大数据管理系统与大规模数据分析 - June 3'rd, 2018

Database Concepts

https://github.com/rh01

数据仓库

ETL: Extract,Transform,Load

表

- 列 (Column)：一个属性，有明确的数据类型
  - 比如: 数值类型, 字符串类型等等
  - 必须为原子类型-不可分割，无嵌套结构
- 行 (Row)：一个记录（tuple, record）

表的数学定义

Ex. K 列的表：$\{< t_1, ..., t_k > | t_1 \in D_1, ..., t_k \in D_k\}$

每张表就是一个集合 $\{< t_1, ..., t_k >\}$

每个元素 t_j 都属于其对应类型的所有可能取值的集合 D_j，比如 $t_k \in D_k$

Schema vs. Instance

- Schema $\longrightarrow$ 列的类型（只需要定义一次）
- Instance $\longrightarrow$ 列的值

Key

Key 是特殊的列，取值是唯一的, 唯一确定一条记录. 主要有以下几种 Key

- 主键（Primary Key）$\longrightarrow$ 唯一确定本表的一条记录，比如主键 id
- 外键（Foreign Key）$\longrightarrow$ 是另一张表的主键,用于唯一确定另一张表的某个记录,比如选课表的外键 course_id 就是课程表的主键

SQL

SQL Create Table

简单创建表的 SQL 语句接口为:

```
create table 表名 (
    列名 类型,
    列名 类型,
    列名 类型,
```

```
    ......
);
```

指定某些列为主键的创建表的 SQL 为

```
CREATE TABLE 表名 (
    列名 类型,
    列名 类型,
    列名 类型,
    ......,
    PRINMARY KEY(列1,列2,...)
);
```

指定某些列为主键，且引用另外一张表 (表 #) 的主键 (列 #) 为外键创建表的 SQL 语句接口为：

```
CREATE TABLE 表名 (
    列名 类型,
    列名 类型,
    列名 类型,
    列3  类型,
    ......,
    PRINMARY KEY(列1,列2,...),
    FOREIGN KEY(列3) REFERENCE 表#(列#)
);
```

SQL INSERT

插入数据的 SQL 语句接口如下，表示对表的指定列插入数据

```
INSERT INTO 表名(列名1,列名2,...)
VALUES (值1,值2,...),(值11,值12,...);
```

SQL Delete

从表中删除满足特定条件的数据，SQL 如下：

```
DELETE FROM 表名 WHERE 条件
```

SQL UPDATE

更新满足特定条件的数据，SQL 结构如下：

```
UPDATE 表名 SET 列=值 WHERE 条件
```

主要的关系运算

选择操作 (SELECT, 数学符号 σ)，投影操作（PROJECT，数学符号 π）, 连接操作（JOIN，数学符号 $\bowtie$）

选择操作

关系代数表示 $\longrightarrow \sigma_{\mathbf{condition\_col}}(\mathbf{TABLE})$

从一个表中提取一些行的 SQL 表示：

```
SELECT * FROM 表名 WHERE condition_col满足一定的条件
```

投影操作

从一个表中提取一些列，关系代数为：$\pi_{列1,列2,...}\mathbf{TABLE}$；SQL 格式为

```
SELECT 列名,列名,... FROM 表名
```

SQL 表达选择 + 投影

从一个表中提取满足特定条件的一些列，关系代数为：$\pi_{列1,列2,...}(\sigma_{\mathbf{condition\_col}}(\mathbf{TABLE}))$；SQL 格式为

```
SELECT 列名,列名,... FROM 表名 WHERE 条件
```

连接操作 **JOIN**

Equi-join（等值连接）, 已知两个表 R 和 S，R 表的 a 列和 S 表的 b 列, 以 $R.a = S.b$ 为条件的连接 $\longrightarrow$ 找到两个表中互相匹配的记录.

关系代数表示：$\mathbf{R} \bowtie_{\mathbf{R.a=S.b}} \mathbf{S}$

连接操作往往发生在两个表或多表之间关联查询，Join 发生在 Foreign Key 与 Primary Key 之间。

Ex. 输出每个学生所选的课程，有三张表，分别为选课表，学生表，课程表，那么 SQL 语句为：

```
SELECT Student.Name, Course.Name
FROM Student, Course, TakeCourse
WHERE TakeCourse.CourseID = Course.ID
AND TakeCourse.StudentID = Student.ID;
```

Group by: 分组统计

Ex. 统计各系 2013-2014 年入学的学生人数

ID	Name	Birthday	Gender	Major	Year	GPA
131234	张飞	1995/1/1	男	计算机	2013	85
145678	貂蝉	1996/3/3	女	经管	2014	90
129012	孙权	1994/5/5	男	法律	2012	80
121101	关羽	1994/6/6	男	计算机	2012	90
142233	赵云	1996/7/7	男	计算机	2014	95

表 1: Student 表

SQL 語句為：

```
SELECT Major, COUNT(*)
FROM Student
WHERE Year >= 2013 AND Year <= 2014
GROUP BY Major;
```

統計函數

SQL 定义的统计函数包括 SUM, COUNT, AVG, MAX, MIN

HAVING: 在 GROUP BY 的基础上选择

Ex. 统计各系 2013-2014 年入学的学生人数，过滤掉人数 <2 的系.

SQL 语句为：

```
SELECT Major, COUNT(*) AS Cnt
FROM Student
WHERE Year >= 2012 AND Year <= 2014
GROUP BY Major
HAVING Cnt >= 2;
```

ORDER BY

DESC 表示递减的顺序，ASC 表示递增的顺序，用 SQL 语句表示为：

```
...
ORDER BY Cnt DESC/ASC;
```

Conclusion

```
SELECT 列名,...,列名          投影
FROM 表,...,表                选择，连接
WHERE 条件                    选择，连接
GROUP BY 列名,...,列名        分组统计
HAVING 条件                   分组后选择
ORDER BY 列名,...,列名        结果排序
```

