GREENPLUM DATABASE®

Greenplum分布式事务和两阶段提交协议

钉钉直播 | 10月21日 20：00 - 21：00

Greenplum中文社区

https://cn.greenplum.org

博文 · 资料 · 文档 · 项目

全新的问答论坛

https://cn.greenplum.org/askgp

GREENPLUM DATABASE®

微信技术讨论群
添加入群小助手：gp_assistant

微信公众号
技术干货、行业热点、活动预告

欢迎访问Greenplum中文社区：cn.greenplum.org

Greenplum分布式事务和两阶段提交协议

Outline

GREENPLUM DATABASE

- 事务实现原理和Write Ahead Logging（WAL）
- 分布式事务和两阶段提交的原理
- Greenplum两阶段提交协议的实现
- Greenplum两阶段提交协议的优化

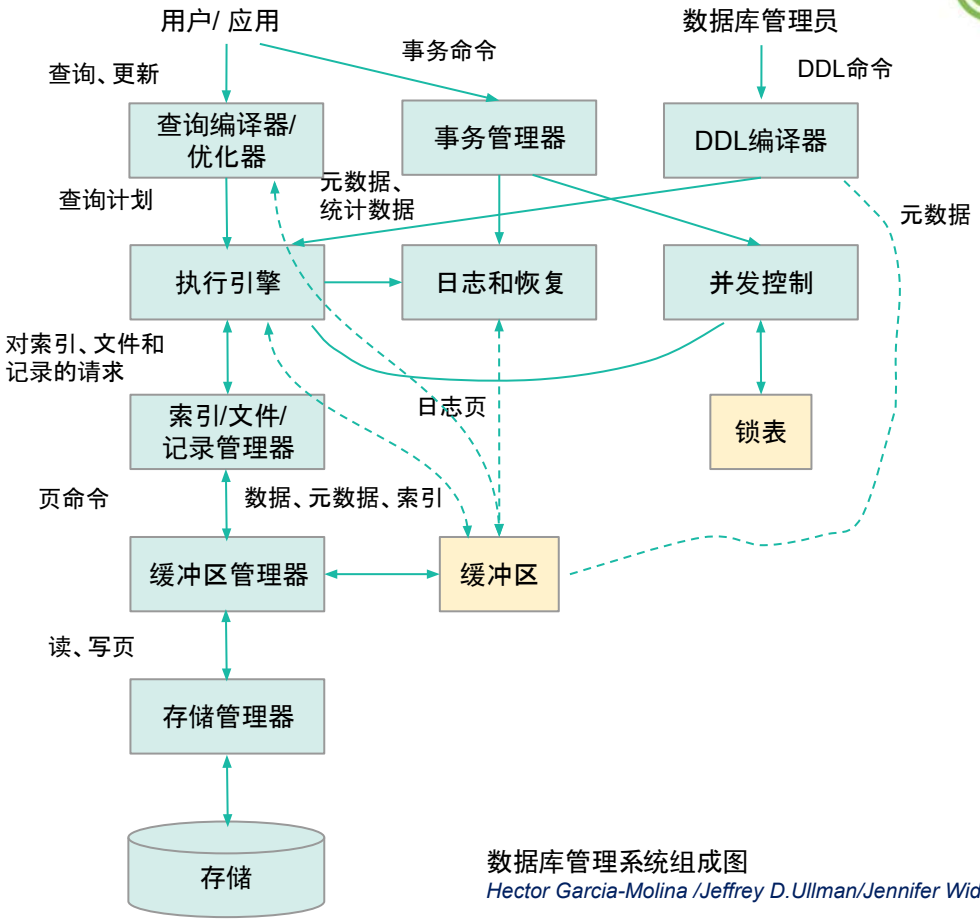
事务的属性：ACID

GREENPLUM DATABASE

属性	含义	数据库系统的实现
Atomic 原子性	事务中的操作要么全部正确执行，要么完全不执行。	Write Ahead Logging，分布式事务：两阶段提交协议
Consistency 一致性	数据库系统必须保证事务的执行使得数据库从一个一致性状态转移到另一个一致性状态。（满足完整性约束）	实现对A、I、D三个属性的支持
Isolation 隔离性	多个事务并发地执行，对每个事务来说，它并不会感知系统中有其他事务在同时执行。	多版本并发控制Multi-Version Concurrency Control、两阶段加锁（Two Phase Locking, 2PL）、乐观并发控制（OCC）
Durability 持久性	一个事务在提交之后，该事务对数据库的改变是持久的。	Write Ahead Logging + 存储管理

Jim Gray于1981年VLDB描述了事务的原子性、一致性和持久性，在此基础上，Haerder和Reuter在1983年中提出了事务的隔离性并提出术语“ACID”，自此，事务的ACID四个性质成为业内标准术语

