minibend

从零开始打造查询引擎

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

Databend Internals

@PsiACE

Databend

第一弹

CONTENT

1 minibend

简介：目标与设计

2 数据库基础概念

查询以及存储

3 Rust 不完全指南

如何快速上手 Rust 代码

4 前进四

回顾、展望和参考资料

Databend

PART 01

minibend

简介：目标与设计

What

minibend 是什么

Databend 内幕大揭秘　在线阅读

Databend 内幕大揭秘

与 Databend 一同探秘数据库系统

即刻启程

CC-BY 4.0 & Apache 2.0 GitHub

基础导览

化繁为简，精心编排。
专设基础知识导读，即刻开启无痛学习。

特性探索

能用 -> 易用 -> 好用。
以 Databend 为例，揭示现代云数仓特性。

源码解读

在 Real World 中寻找版本答案。
透过 Databend 深入数据库设计与实现。

开源贡献

为 Databend 添砖加瓦。
一起学习如何为开源现代云数仓做贡献。

实战演练

从 0 到 1 ，构建属于你的实时数仓。
从 Minibend 开始数据库内核研发之旅。

专题研讨

不定期的论文/技术研讨会。
关注数据库、分布式等相关领域新动态。

由 GitHub Pages 、Zola 及 AdiDoks 强力驱动　　隐私政策　行为准则

minibend

- 使用 Rust 构建
- 从零开始的查询引擎
- Databend Internals 实战部分

Why

为什么设计和实现 minibend

Databend

Cloud Data Warehouse

Built for Elasticity and Efficiency.

Free and Open.