RDBMS 的系统架构

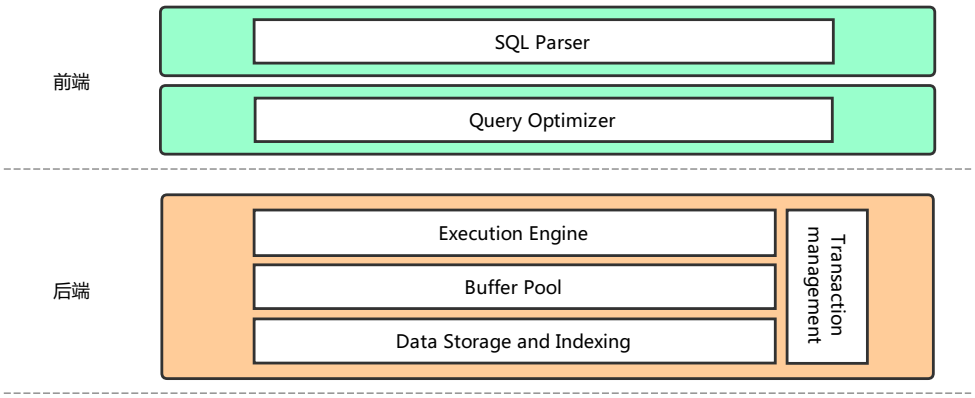

图 1: RDBMS 的系统架构.

SQL Parser: 用于 SQL 语句的解析 $\longrightarrow$ 内部表达（parse tree）, 完成以下功能：语法解析，语法检查，表名、列名、类型检查。

Query Optimizer: 主要目标是将内部表达 $\longrightarrow$ 执行方案（Query Plan），原理是通过估计 Query Plan 的运行时间和空间代价，然后选出最佳方案。

Execution Engine: 主要目标是将 Query Plan（执行方案）$\longrightarrow$SQL 语句的结果，原理是根据 query plan 完成相关的运算与操作，然后对数据的访问和对关系型运算的实现。

Buffer Pool: 主要目标是为磁盘的近期或经常访问的数据提供缓冲。

Transcation Management: 事务管理，主要目标是实现 ACID，进行写日志，加锁，保证并行事务运行的正确性。

数据存储与访问

数据库 vs. 文件系统

Ex. 从数据存储的角度对比

文件系统	数据库
存储文件 (file)	存储数据表 (table)
通用的，存储任何数据和程序	专用的，针对关系型数据进行存储
文件是无结构的，是一串字节组成的	数据表由记录组成，每个记录由多个属性组成
操作系统内核中实现	用户态程序中实现
提供基本的编程接口 (Open,Close,Read,Write)	提供 SQL 接口
数据存储在外存（硬盘）	数据存储在外存（硬盘）
根据硬盘特征，数据分成定长的数据块	根据硬盘特征，数据分成定长的数据块

表 2: 文件系统 VS 数据库

数据在硬盘上的存储

Ex. 硬盘最小存储访问单位为一个扇区：512B，**文件系统访问硬盘的单位**通常为：4KB=8 个扇区 = 文件系统的 page，**RDBMS 最小的存储单位**是database page size，往往等于 1 ～多个个文件系统的 page, 即 4KB，8KB，16KB 等等.

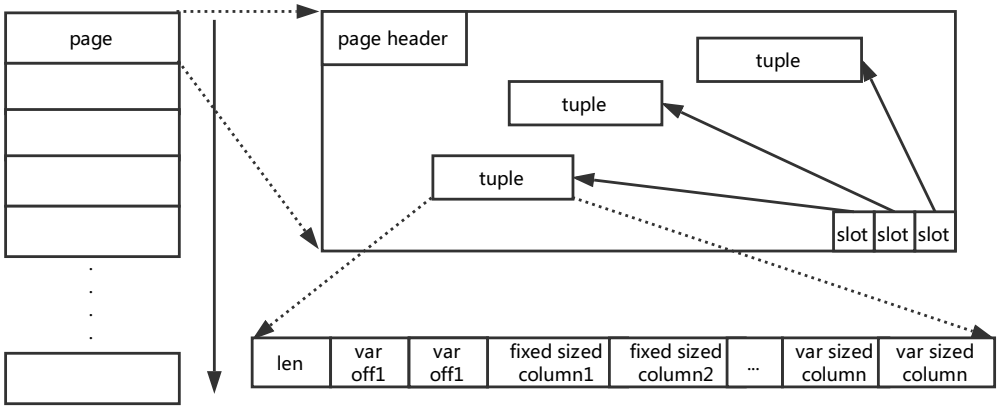

图 2: Page Tuple 的数据结构.

Ex. 下面根据一个例子来看一下，数据的表的物理存储结构是什么样的？

```
CREATE TABLE Student (
    ID integer NOT NULL, Name varchar(20), Birthday date,
    Gender enum(M, F), Major varchar(20), Year year, GPA float,
    PRINMARY KEY (ID)
);
```

tuple 数据为

ID	Name	Birthday	Gender	Major	Year	GPA
131234	张飞	1995/1/1	男	计算机	2013	85

表 3: 学生表

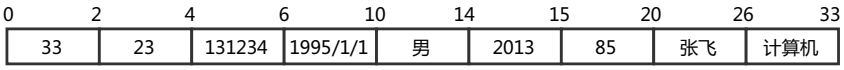

图 3: tuple 的物理结构.

Selective Data Access (有选择性的访问)

建立索引：tree-based index（有序，支持点查询和范围查询）和 hash-based index（无序，只支持点查询），

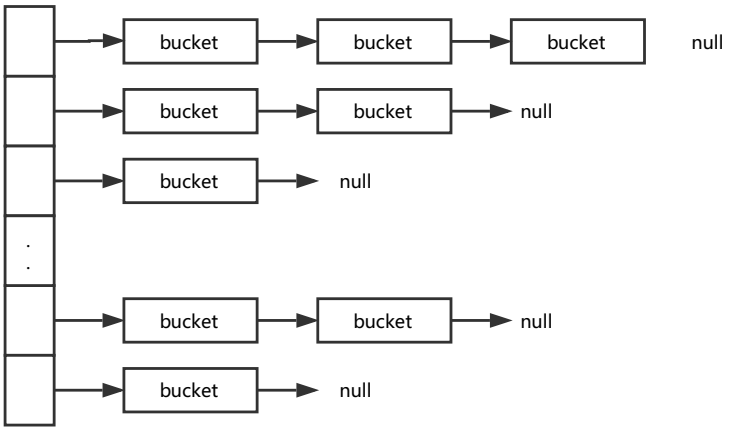

图 4: Chained Hash Table on Disk.