Disk-Oriented DBMS Components



数据库管理系统组成图

Hector Garcia-Molina /Jeffrey D.Ullman/Jennifer Widom《数据库系统实现》

存储介质的类型

- **Volatile storage** 易失性存储器

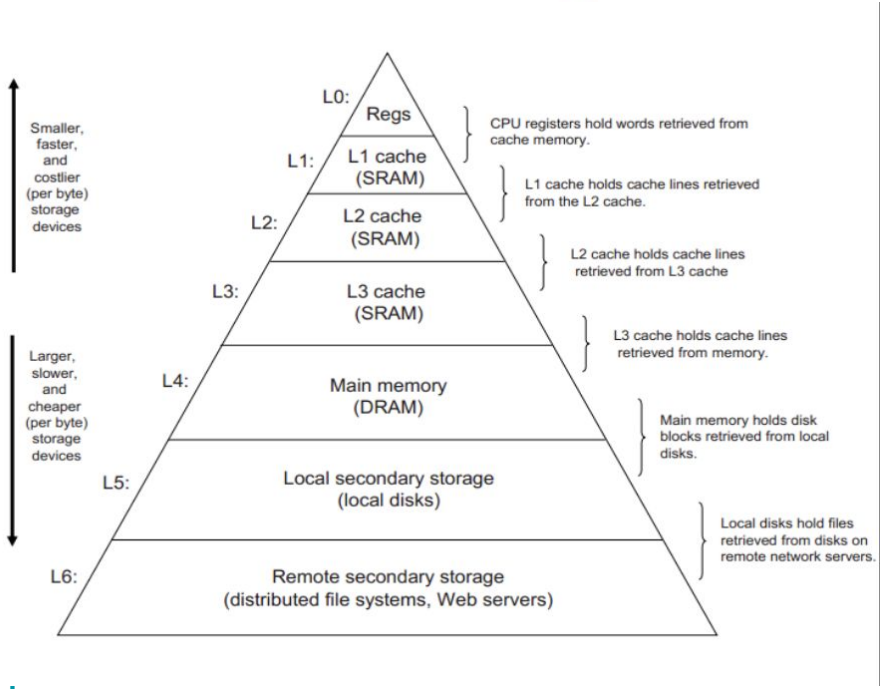
  DRAM, Cache, Register

- **Non-volatile storage** 非易失性存储器

  Disk, SSD, NVM

- **Stable stage** 稳定存储器

  theoretically *never* cannot be guaranteed



图片来源: Power consumption estimation using in-memory database computation

不同存储介质的访问时间

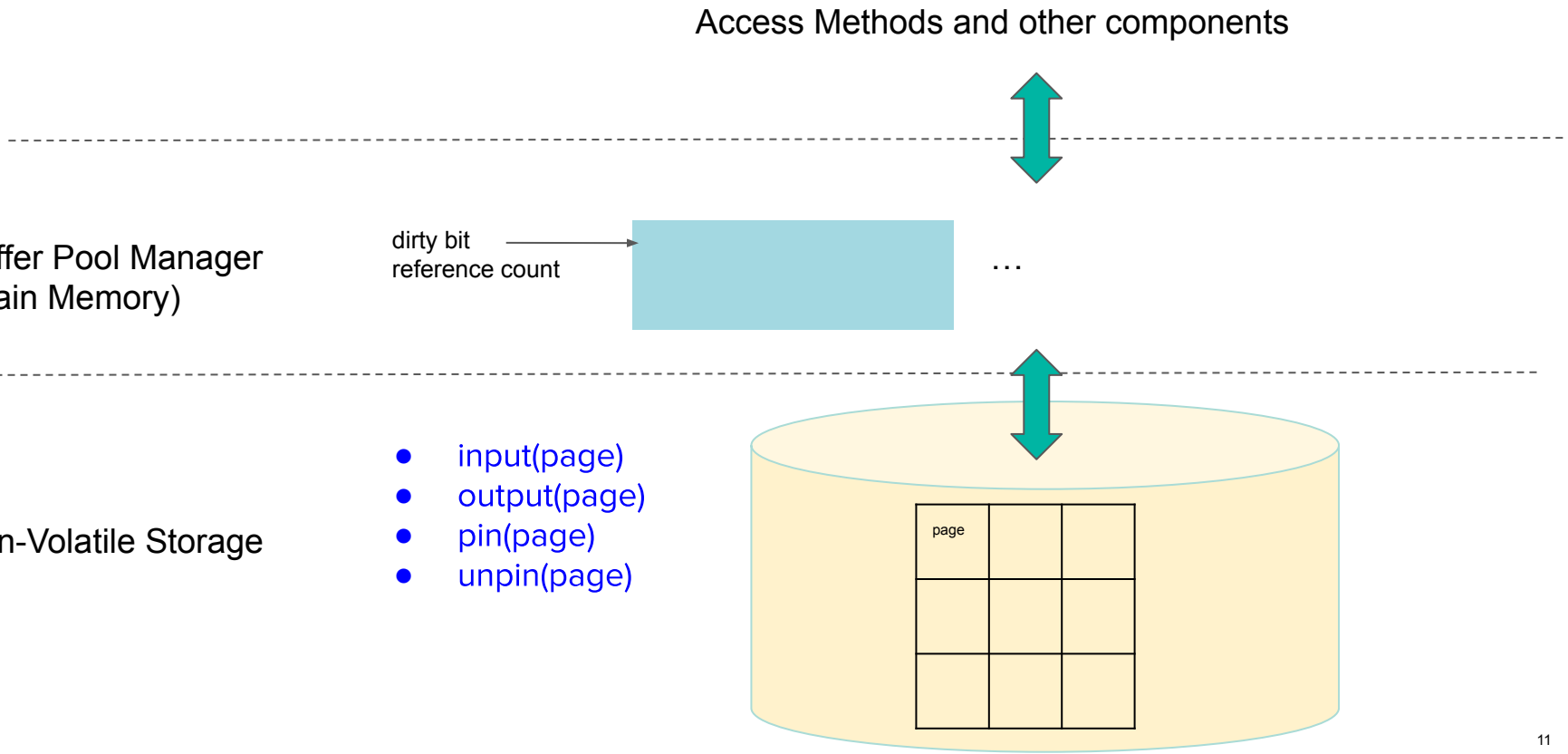
GREENPLUM DATABASE

Table 2.2 Example Time Scale of System Latencies

Event	Latency	Scaled
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access (DRAM, from CPU)	120 ns	6 min
Solid-state disk I/O (flash memory)	50–150 µs	2–6 days
Rotational disk I/O	1–10 ms	1–12 months
Internet: San Francisco to New York	40 ms	4 years
Internet: San Francisco to United Kingdom	81 ms	8 years
Internet: San Francisco to Australia	183 ms	19 years
TCP packet retransmit	1–3 s	105–317 years
OS virtualization system reboot	4 s	423 years
SCSI command time-out	30 s	3 millennia
Hardware (HW) virtualization system reboot	40 s	4 millennia
Physical system reboot	5 m	32 millennia

图片来源:Systems Performance: Enterprise and the Cloud，中译本《性能之巅》，作者Brendan Gregg

缓冲区Buffer Pool



缓冲区管理策略Buffer Management Policy

GREENPLUM DATABASE

- **Force / No-Force**

  Force: 事务提交时，所修改的页面**必须强制**刷回到持久存储中

  No-Force: 事务提交时，所修改的页面**不需要强制**刷回到持久存储中

- **Steal / No-Steal**

  Steal: **允许**Buffer Pool里未提交事务所修改的脏页刷回到持久存储

  No-steal: **不允许**Buffer Pool里未提交事务所修改的脏页刷到持久存储中

缓冲区管理策略

GREENPLUM DATABASE

- **Force策略的问题**
  对持久存储器进行频繁的随机写操作，性能下降。

- **No-Steal策略的问题**
  不允许未提交事务的脏页换出，系统的并发量不高。

- **No-Force / Steal 有更好的性能，但是怎么保证事务的原子性和持久性？**
  - **No-Force:** 事务提交，所修改的数据页没有刷回至持久存储，如果发生断电或者系统崩溃。
  - **Steal:** Buffer Pool中未提交的事务所修改的脏页刷回到持久存储，如果发生断电或者系统崩溃。

Solution: Logging

GREENPLUM DATABASE

- **No-Force → Redo Log**

  事务提交时，数据页不需要刷回持久存储，为了保证持久性，先把Redo Log写入日志文件。Redo log记录修改数据对象的新值(After Image, AFIM)

- **Steal → Undo Log**

  允许Buffer Pool未提交事务所修改的脏页刷回到持久存储，为了保证原子性，先把Undo Log写入日志文件。Undo Log记录修改数据对象的旧值(Before Image, BFIM)