https://github.com/datafuselabs/databend 开发迭代很快，且代码量庞大，新开发者参与门槛较高。

现有的教程，不是 Rust 实现/缺乏一定的 step by step 指南，而且设计思想/实现逻辑上和 Databend 存在差异。

How

计划如何开着这一系列内容

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

提供的材料： 视频、文章、代码

更新的频率： 每月大概提供一至两期

内容的划分： 相关知识导读
设计和实现
精选论文摘要
回顾和展望

Databend

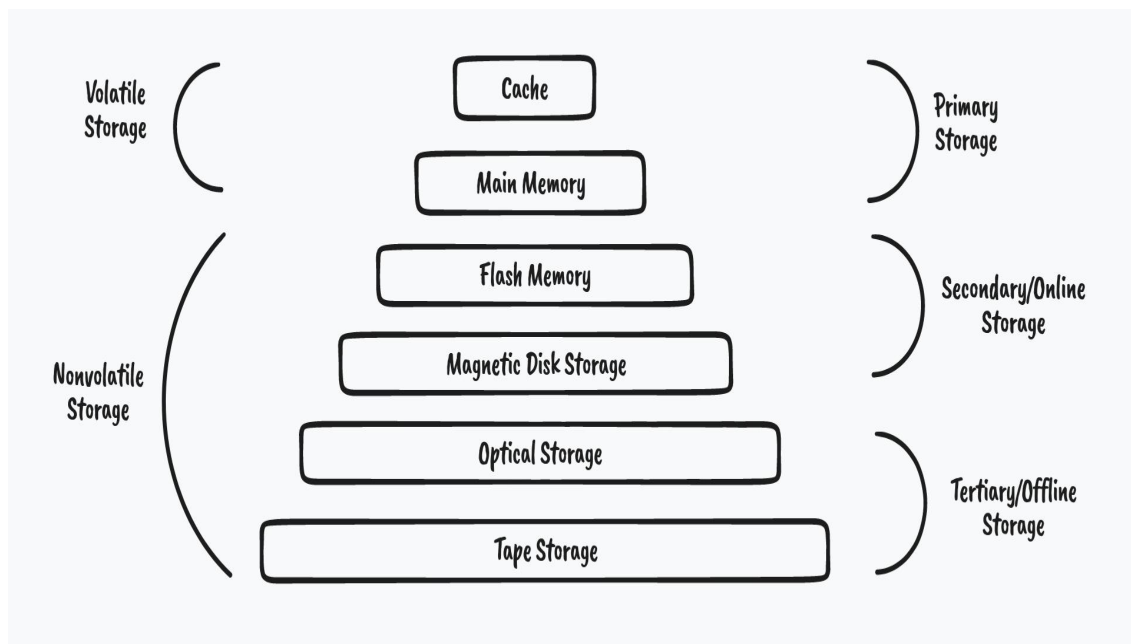
PART 02

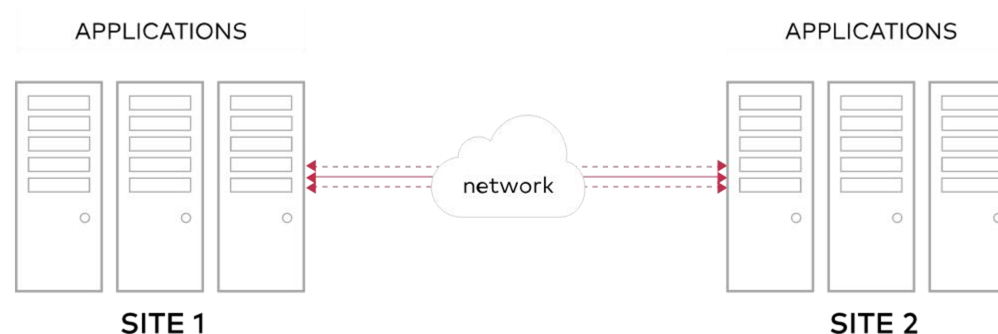
数据库基础概念

查询以及存储

存 储

存储模型和存储形态的一个快速概览





从单机存储进入到云存储时代，为数据库查询引擎的设计和实现带来了新的机遇和挑战

当然，过去的经验和见解并非失效，同样可以通过现有的一些技术改善数据库系统性能

索引

为什么我们需要一些索引

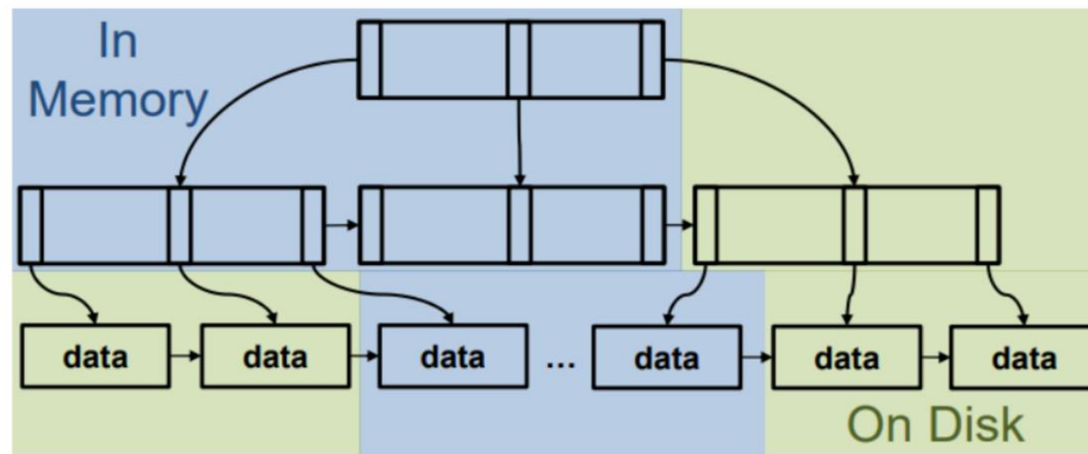
索引

加快数据查询的速度，