在硬盘中存储时，可以将 bucket 设置成 page 大小. 当 chain 上平均 bucket 数太多时，需要增大 size，这时需要重新 hashing。（存在 hash table design 可以降低 re-hashing 的代价）

B+ Tree

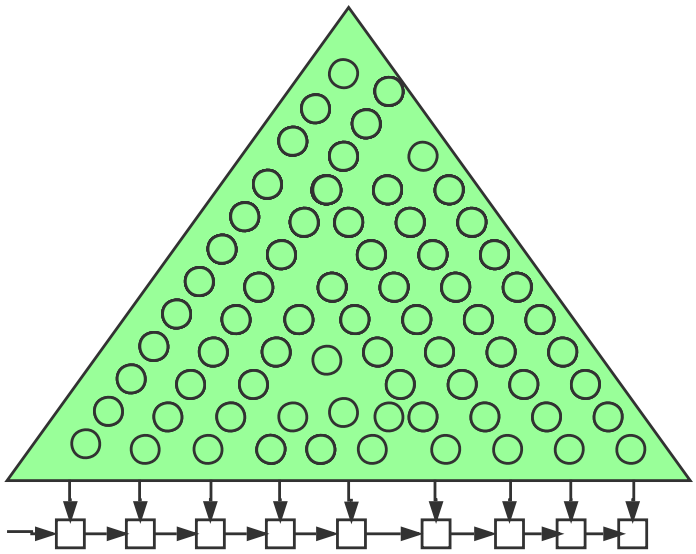

图 5: B+ Tree.

特点：每个节点是一个 Page，所有 key 存储在叶子节点，内部节点起索引作用.Keys 按照从小到大顺序排列: $key_1 < key_2 < \cdots < key_n$，叶节点自左向右也是从小到大排序，以 sibling pointer 链起来 (ptr= record ID; sibling = page ID)

B+ Tree: Search

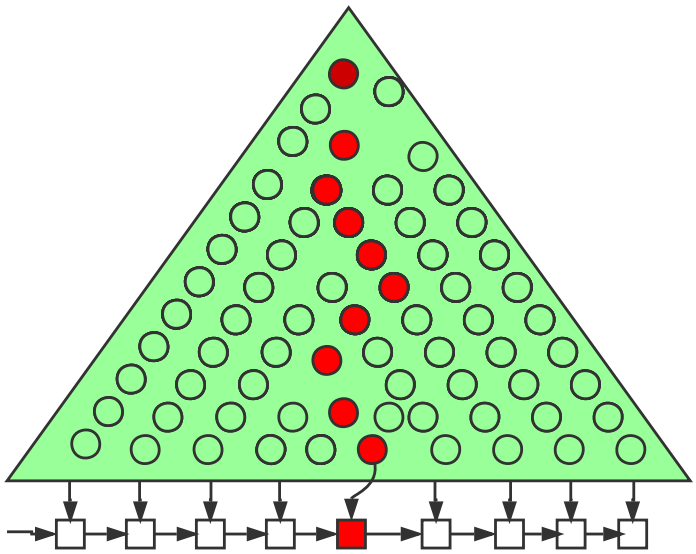

图 6: B+ Tree Search.

从根节点到叶节点 $\longrightarrow$ 每个节点中进行二分查找 (内部节点：找到包括 search key 的子树 $\longrightarrow$ 叶节点：找到匹配)

B+ Tree: Insertion

Search，然后在节点中插入，若叶节点未满，插入叶节点；若叶节点满了，node split(节点分裂)

B+ Tree: Deletion

Search，然后在节点中删除

何时进行 node merge? 原设计：当节点中 key 个数小于一半；实际实现：数据总趋势是增长的，可以只有节点为空时才 node merge，或者完全不进行 node merge

B+ Tree：Range Scan

找到起始叶结点，包括范围起始值 $\longrightarrow$ 沿着叶的链接读下一个叶结点 $\longrightarrow$ 直至遇到范围终止值.

Selective Data Access(有选择性的访问)

- 使用 index(索引)
  - Tree based：有序，支持**点查询**和**范围查询**.
  - Hash based：无序，只支持点查询
- Clustered index(主索引) 与 Secondary index(二级索引)
  - Clustered: 记录就存在 index 中，记录顺序就是 index 顺序
  - Secondary: 记录顺序不是 index 顺序，index 中存储 page ID 和 in-page tuple slot ID.

顺序索引 VS. 二级索引

顺序访问	二级索引访问
需要处理每一个记录	有选择地处理记录
顺序读每一个 page	随机读相关的 page

表 4: 顺序索引 VS. 二级索引

到底应该采用哪种方式呢？

- 由最终选中了多大比例的记录决定：**selectivity**.
- 可以根据预测的 selectivity、硬盘顺序读和随机读的性能，估算两种方式的执行时间.
- **选择时间小的方案**.
- 这就是 query optimizer 的一个任务.

Buffer Pool

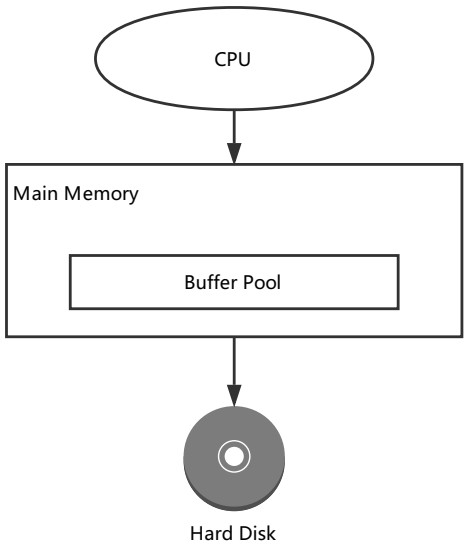

图 7: Buffer Pool-提高性能，减少 I/O.