缓冲区管理策略和事务恢复的关系

	Force	No-Force
Steal	Undo / No-Redo	Undo + Redo (performance: fastest recovery: slowest)
No-steal	No-Undo / No-Redo (performance:slowest recovery:fastest)	Redo / No-Undo

Recovery without Logging:
Shadow Paging（LMDB）

假设 A=8, B=10, C =12
T1: A → 10
T2: B → 8; C → 10

日志

<START T1>
<T1, A, 8>
<START T2>
<T2, B, 10>
<COMMIT T1>
<T2, C, 12>

Undo log(记录旧值)
(Steal / Force)
恢复时，从后往前，对于未提交的事务的日志做undo操作。

日志

<START T1>
<T1, A, 10>
<START T2>
<T2, B, 8>
<COMMIT T1>
<T2, C, 10>

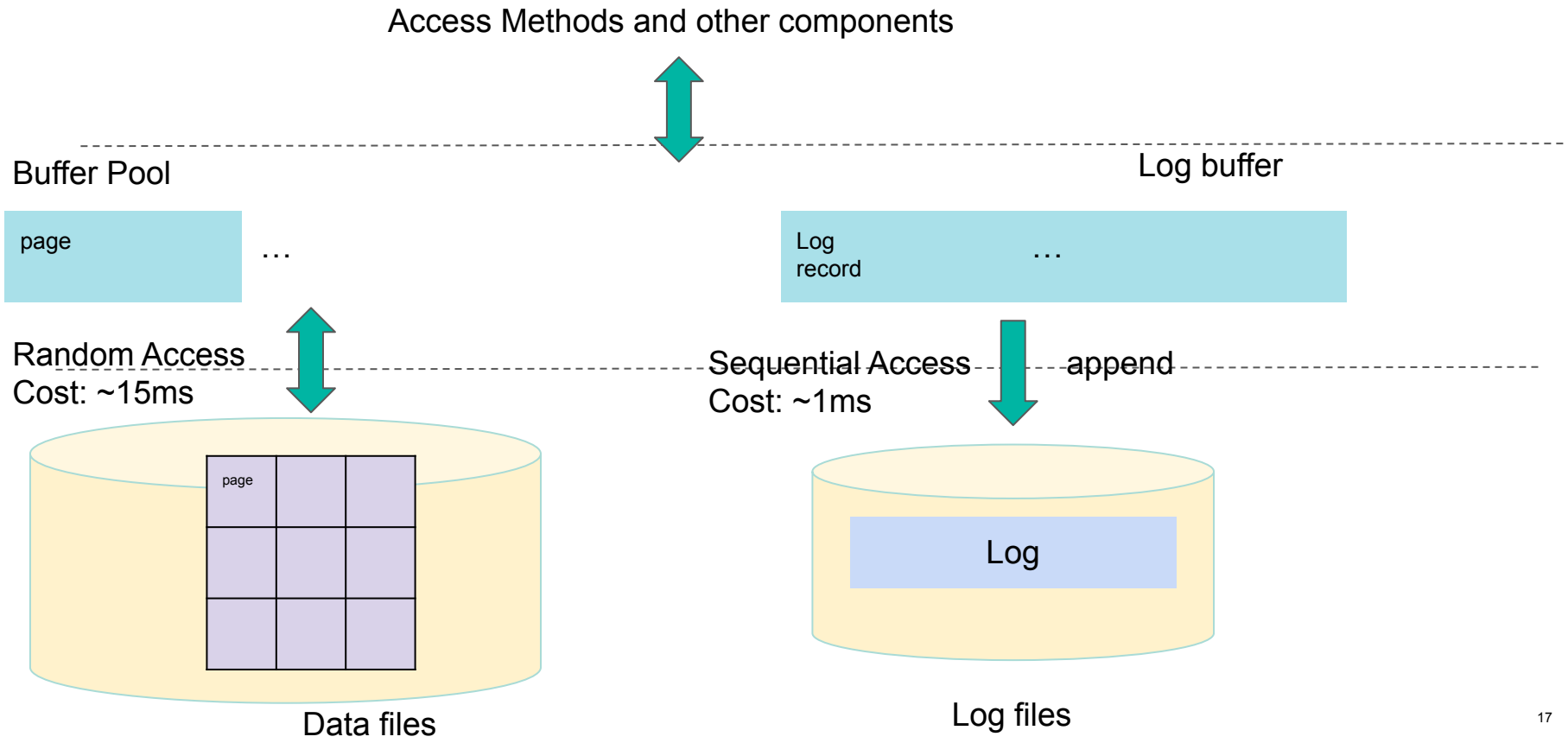
Redo log(记录新值)
(No-Steal / No-Force)
恢复时，从前往后，对提交的事务的日志做redo操作。

日志

<START T1>
<T1, A, 8, 10>
<START T2>
<T2, B, 10, 8>
<COMMIT T1>
<T2, C, 12, 10>

Undo+Redo log(记录旧值和新值)
(Steal / No-Force)
恢复时，从前往后对提交事务做redo操作，从后往前对未提交事务做undo操作 。

Buffer Pool and Log Buffer



Write Ahead Logging

GREENPLUM DATABASE

1. All log records pertaining to an updated page are written to non-volatile storage before the page itself is allowed to be overwritten in non-volatile storage.
2. A transaction is not considered to be committed until all of its log records(including its commit record) have been written to stable storage.

第一点：Steal policy。更新non-volatile storage中的页面时，必须记录undo log。保证事务的原子性。

第二点：No-Force policy。提交事务时，必须记录redo log。保证事务的持久性。

采用WAL协议的恢复算法: Dr. C. Mohan ARIES: Algorithms for Recovery and Isolation Exploiting Semantics, 1993, IBM DB2

PostgreSQL和Greenplum采用的策略

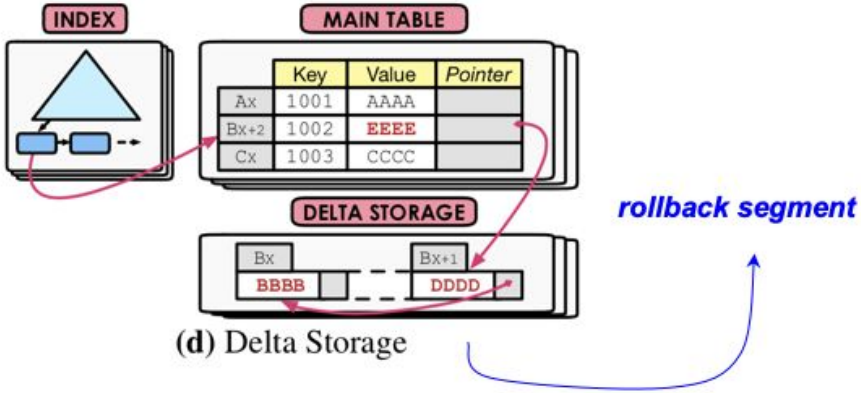
- Steal + No-force
- redo log，没有undo log，事务回滚不需要做undo操作
  - PG采用的是MVCC，更新操作不是in-place update，而是重新创建tuple，可见性判断
  - Robert Haas 2018, “DO or UNDO - there is no VACUUM”: zheap, in-place update

