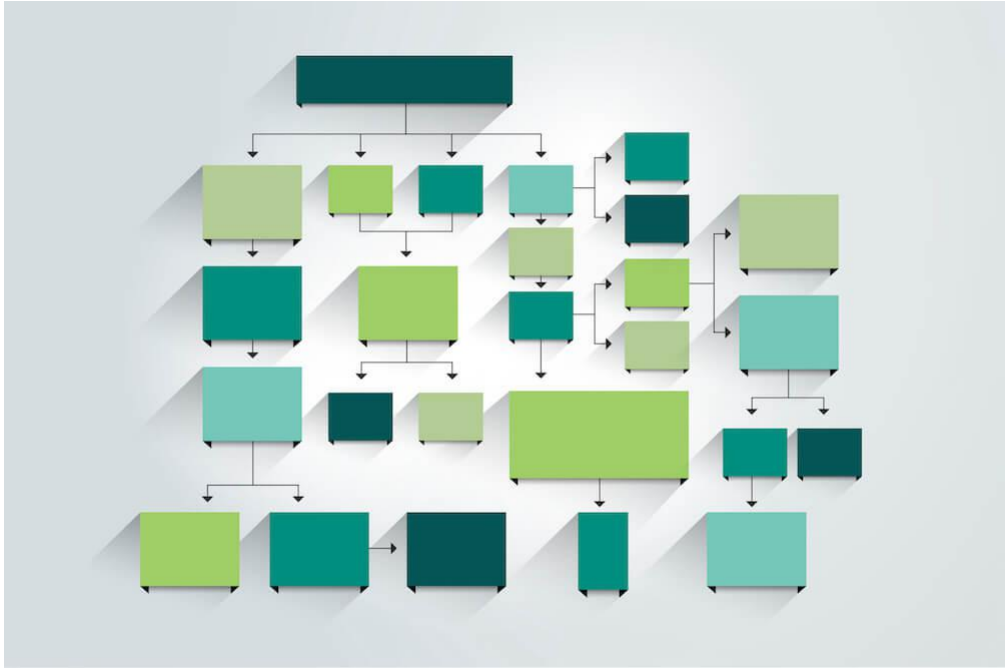
当然，在这一期的代码时间，我们初步建立了 minibend 的基础，并支持使用 Parquet 文件作为数据源。

展望

下期内容更加精彩

Databend

- 深入类型系统
- 逻辑计划和表达式



参考资料

更多有价值的资料看这里

Databend

一个是 风空之岛 ，@mwish 的技术博客，有关于 Parquet 的一个更详细的系列介绍，并且还有论文阅读的部分。

另一个是 数据库内核月报 ，来自阿里云 PolarDB 数据库内核团队。

数据库内核月报 — 2022 / 12

PolarDB MySQL 新特性 - Partial Result Cache	# 01
MySQL Temporal Data Types	# 02
Innodb 中的 Btree 实现 (一) · 引言 & insert 篇	# 03
MySQL · 业务场景 · 业务并发扣款，金额未扣	# 04
PolarDB MySQL · 功能特性 · Fast Query Cache技术详解与最佳实践	# 05
PolarDB MySQL · 功能特性 · 大表分页查询优化	# 06
PolarDB MySQL · 功能特性 · SQL Trace	# 07

Databend

Thank !

- https://psiace.github.io/databend-internals/

Databend Internals

@PsiACE