数据访问的局部性 (locality)

- Temporal locality (时间局部性)
  - 同一个数据元素可能会在一段时间内多次被访问
  - Buffer pool
- Spatial locality (空间局部性)
  - 位置相近的数据元素可能会被一起访问
  - Page 为单位读写

Buffer Pool 的组成

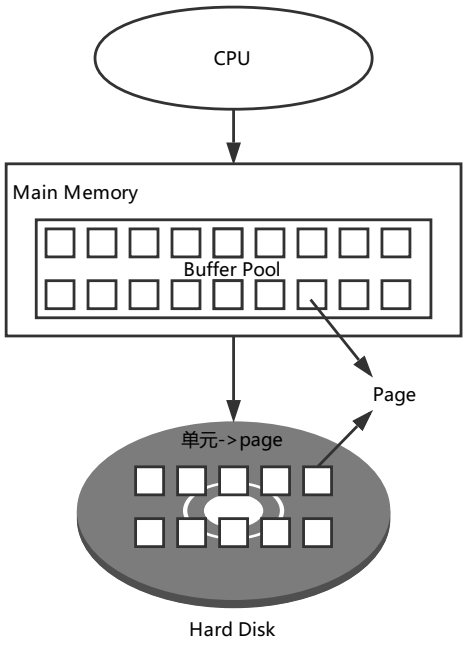

图 8: Buffer Pool 的组成.

Buffer pool 的内存空间分成 page 大小的单元（frame），每个 frame 可以缓冲硬盘中的一个 page。

访问一个 Page

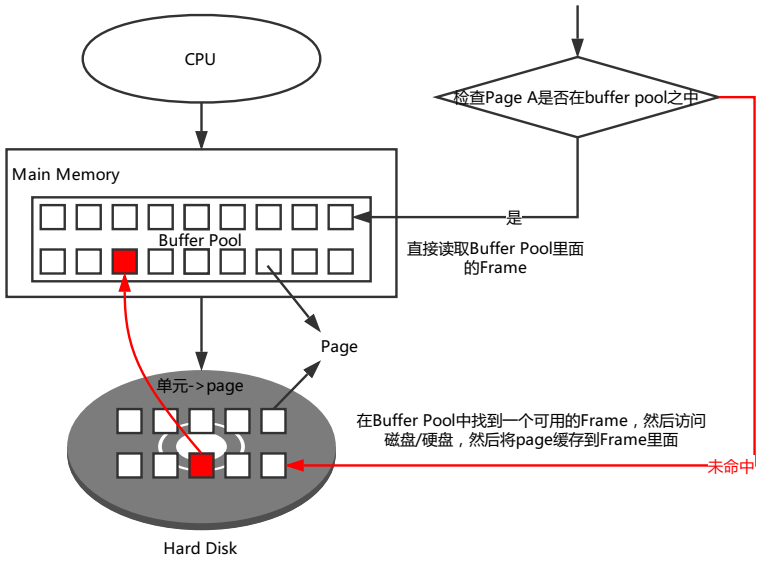

图 9: 访问一个 Page 的流程.

Replacement (替换)

如果没有空闲的 frame，那么怎么办？

- 需要找一个已缓存的 page，替换掉
  - 这个 page 被称作 Victim page
  - 如果这个 page 被修改过，那么需要写回硬盘
- 替换策略？（如何选择 Victim?）
  - 目标：尽量减少 I/O 代价，希望 Victim 在近期不可能被访问
  - 算法：通常是 LRU（Least Recently Used）的某种变形

替换策略

- Random：随机替换
- FIFO(First In First Out)：替换最老的页
- LRU (Least Recently Used)：最近最少使用

LRU

替换：找到时间戳最早（小）的页为 Victim

问题： 替换操作是 **O(N)**!

Clock 算法

数据结构：Buffer head 记录 R，取值为 0 或 1

替换，顺时针旋转，依次查看下一个页

- if (R == 1) then R=0; 继续旋转;
- if (R == 0) then 选中为 Victim

R=0 意味着在旋转了一圈的时间里，都没有被访问!

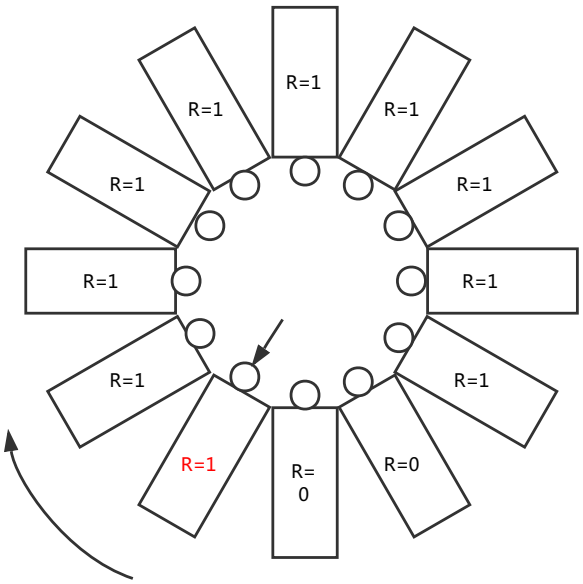

图 10: Clock 算法.

运算的实现

Operator Tree

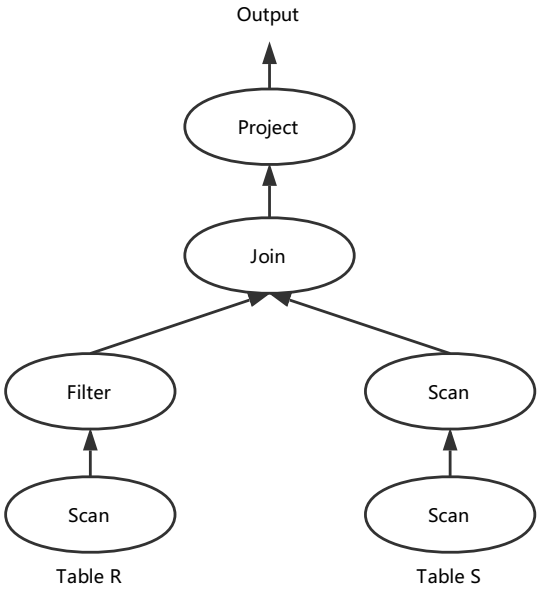

图 11: Operator Tree.