思考：

1. MySQL同样采用MVCC，事务恢复的时候为什么需要undo log？
2. 出现新硬件（NVRAM）并不断得到广泛 应用，WAL是否适合新硬件特点？

（业内的探索：CMU, VLDB 2016, Write-Behind Logging）

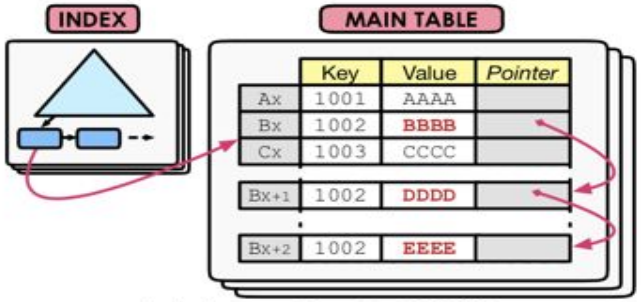
Version Storage



(d) Delta Storage

stores master versions in the main table and a sequence of delta versions are kept in a **separate delta storage**.

MySQL、Oracle



(a) Append-only (O2N)

all of the tuple versions for a table are stored in the same storage space.

PostgreSQL

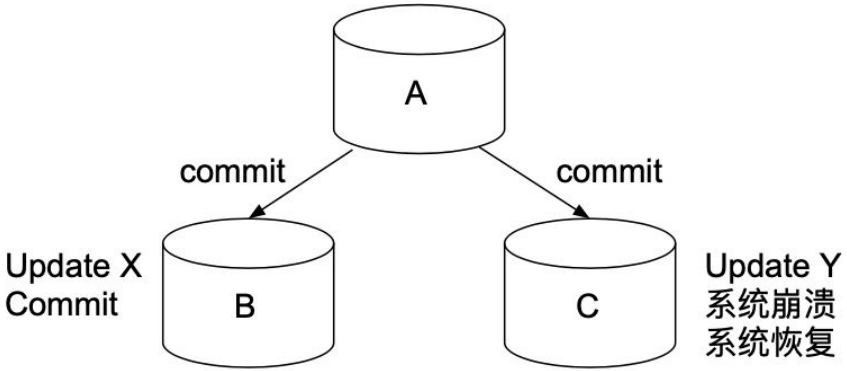
引用来源:Yingjun Wu, An Empirical Evaluation of In-Memory Multi-Version Concurrency Control, VLDB 2017

Outline

- 事务的实现原理和Write Ahead Log（WAL）
- 分布式事务和两阶段提交的原理
- Greenplum两阶段提交协议的实现
- Greenplum两阶段提交协议的优化

分布式事务

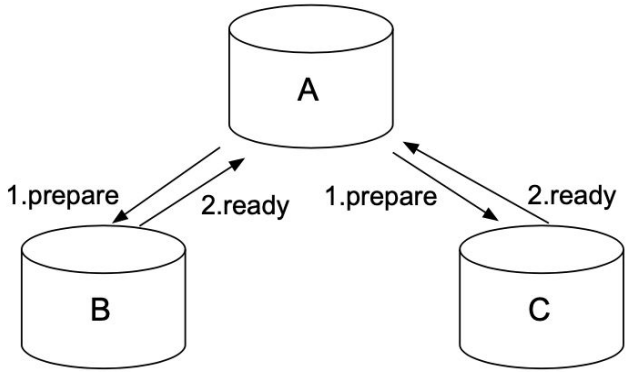
- 分布式事务，分布式环境下的事务，是一个典型的嵌套式事务，一个事务由多个工作节点的子事务组成。
- 必须保证参与分布式事务的各个场地（节点）的事务，要么全部提交，要么全部rollback，不能出现部分提交的情况。

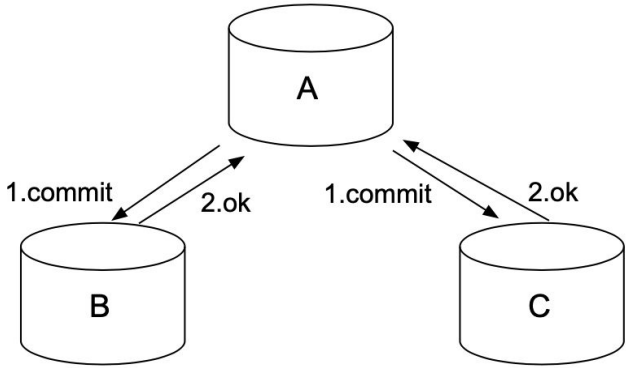


一阶段提交不能保证
分布式事务的原子性

两阶段提交协议

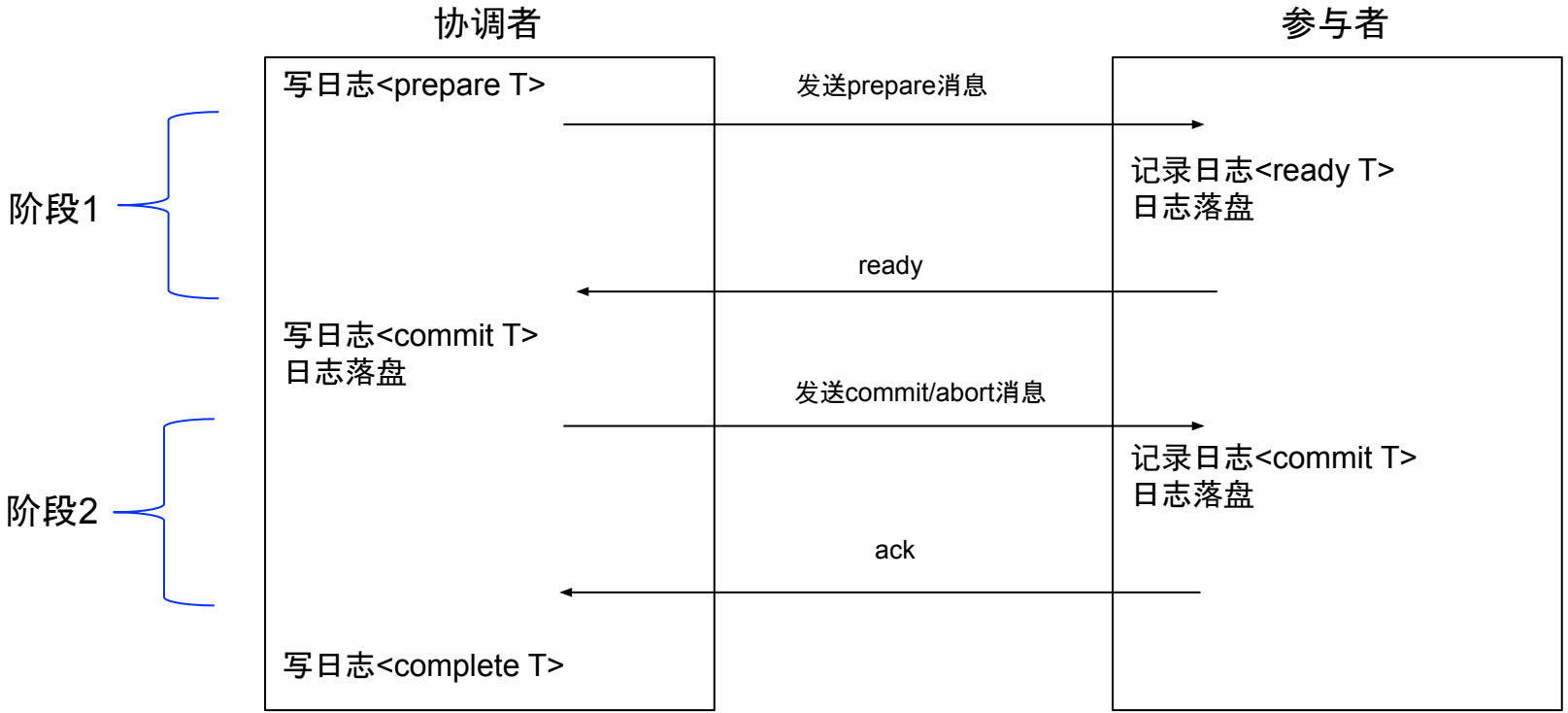
- A节点是事务的协调者（coordinator）
- B和C是事务的参与者（participant）