但构建和维护也需要花费一定时间。

B-Tree-Index 将键映射到有序数组中的位置。

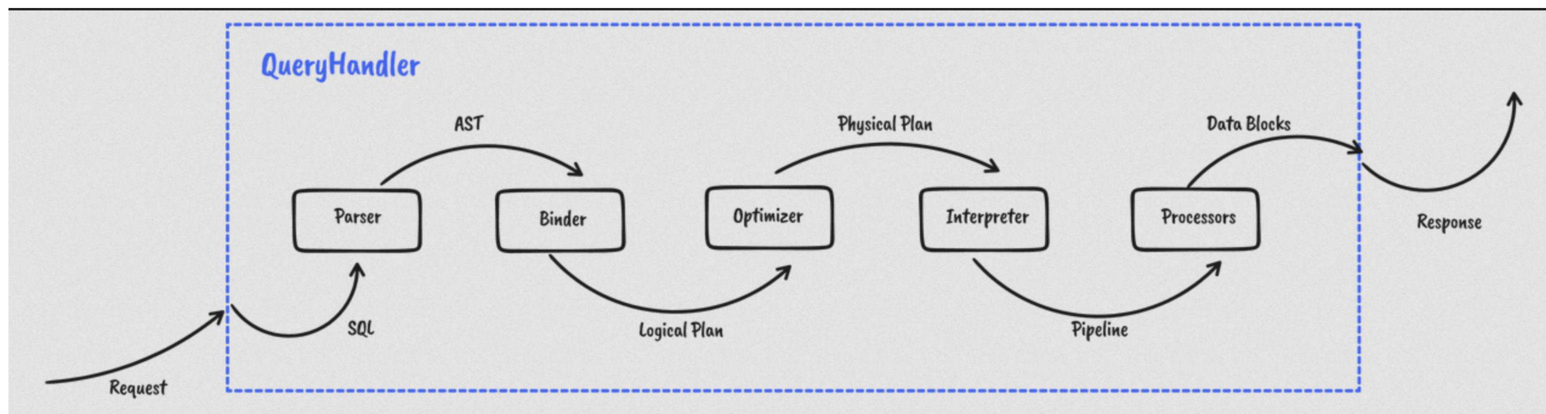
Hash-Index 将键映射到无序数组中的位置。

BitMap-Index 用于判断数据记录是否存在。



查询执行

查询引擎的工作是如何进行的



Morsel-Driven Parallelism

列式存储与向量化执行

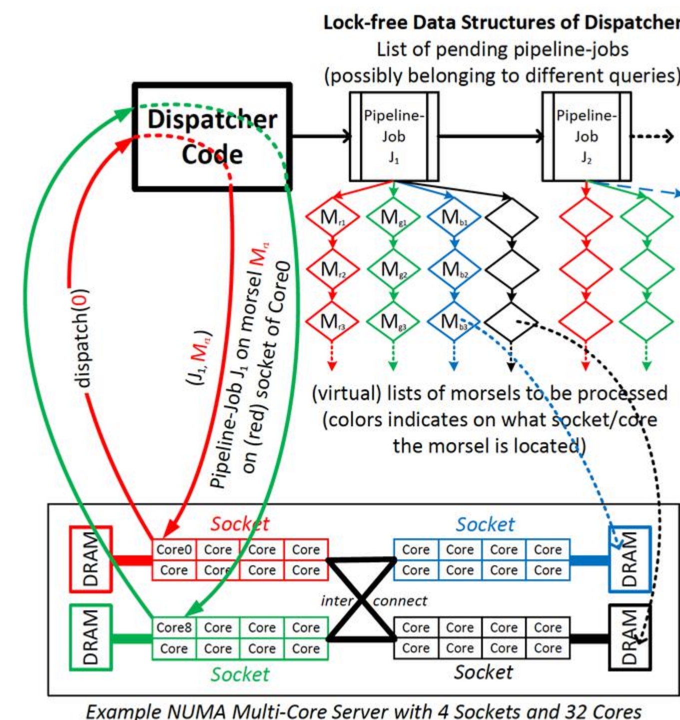


Figure 5: Dispatcher assigns pipeline-jobs on morsels to threads depending on the core

- 解析 SQL 语法，形成 AST（抽象语法树）。
- 通过 Binder 对其进行语义分析，并且生成一个初始的 Logical Plan（逻辑计划）。
- 得到初始的 Logical Plan 后，优化器会对其进行改写和优化，最终生成一个可执行的 Physical Plan 。
- 通过 Optimizer 生成 Physical Plan 后，将其翻译成可执行的 Pipeline 。
- Pipeline 则会交由 Processor 执行框架进行计算。