- Operator at a time
  - 完全处理一个运算再处理下一个运行，会产生大量中间结果
- Pull（Tuple at a time）
  - 每个 Operator 实现 Open, Close, GetNext 方法
  - 父节点**调用子节点的 GetNext()** 取得下一个子节点的输出
- Push：多线程
  - 子节点把输出放入中间结果缓冲，然后**通知父节点**去读

Selection & Projection

- Selection: 行的过滤
  - 支持多种数据类型：数值类型，字符串类型等
  - 实现比较操作、数学运算、逻辑运算
- Projection: 列的提取
  - Query plan 生成时，同时产生中间结果记录的 schema
  - 主要功能：从一个记录中提取属性，生成一个结果记录

Join 的实现

关系代数为 $R \bowtie_{R.a=S.b} S$.

Nested Loop Join

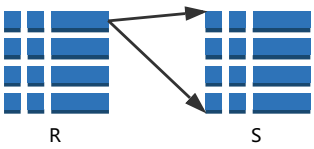

图 12: Nested Loop Join.

```
foreach tuple r $\in$ R {
    foreach tuple s  $\in$ S {
        if (r.a=s.b) output(r,s);
    }
}
```

```
R有 M_R个Page
S有 M_S个Page
每个Page有B个记录
```

- 外循环读 R
  - 读了一遍 R
- 内循环读 S
  - 对于 R 的每一个记录读所有的 S
  - 总共读了 BM_{R} 遍 S
- 总共读的 Page 数为：$M_{\mathrm{R}} + BM_{\mathrm{R}}M_{\mathrm{S}}$

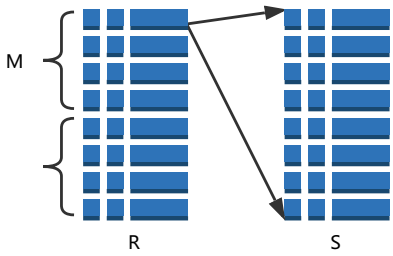

图 13: Blocked Nested Loop Join.

- 外循环读 R
  - 读了一遍 R
- 内循环读 S
  - 总共读了 $\frac{M_R}{M}$
- 总共读的 page 数：$M_{\mathrm{R}} + \frac{M_{\mathrm{S}}M_{\mathrm{R}}}{M}$

```
foreach tuple r $\in$ R {
    lookup index to look for match s in S
    if (found) output(r,s);
}
```

Hash Join

读 R 建立 hash table; 读 S 访问 hash table 找到所有的匹配;

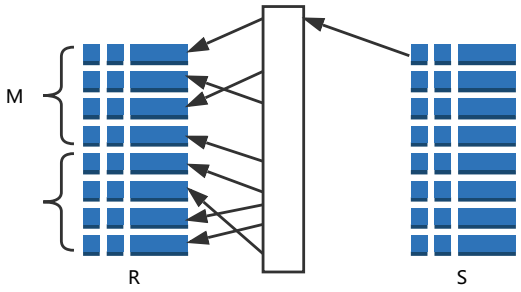

图 14: Simple Hash Join.

R 比内存大怎么办？⟶ 把 R 和 S 划分成小块 (对 R 进行 I/O partitioning; 对 S 进行 I/O partitioning)

PartitionID = hash(join key) % PartitionNumber

GRACE Hash Join

```
for (j=0; j< ParitionNumber; j++) {
    simple hash join 计算R_j \bowtie S_j;
}
```

- 对 R 进行 I/O partitioning
  - 读 M_R 个 Page，写 M_R 个 Page
- 对 S 进行 I/O partitioning
  - 读 M_S 个 Page，写 M_S 个 Page
- Simple hash join 计算所有的 $R_j \bowtie S_j$
  - 读 $M_R + M_S$ 个 Page
- 总代价（不考虑输出）
  - 读 $2M_R + 2M_S$ 个 Page，写 $M_R + M_S$ 个 Page

Sort Merge Join

Ex. 如果把 R 按照 R.a 的顺序排序 ⟶ 如果把 S 按照 S.b 的顺序排序 ⟶ 那么可以 Merge(归并) 找出所有的匹配

- 共有 $\frac{M_R}{M} + \frac{M_S}{M}$ 个 Run
- 所以需要 $log_{M-1}(\frac{M_R}{M} + \frac{M_S}{M})$) 层才能完成全部归并
- 通常代价比 Hash Join 稍差
- 当一个表已经有序的情况下，会被使用

Query Optimization(查询优化)

Ex. 依据统计信息，对以下方面进行优化。

- 访问方式：顺序扫描？索引？
- 采用哪种算法
- 多个连接的先后次序

事务处理

OLTP: Online Transaction Processing

什么叫事务？（Transaction）

一个事务可能包含多个操作，事务中的所有操作满足 ACID 性质。

SQL 語句開始一個事務：

```
# 成功的事务
BEGIN TRANSACTION;
.....
COMMIT TRANSACTION;
# 可以用rollback回卷事务
BEGIN TRANSACTION;
.....
ROLLBACK TRANSACTION;
```

ACID 特性

- **A**tomicity（原子性）
  - all or nothing
  - 要么完全执行，要么完全没有执行
- **C**onsistency（一致性）
  - 从一个正确状态转换到另一个正确状态（正确指：constraints, triggers 等）
- **I**solation（隔离性）
  - 每个事务与其它并发事务互不影响
- **D**urability（持久性）
  - Transaction commit 后，结果持久有效，crash 也不消失

(数据竞争 Data Race)

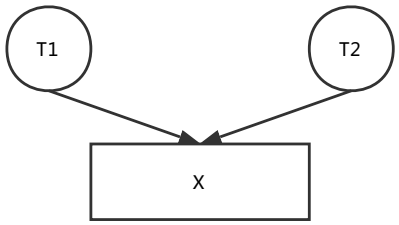

图 15: 数据竞争.

当两个并发访问都是写，或者一个读一个写时.

Schedule(调度/执行顺序)

更复杂的情况

两个 Transactions 并发访问多个共享的数据元素

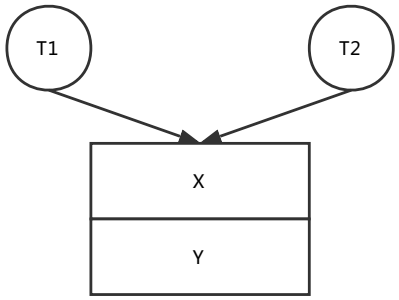

图 16: 更复杂的情况.

正确性问题

如何判断一组 Transactions 正确执行？**存在一个顺序，按照这个顺序依次串行执行这些 Transactions，得到的结果与并行执行相同**.