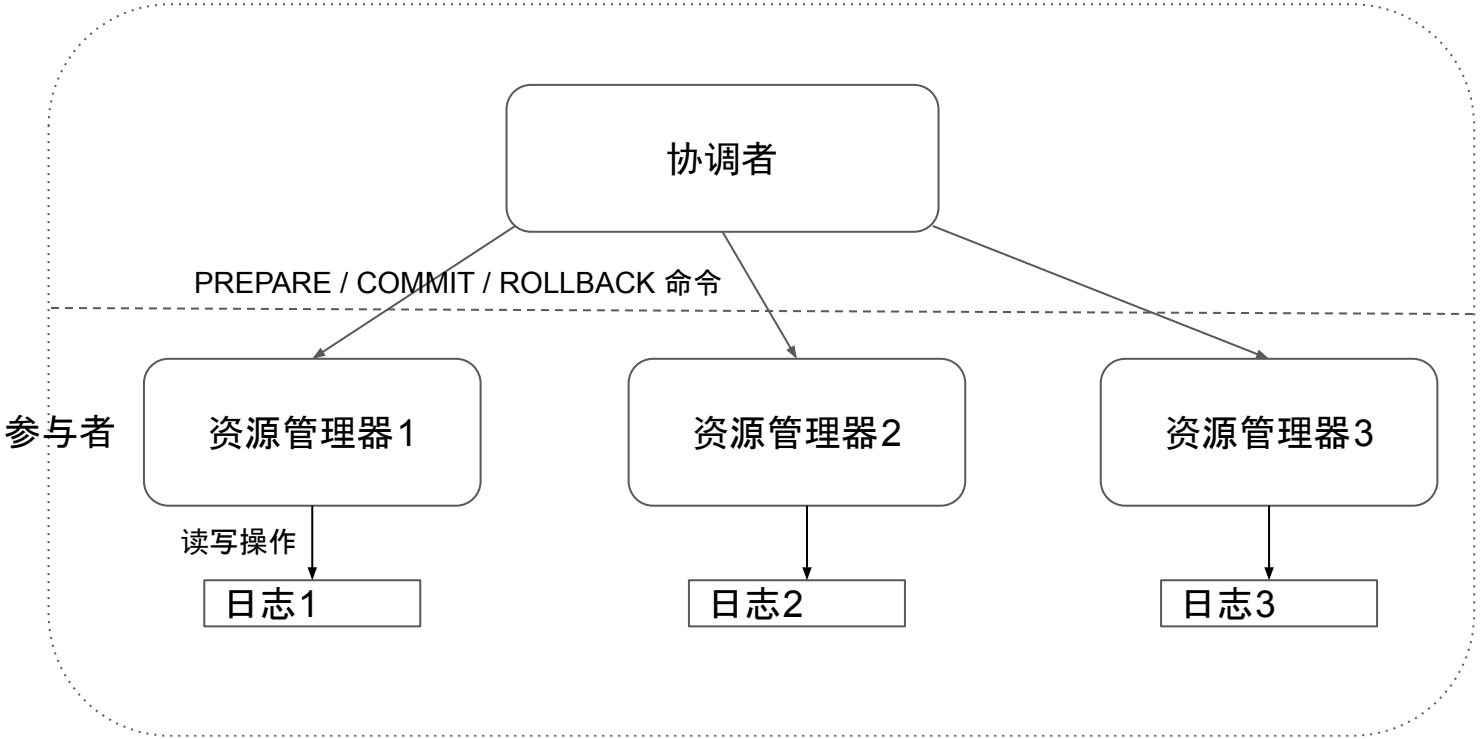


- Jim Gray等研究者在1978年提出了两阶段提交协议，用于保证分布式事务提交的原子性
- 可以用于单机集中式系统，由事务管理器协调多个资源管理器；也可以用于分布式系统，由一个全局的事务管理器协调各个子系统的局部事务管理器完成两阶段提交
- 广泛应用于商业分布式数据库

两阶段提交与日志操作


协调者
参与者
写日志<prepare T>
发送prepare消息
记录日志<ready T>
日志落盘
阶段1
ready
写日志<commit T>
日志落盘
发送commit/abort消息
记录日志<commit T>
日志落盘
阶段2
ack
写日志<complete T>