查询优化

对查询优化建立一个快速概念

两种主要的查询优化方案，一种是基于关系代数和算法的等价优化方案，一种是基于评估成本的优化方案。

查询优化通常包含以下四个步骤：

- 构建框架来列举可能的计划
- 编写转换规则
- 引入成本模型来评估不同的计划
- 选择最理想的计划

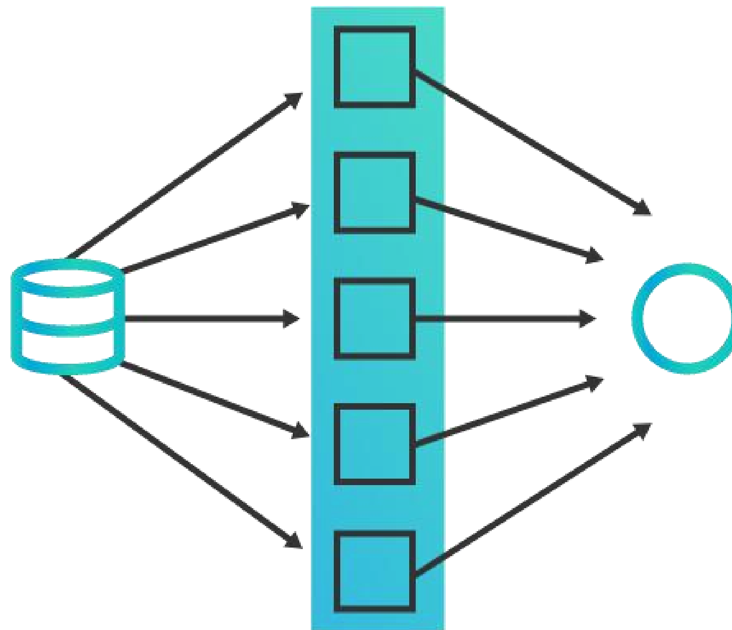
大规模并行处理

面向大数据处理所提出的典型架构

大规模并行处理（MPP，Massively Parallel Processing）意味着可以由多个计算节点（处理器）协同处理程序的不同部分，而每个计算节点都可能具备独立的系统资源（磁盘、内存、操作系统）。

采用大规模并行处理架构设计的系统往往具备以下特性：

- 任务并行执行
- 数据分布式存储
- 分布式计算
- 私有资源
- 水平拓展
- Shared Nothing



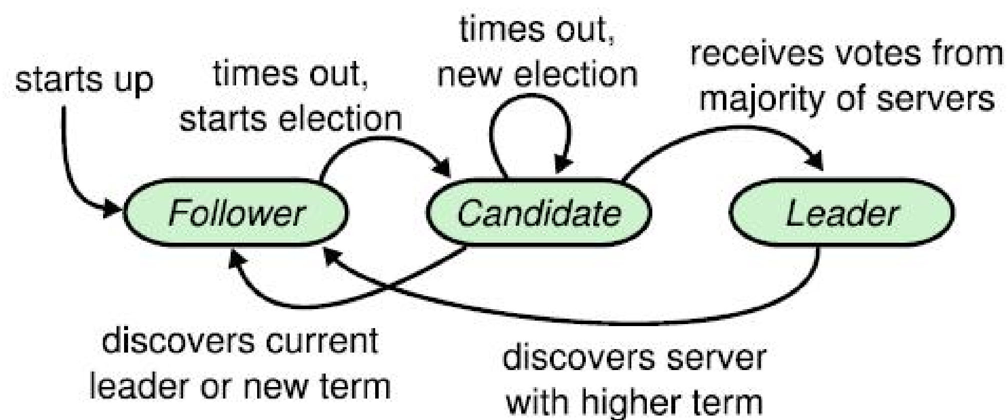
分布式

至于于构建更可靠和可用的系统

分布式系统是由一组计算节点通过网络链接组成的服务系统，作为整体对外提供服务。

作为一个松耦合的系统，分布式系统具有以下特点：

- 分布性：节点在地理位置上可能是分散的。
- 对等性：分布式系统各个节点是对等的，节点不仅可以协同完成任务，同时也可以自治处理任务。
- 并发性：分布式系统中的各个节点可以并发操作共享资源。