Serializable(可串行化)

Ex. 判断一组并行 Transactions 是否正确执行的标准: **并行執行結果 = 某个顺序的串行执行结果**.

数据冲突引起的问题

- Read uncommitted data (读脏数据) (写读)
  - 在 T2 commit 之前，T1 读了 T2 已经修改了的数据
- Unrepeatable reads(不可重复读) (读写)

  - 在 T2 commit 之前，T1 写了 T2 已经读的数据
  - 如果 T2 再次读同一个数据，那么将发现不同的值
- Overwrite uncommitted data (更新丢失) (写写)
  - 在 T2 commit 之前，T1 重写了 T2 已经修改了的数据

Isolation Level

	读脏数据（写读）	不可重复读 (读写)	更新丢失 (写写)
Serializable	no	no	no
Repeatable Read	no	no	possible
Read committed	no	possible	possible
Read committed	no	possible	possible

表 5: Isolation Level

兩種解決方案

- Pessimistic (悲观)
  - 假设：数据竞争可能经常出现
  - 防止：采用某种机制保证数据竞争不会出现
    * 如果一个 Transaction T1 可能和正在运行的其它 Transaction 有冲突，那么就让这个 T1 等待，一直等到有冲突的其它所有 Transaction 都完成为止，才开始执行。
- Optimistic (乐观)
  - 假设：数据竞争很少见
  - 检查：先执行，在提交前检查是否没有数据竞争
    * 允许所有 Transaction 都直接执行
    * 但是 Transaction 不直接修改数据，而是把修改保留起来
    * 当 Transaction 结束时，检查这些修改是否有数据竞争
      · 没有竞争，成功结束，真正修改数据
      · 有竞争，丢弃结果，重新计算

Pessimistic: 加锁

- 使用加锁协议来实现
- 对于每个事务中的 SQL 语句，数据库系统自动检测其中的读、写的数据
- 对事务中的读写数据进行加锁
- 通常采用两阶段加锁（2 Phase Locking）

2 Phase Locking

Ex. 在 Transaction 开始时，对**每个需要访问的数据**加锁 $\longrightarrow$ 如果不能加锁，就等待，直到加锁成功 $\longrightarrow$ 执行 Transaction 的内容 $\longrightarrow$ 在 Transaction commit 前，集中进行解锁 $\longrightarrow$ Commit

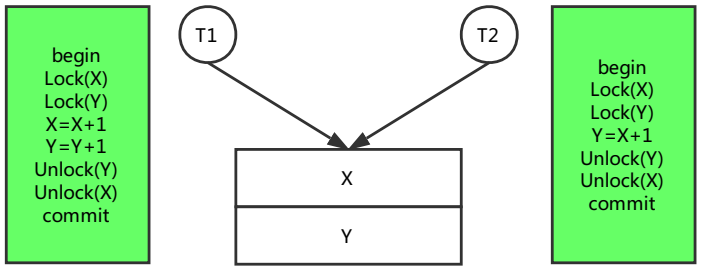

图 17: 2PL-example.

实现细节 1：读写的锁是不同的

Shared lock(S)：保护读操作,Exclusive lock(X)：保护写操作.

实现细节 2：Lock Granularity

- 锁的粒度是不同的
  - Table?
  - Record?
  - Index?
  - Leaf node?
- Intent locks
  - IS(a)：将对 a 下面更细粒度的数据元素进行读
  - IX(a)：将对 a 下面更细粒度的数据元素进行写
- 为了得到 S,IS: 所有祖先必须为 IS 或 IX
- 为了得到 X,IX: 所有祖先必须为 IX

	IS(intent shared)	IX(intent exclusive)	S(shared)	X(exclusive)
IS	√	√	√	X
IX	√	√	X	X
S	√	X	√	X
X	X	X	X	X

表 6: 锁的粒度

实现细节 3：deadlock

出现的条件：circular wait 循环等待

如何解决 deadlock 问题？

- 死锁避免

  - 规定 lock 对象的顺序
  - 按照顺序请求 lock
  - 适用于 lock 对象少的情况
- 数据库的 lock 对象很多，不适合死锁避免
- 死锁检测
  - 周期地对长期等待的 Transactions 检查是否有 circular wait
  - 如果有，那么就选择环上其中一个 Transaction abort

乐观的并发控制：不采用加锁

- 事务执行分为三个阶段
  - 读：事务开始执行，读数据到私有工作区，并在私有工作区上完成事务的处理请求，完成修改操作
  - 验证：如果事务决定提交，检查事务是否与其它事务冲突
    * 如果存在冲突，那么终止事务，清空私有工作区
    * 重试事务
  - 写：验证通过，没有发现冲突，那么把私有工作区的修改复制到数据库公共数据中
- 优点：当冲突很少时，没有加锁的开销
- 缺点：当冲突很多时，可能不断地重试，浪费大量资源，甚至无法前进

另一种并发控制方法:Snapshot Isolation

arg

- 一种 Optimistic concurrency control
- Snapshot: 一个时点的数据库数据状态
- Transaction
  - 在起始时点的 snapshot
  - 读：这个 snapshot 的数据
  - 写：先临时保存起来，在 **commit 时检查**有无冲突，有冲突就 abort
    * First writer wins

Durability (持久性) 如何实现？ Transaction commit 后，结果持久有效，crash 不消失

- 想法一
  - 在 transaction commit 时，把所有的修改都写回硬盘
  - 只有当写硬盘完成后，才 commit
- 存在的问题
  - 正确性问题：如果写多个 page，中间掉电，怎么办？Atomicity 被破坏了！
  - 性能问题：随机写硬盘，等待写完成

解决方案：WAL (Write Ahead Logging)

Transactional Logging(事务日志)

- 写操作：产生一个事务日志记录
- Commit：产生一个 commit 日志记录
  - (LSN, tID, commit)
- Abort：产生一个 abort 日志记录
  - (LSN, tID, abort)
- 日志记录被追加 (append) 到日志文件末尾
  - 日志文件是一个 append-only 的文件
  - 文件中日志按照 LSN 顺序添加