2PC同样可以应用在单机系统上



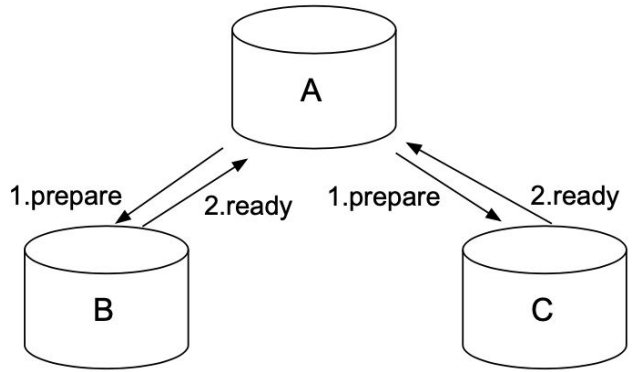
两阶段提交协议需要处理的故障

1. 参与者故障

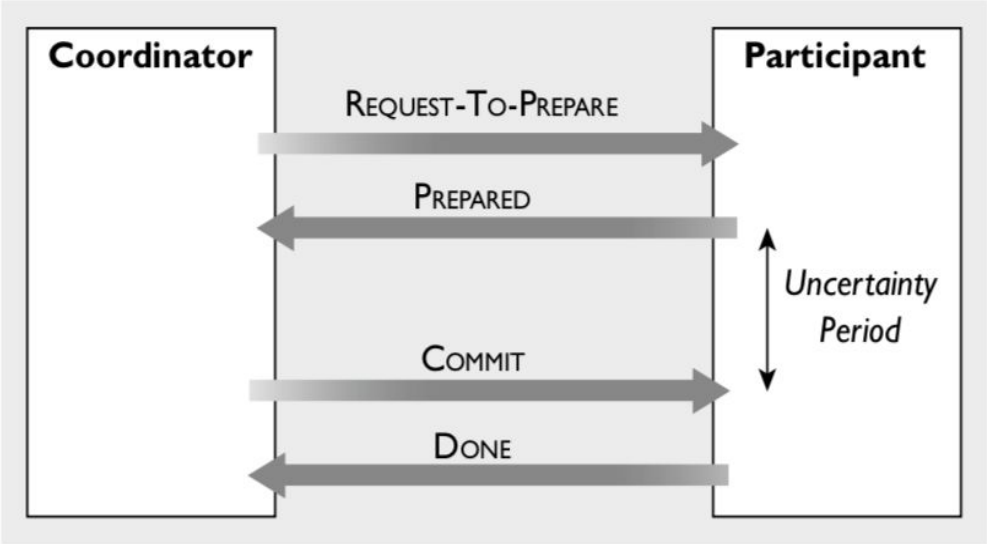
参与者恢复后，根据日志 记录来决定重做或者撤 销事务T，是否有<Ready T>记录？是否有<Commit T>或者<Rollback T>，如果没有，可以 询问参与者。

2. 协调者故障

如果协调者发生故障，参与者必 须决定提交或者撤 销事务，在某些情况下，参与者并不知道是否提交事 务，所以必须等协调者从失败中恢复。



两阶段提交存在的问题



1. Uncertainty Period
2. Blocking

图片来源：Bernstein的著作Principles of Transaction Processing

Outline

- 事务的实现原理和Write Ahead Log（WAL）
- 分布式事务和两阶段提交的原理
- Greenplum两阶段提交协议的实现
- Greenplum两阶段提交协议的优化

PostgreSQL的两阶段提交

GREENPLUM DATABASE

```
postgres=# BEGIN;
BEGIN
postgres=# INSERT INTO TEST VALUES(1),(2),(3);
INSERT 0 3
postgres=# SELECT * FROM TEST;
 c1
----
  1
  2
  3
(3 rows)

postgres=# PREPARE TRANSACTION 'T1';
PREPARE TRANSACTION
postgres=# SELECT * FROM TEST;
 c1
----
(0 rows)
```

```
postgres=# SELECT * FROM pg_prepared_xacts;
 transaction | gid |           prepared            | owner | database
-------------+-----+-------------------------------+-------+----------
         535 | T1  | 2020-01-02 17:32:18.810122+08 | linw  |
postgres
(1 row)

postgres=# COMMIT PREPARED 'T1';
COMMIT PREPARED
postgres=# SELECT * FROM TEST;
 c1
----
  1
  2
  3
(3 rows)
```

- **PREPARE TRANSACTION**
- **COMMIT PREPARED**
- **ROLLBACK PREPARED**

问题 1：协调者向参与者发prepare之后，参与者完成prepare相应操作，在发送ready之前，会把日志落盘。那参与者申请的锁会不会释放？

```
postgres=# begin ;
BEGIN
postgres=*# update t1 set c1 = 14 where c1 =15 ;
UPDATE 1
postgres=*# prepare transaction 't1';
PREPARE TRANSACTION
```

执行完这句后，select * from pg_locks，会观察到，这个事务申请的RowExclusive锁还在pg_lock里，

在PG里，执行完PREPARE语句之后，此时把数据库停掉（或者杀掉所有数据库进程）再启动起来，会发现pg_locks里，prepared事务所申请的还在pg_lock表里。

问题2: 既然pg_locks是一个内存的数据结构，记录各个backend进程申请的锁，那数据库重启后，为什么已经prepared事务申请的锁仍在pg_lock表呢？

prepared事务的恢复过程：

当执行prepare时候，PG会把该事务的lock信息当做prepare日志记录的一部分记录在日志文件（xlog）里。当数据库重新启动，会读这个日志文件（xlog）这条日志记录，把锁“还原”到pg_lock表里。

1. StartupXlog函数发现XLOG_XACT_PREPARE日志记录进行redo，调用函数recreateTwoPhaseFile将该日志记录中的信息放到pg_twophase目录下的文件里，每一个prepared事务对应一个文件
2. StartupXlog函数调用recoverPreparedTransaction函数读取pg_twophase目录下的文件并进行相关操作，为该事务重新获取锁。
3. 恢复成功后，删掉pg_twophase目录下的文件