Databend

PART 03

Rust 不完全指南

如何快速上手 Rust 代码

什么是 Rust

the rust programming language

Rust

一门赋予每个人
构建可靠且高效软件能力的语言。

Rust 是 Mozilla Research 开发的编程语言，第一个版本发布于 2012 年 1 月。

高性能、可靠性、生产力。

函数和类型

从一些看起来更基本的要素开始

```rust
#[allow(dead_code)]
// Functions
// `i32` is the type for 32-bit signed integers
fn add2(x: i32, y: i32) -> i32 {
    // Implicit return (no semicolon)
    x + y
}
```

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

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

```rust
// Struct with Generics
struct Foo<T> { bar: T }

// Tuple Struct
struct Point2(i32, i32);

// Enum with fields
enum OptionalI32 {
    AnI32(i32),
    Nothing,
}

// Methods and Traits
trait Frobnicate<T> {
    fn frobnicate(self) -> Option<T>;
}

impl<T> Frobnicate<T> for Foo<T> {
    fn frobnicate(self) -> Option<T> {
        Some(self.bar)
    }
}
```

模式匹配与控制流

分支、更多分支

```rust
let foo = OptionalI32::AnI32(1);
match foo {
    OptionalI32::AnI32(n) => println!("it's an i32: {}", n),
    OptionalI32::Nothing  => println!("it's nothing!"),
}

// Advanced pattern matching
struct FooBar { x: i32, y: OptionalI32 }
let bar = FooBar { x: 15, y: OptionalI32::AnI32(32) };

match bar {
    FooBar { x: 0, y: OptionalI32::AnI32(0) } =>
        println!("The numbers are zero!"),
    FooBar { x: n, y: OptionalI32::AnI32(m) } if n == m =>
        println!("The numbers are the same"),
    FooBar { x: n, y: OptionalI32::AnI32(m) } =>
        println!("Different numbers: {} {}", n, m),
    FooBar { x: _, y: OptionalI32::Nothing } =>
        println!("The second number is Nothing!"),
}
```

```rust
// for and ranges
for i in 0u32..10 {
    print!("{} ", i);
}
println!("");
// prints `0 1 2 3 4 5 6 7 8 9 `

// `if` as expression
let value = if true {
    "good"
} else {
    "bad"
};

// `while` loop
while 1 == 1 {
    println!("The universe is operating normally.");
    // break statement gets out of the while loop.
    //  It avoids useless iterations.
    break
}

// infinite loop
loop {
    println!("Hello!");
    // break statement gets out of the loop
    break
}
```

内存安全与指针

现在，一起来看看内存安全相关的部分

```
    // Owned pointer – only one thing can ‘own’ this pointer at a time
    // This means that when the `Box` leaves its scope, it can be
automatically deallocated safely.
    let mut mine: Box<i32> = Box::new(3);
    *mine = 5; // dereference
    // Here, `now_its_mine` takes ownership of `mine`. In other words,
`mine` is moved.
    let mut now_its_mine = mine;
    *now_its_mine += 2;

    println!("{}", now_its_mine); // 7
    // println!("{}", mine); // this would not compile because
`now_its_mine` now owns the pointer
```

```
    // Reference – an immutable pointer that refers to other data
    // When a reference is taken to a value, we say that the value has been
‘borrowed’.
    // While a value is borrowed immutably, it cannot be mutated or moved.
    // A borrow is active until the last use of the borrowing variable.
    let mut var = 4;
    var = 3;
    let ref_var: &i32 = &var;

    println!("{}", var); // Unlike `mine`, `var` can still be used
    println!("{}", *ref_var);
    // var = 5; // this would not compile because `var` is borrowed
    // *ref_var = 6; // this would not either, because `ref_var` is an
immutable reference
    ref_var; // no-op, but counts as a use and keeps the borrow active
    var = 2; // ref_var is no longer used after the line above, so the
borrow has ended
```

```
    // Mutable reference
    // While a value is mutably borrowed, it cannot be accessed at all.
    let mut var2 = 4;
    let ref_var2: &mut i32 = &mut var2;
    *ref_var2 += 2;         // '*' is used to point to the mutably borrowed
var2

    println!("{}", *ref_var2); // 6 , // var2 would not compile.
    // ref_var2 is of type &mut i32, so stores a reference to an i32, not
the value.
    // var2 = 2; // this would not compile because `var2` is borrowed.
    ref_var2; // no-op, but counts as a use and keeps the borrow active
until here
```