Write-Ahead Logging

写/commit 前记录日志

WAL 怎样保证 Durability

- 条件：日志是 Durable 的
- 当出现掉电时，可以根据日志发现所有写操作
  - 总是先记录意向，然后实际操作
  - 所以只有存在日志记录，相应的操作才有可能发生
- 对于 Transaction X，寻找 X 的 commit 日志记录
  - 如果找到，那么 X 已经 commit 了
  - 如果没找到，那么 X 没有完成
- 已 Commit
  - 根据日志记录，确保所有的写操作都完成了
- 没有 commit
  - 根据日志记录，对每个写操作检查和恢复原值

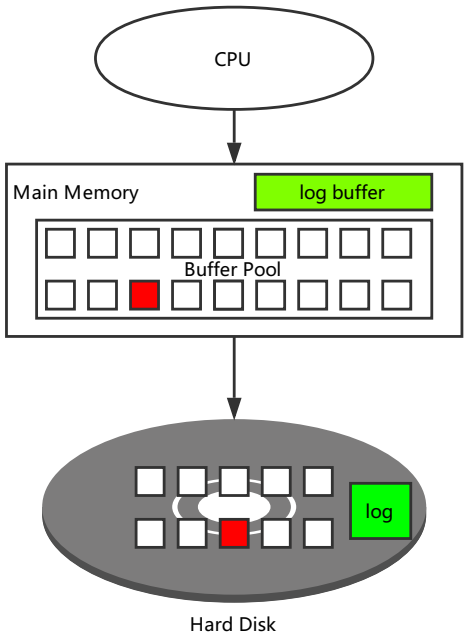

图 18: WAL.

- Log: 硬盘上日志文件
- Log buffer：在内存中分配一个缓冲区
- 日志写在 log buffer 中
- 当 commit 时 write+flush log buffer
- 存在的问题
  - Dirty page 可能被写回硬盘！
  - 掉电后，硬盘上数据已经
  - 修改，但是 log 没有记录！
- 解决方案
  - Page header 记录本 page 最新写的 LSN
  - Buffer pool 在替代写回一个 dirty page 时，必须保证 page LSN 之前的所有日志已经 flush 过了
- 保证：日志记录一定是先于修改后的数据出现在硬盘上

Checkpoint（检查点）

- 为什么要用 checkpoint?
  - 为了使崩溃恢复的时间可控
  - 如果没有 checkpoint，可能需要读整个日志，redo/undo 很多工作
- 定期执行 checkpoint
- checkpoint 的内容

– 当前活动的事务表：包括事务的最新日志的 LSN
– 当前脏页表：每个页最早的尚未写回硬盘的 LSN

Crash Recovery

图 19: Crash Recovery

ARIES 算法: 分析阶段 ⟶redo 阶段 ⟶undo 阶段

崩溃恢复：分析阶段

- 找到最后一个检查点
  – 检查点的位置记录在硬盘上一个特定文件中
  – 读这个文件，可以得知最后一个检查点的位置
- 找到日志崩溃点
  – 如果是掉电等故障，必须找到日志的崩溃点
  – 当日志是循环写时，需要从检查点扫描日志，检查每个日志页的校验码，发现**校验码出错的位置，或者 LSN 变小的位置**.
- 确定崩溃时的活跃事务和脏页
  – 最后一个检查点时的活跃事务表和脏页表
  – 正向扫描日志，遇到 commit, rollback, begin 更新事务表
    * 同时记录每个活动事务的最新 LSN
  – 遇到写更新脏页表
    * 同时记录每个页的最早尚未写回硬盘的 LSN

崩溃恢复：Redo 阶段

- 目标：把系统恢复到崩溃前瞬间的状态
- 找到所有脏页的最早的 LSN
- 从这个 LSN 向日志尾正向读日志
  – Redo 每个日志修改记录
- 对于一个日志记录
  – 如果其涉及的页不在脏页表中，那么跳过
  – 如果数据页的 LSN>= 日志的 LSN，那么跳过
    * 数据页已经包含了这个修改

  - 其它情况，修改数据页

崩溃恢复：Undo 阶段

- 目标：清除未提交的事务的修改
- 对于所有在崩溃时活跃的事务
  - 找到这个事务最新的 LSN
  - 通过反向链表，读这个事务的所有日志记录
- undo 所有未提交事务的修改
  - Undo 时，比较数据页的 LSN 和日志的 LSN
  - if (数据页 LSN>= 日志 LSN) 时，才进行 undo

介质故障的恢复

- 如果硬盘坏了，那么日志可能也损坏了
  - 无法正常恢复
- 硬件的方法：RAID（冗余盘阵列）
- 如果整个 RAID 坏了，怎么办？
- 需要定期 replicate 备份数据库
  - 备份数据库数据
  - 更频繁地备份事务日志
  - 那么就可以根据数据和日志恢复数据库状态
  - 例如：双机系统

数据仓库 vs. 事务处理

数据仓库	事务处理
少数数据分析操作	大量的并发 transactions
每个操作访问大量的数据	每个 transaction 访问很少的数据
分析操作以读为主	读写

表 7: 数据仓库 vs. 事务处理.

Star Schema：数据仓库中常见

一个很大的 fact table，多个 dimension table

OLAP

- Online Analytical Processing（联机分析处理）
- 数据仓库通常是 OLAP 的基础
  - OLAP 是在数据仓库的基础上实现的
- OLAP 的基本数据模型是多维矩阵

  - 例如，在多个 dimension 上进行 group by 操作
  - 得到的多维矩阵的每项代表一个分组，每项的值是 Fact 表上对于这个分组的聚集统计值
- 称作：Data Cube（数据立方）

行式数据存储

列式数据存储

- 数据仓库的分析查询
  - 大部分情况只涉及一个表的少数几列
  - 会读一大部分记录
- 在这种情况下，行式存储需要读很多无用的数据
- 采用列式存储可以降低读的数据量

分布式数据库

- Shared memory
  - 多芯片、多核
  - 或 Distributed shared memory
- Shared disk
  - 多机连接相同的数据存储设备
- Shared nothing
  - 普通意义上的机群系统
  - 由以太网连接多台服务器

Shared Nothing

系统架构

- 一个 coordinator 运行前端产生并行的 query plan
- 每台 worker 服务器上都有后端
- Coordinator 协调 worker 服务器执行