《数据库系统概念》19.4.1.3节，参与者在响应prepare消息之前记录<ready T, L>

Greenplum实现分布式事务与并发控制

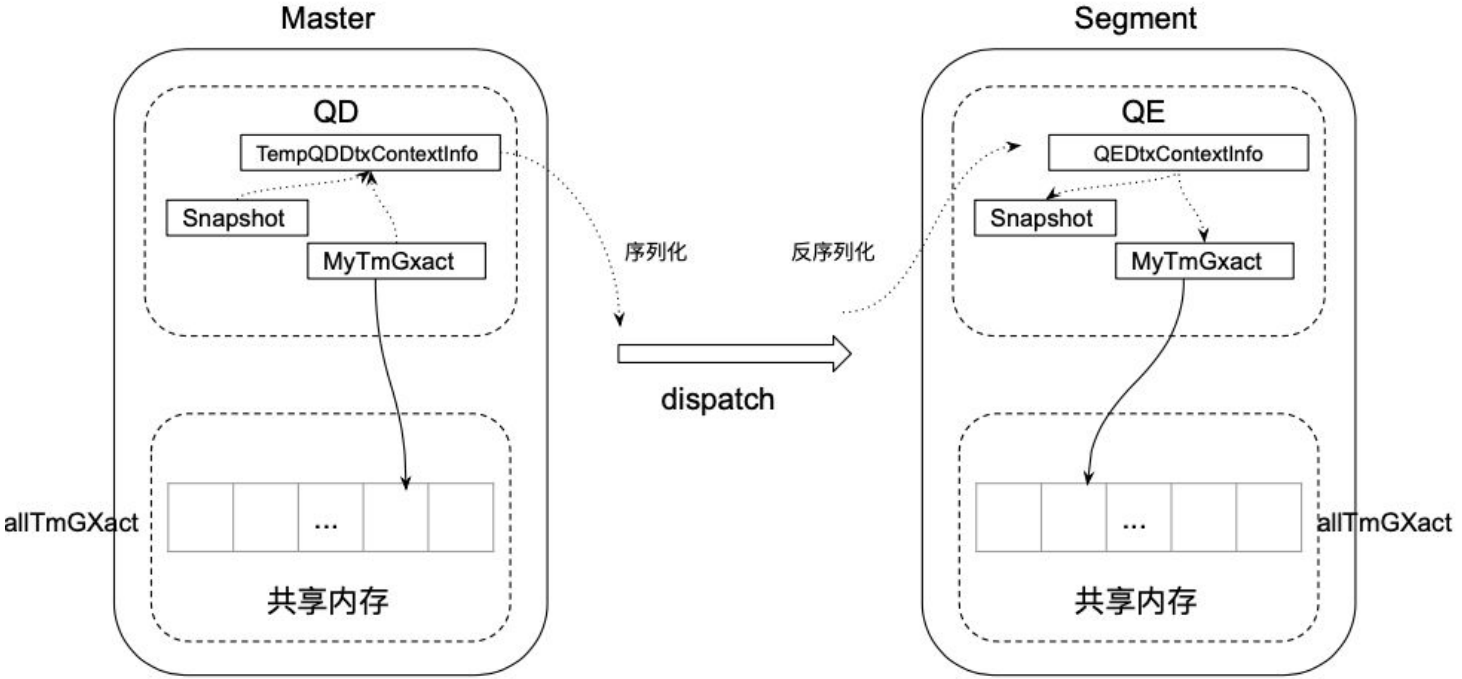
- 本地事务的管理：创建、提交、状态迁移等
- 加锁和MVCC
- 本地事务的死锁检测
- xlog、commit log(CLOG)
- 对PREPARE、COMMIT/ABORT PREPARED语句的处理

Greenplum复用PG的实现

- 分布式事务管理
  - 分布式事务的创建、状态迁移等
  - QD向QE发起两阶段提交
- 分布式快照
  - QD向QE发送全局快照信息
  - Writer QE和Reader QE共享本地快照信息
- distributed log：分布式事务提交日志
  - 用于判断分布式事务是否提交，作用和PG的commit log类似，基于simple LRU实现
- 分布式死锁检测

Greenplum在PG的基础上实现

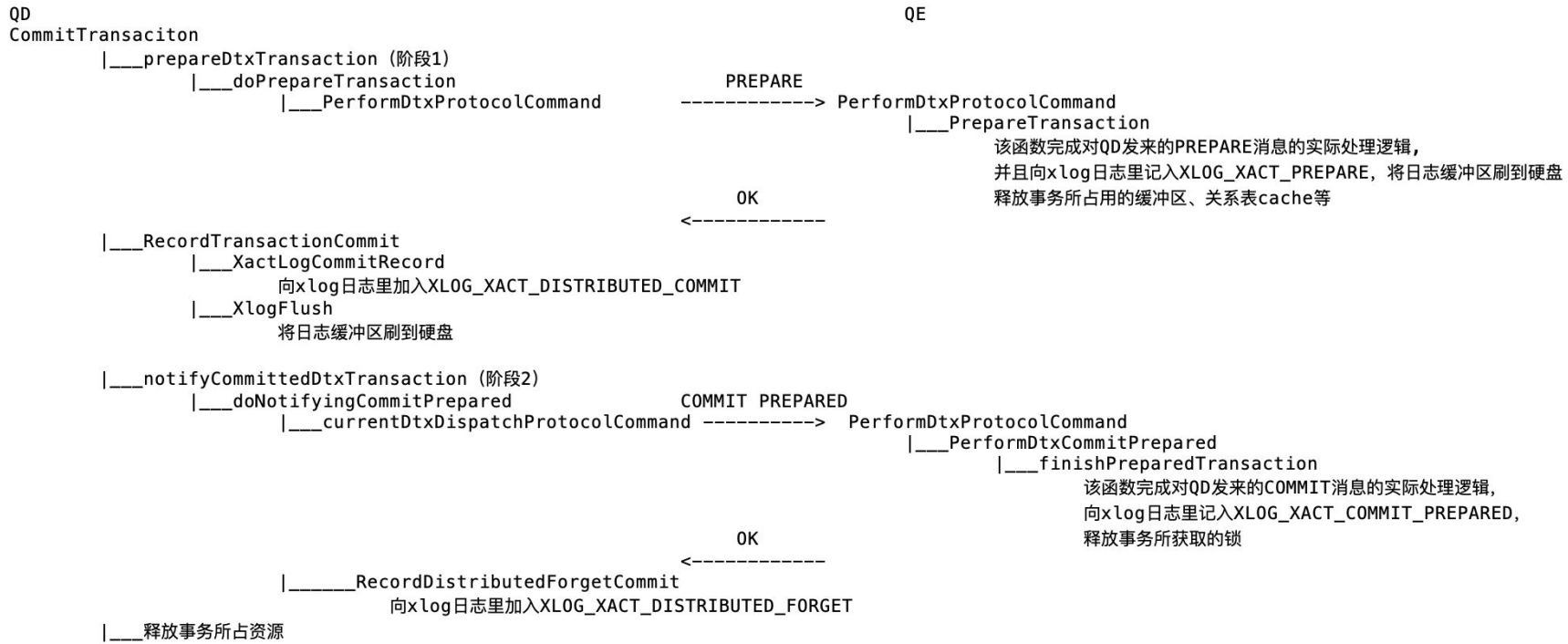
分布式事务信息在QD和QE之间的同步



MyTmGxact: TMGXACT 分布式事务结构体

- 分布式事务id
- 分布式事务管理器启动的时间戳
- 活跃分布式事务中最小的事务id，分布式快照
- session id

Greenplum的两阶段提交函数调用关系



Outline

GREENPLUM DATABASE

- 事务的实现原理和Write Ahead Log（WAL）
- 分布式事务和两阶段提交的原理
- Greenplum两阶段提交协议的实现
- Greenplum两阶段提交协议的优化