Databend

PART 04

前进四

回顾、展望和参考资料

回顾

本期要点汇总

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

- 函数和类型
- 模式匹配与控制流
- 内存安全与指针

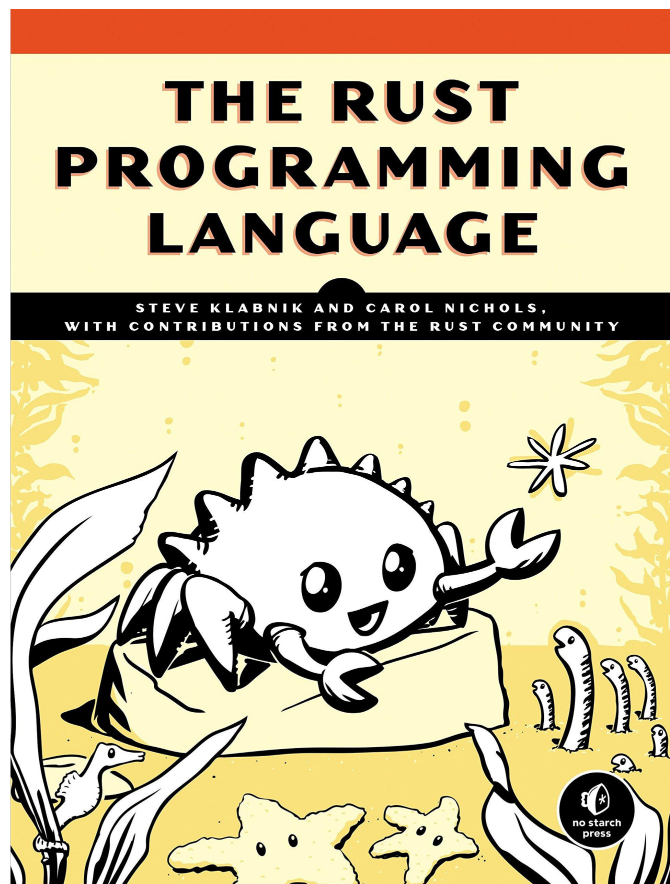
展望

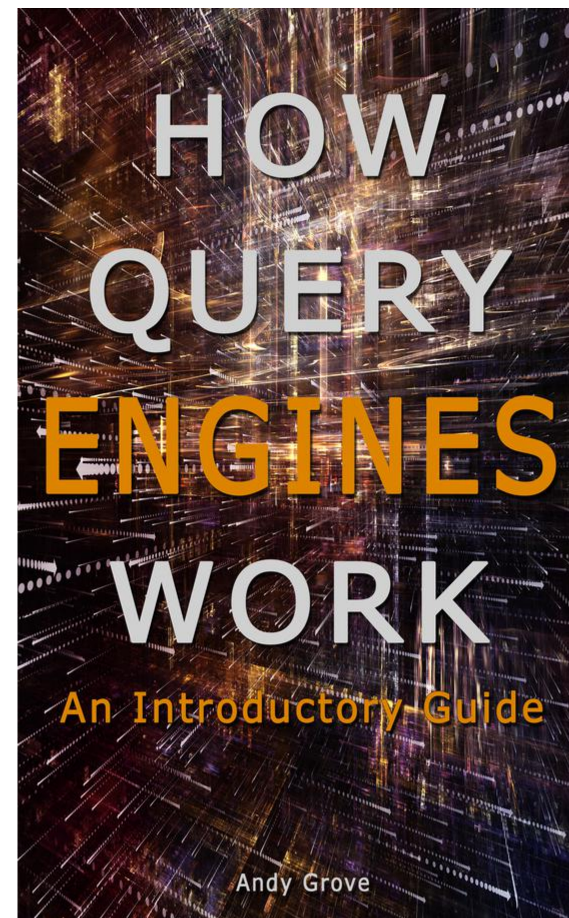
下期内容更加精彩

- Apache Arrow
- 查询引擎中的类型系统
- DataSource

参考资料

更多有价值的资料看这里





Databend

Thank !

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

Databend Internals

@PsiACE