关键技术

- Partitioning（划分）
  - 把数据分布在多台服务器上
  - 通常采用 Horizontal partitioning
    * 把不同的记录分布在不同的服务器上
    * Hash partitioning
      · 类似 GRACE: machine ID = hash(key) % MachineNumber
    * Range partitioning

· 每台服务器负责一个 key 的区间，所有区间都不重叠

- Replication（备份）
  - 为了提高可靠性
  - 对性能的影响
    * 读？可能提高并行性
    * 写？额外代价

分布式事务

如果一个事务读写的数据分布在不同机器上

2 Phase Commit

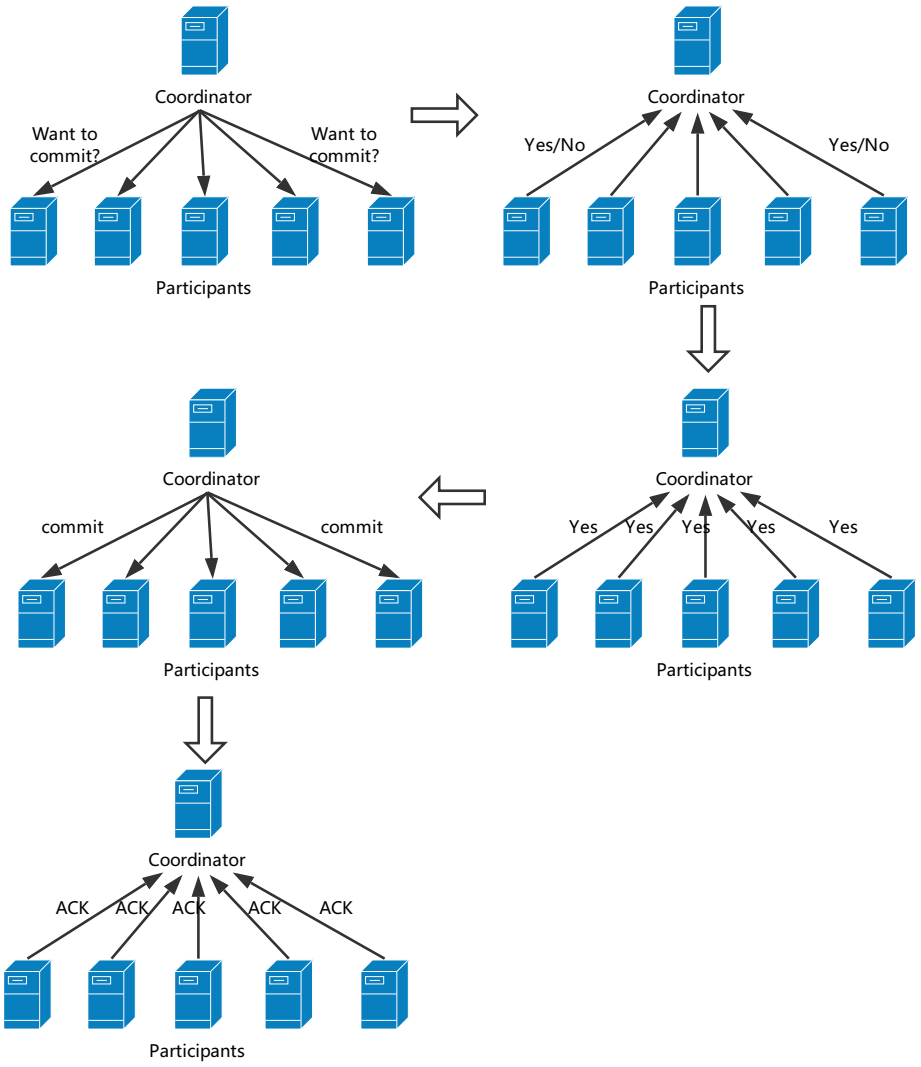

图 20: 2PL-commit

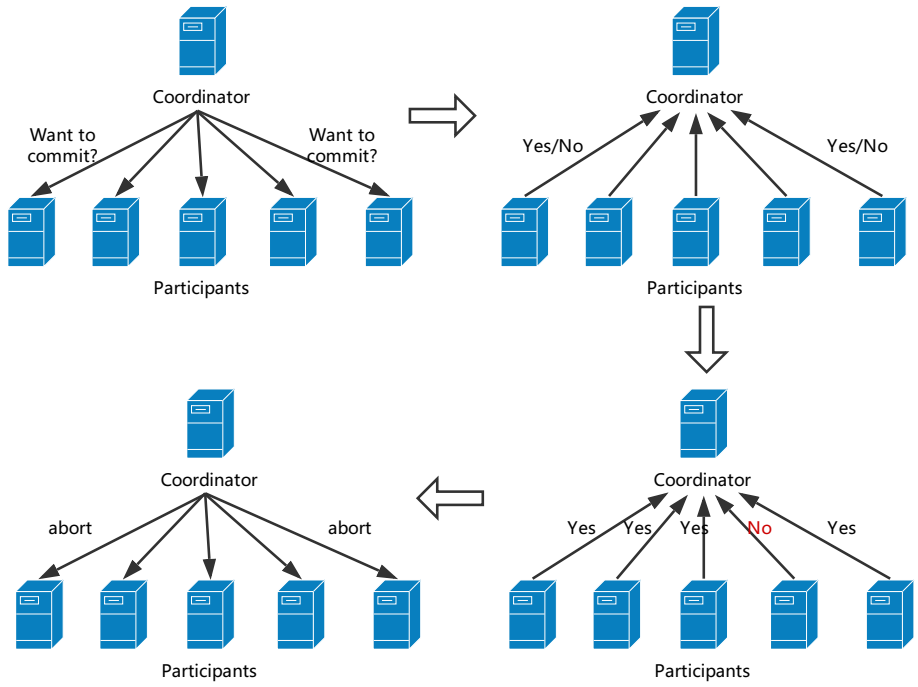

图 21: 2PL-abort

崩溃恢复

- 恢复时日志中可能有下述情况
  - 有 commit 或 abort 记录：那么分布式事务处理结果已经收到，进行相应的本地 commit 或 abort
  - 有 prepare，而没有 commit/abort：那么分布式事务的处理结果未知，需要和 prepare 记录中的 coordinator 进行联系
  - 没有 prepare/commit/abort：那么本地 abort