一阶段提交

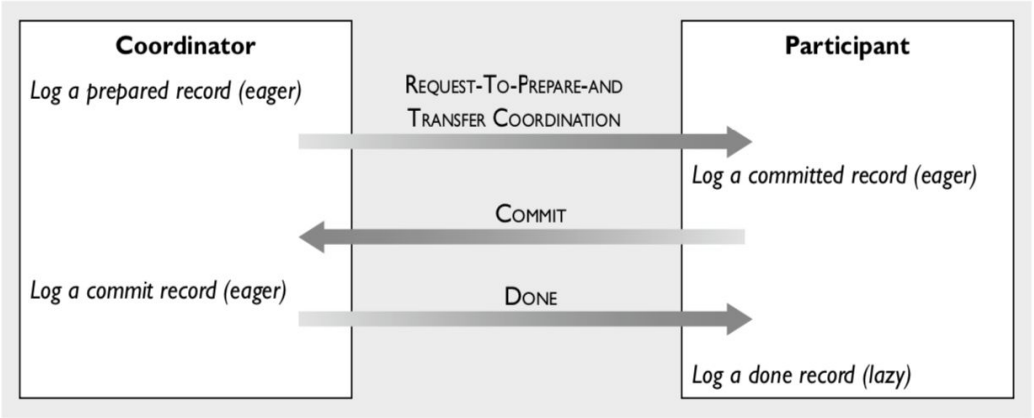
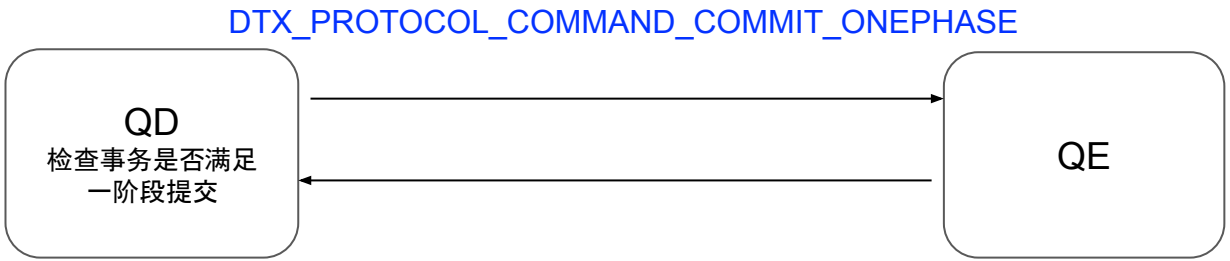


FIGURE 8.6

Transfer of Coordination Optimization. The coordinator prepares and then tells the participant to prepare and commit, and thereby become the coordinator. This saves a message over standard two-phase commit.

图片来源:Bernstein的著作Principles of Transaction Processing

Greenplum分布式事务的一阶段提交

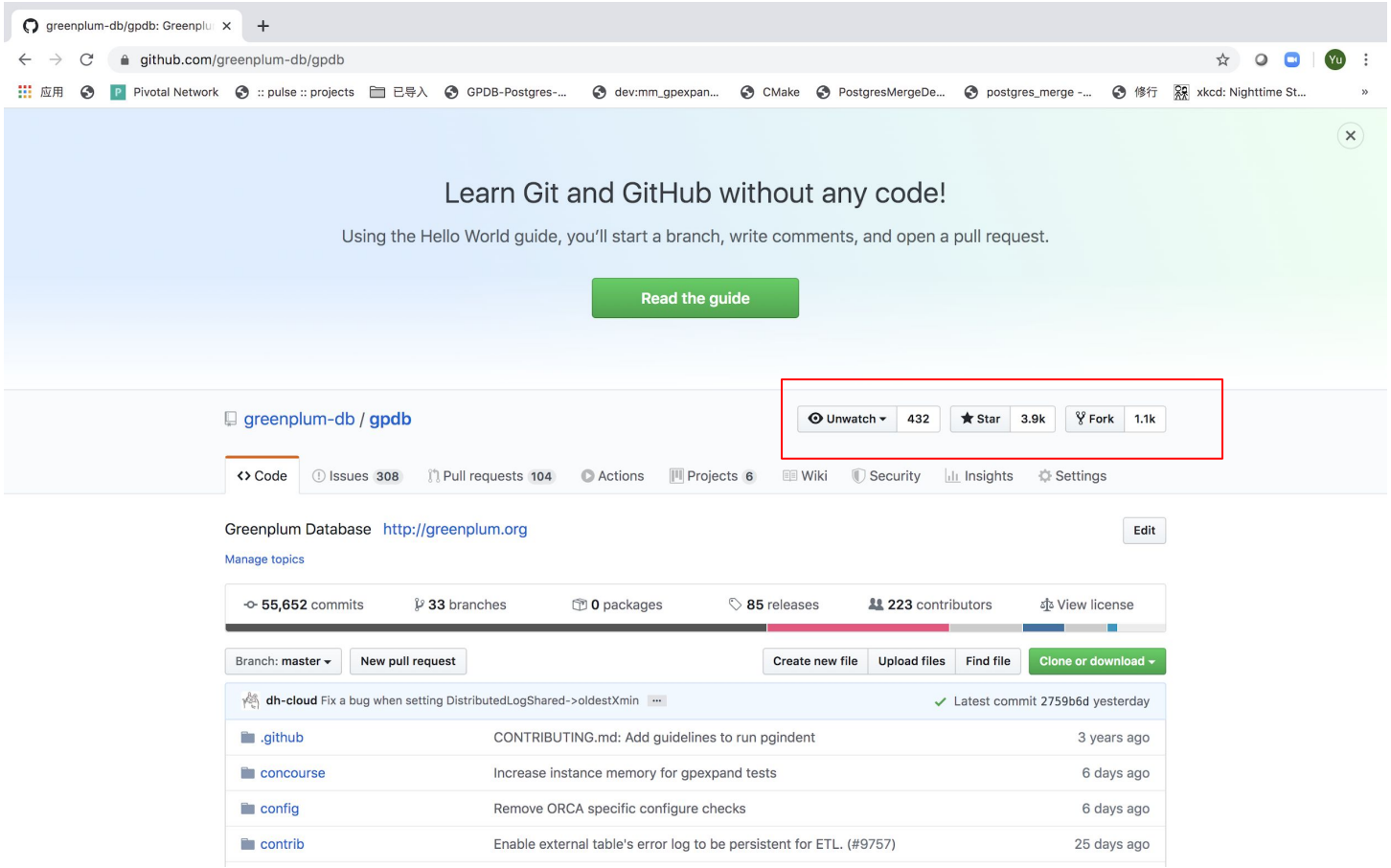


满足一阶段提交的分布式事务：

- 有写操作，参与者只有一个
- 只读事务

准备工作

从源代码开始：下载编译Greenplum源代码



GREENPLUM DATABASE®

微信技术讨论群
添加入群小助手：gp_assistant

微信公众号
技术干货、行业热点、活动预告

欢迎访问Greenplum中文社区：cn.greenplum.org

全新的问答论坛

https://cn.greenplum.org/askgp


GREENPLUM
DATABASE
Q&A