数据管理基础-2

索引

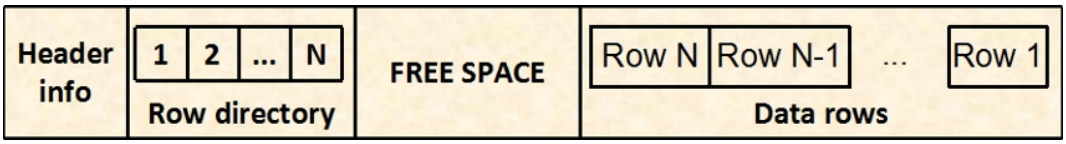
基本概念

- 索引
  - （key-value，row-pointer）
    - key-value 用于查找
    - row-pointer 足够定位磁盘上的行（一次 I / O）
- SQL 创建索引

```
CREATE [UNIQUE] INDEX index_name
  ON table_name (col_name [ASC|DESC]
    {,col_name [ASC|DESC]...});
```

- UNIQUE：key-value 和 row 是一对一的

磁盘存储

- Disk Access
  - 寻道时间
  - 旋转延迟
  - transfer time
- 内存缓冲
  - 优点
    - 节省磁盘 I / O 时间
    - 等待 I / O 时 CPU 可以做其他事
- ORACLE 中创建表空间
  - 表空间（Table Space）
    - 由 OS files cross 组成
  - 数据存储页与行指针
  - 

    - 一个行在大多数架构中一般是连续的字节序列，N 个行放在一个页面中（ORACLE 中叫做 块）
    - Header Info 命名了页面的类型（数据段，索引段）和页面号
    - 行在 Data Rows 中从右到左存储
    - Row Directory 中实体从左到右存储，给定了对应行的起始位置的偏移量
  - 磁盘指针
    - 表中的一个行可以被 页面号 和 slot number 唯一确定

B 树

- 树中的每个节点
  - 占据一整个磁盘页面
  - 有很多 fan-out
- ORACLE 中创建索引

```
CREATE [UNIQUE|BITMAP] INDEX [schema.]index_name
  ON table_name (col_name [ASC|DESC]
    {,col_name [ASC|DESC]...})
  [TABLESPACE tblespace] [STORAGE...] [PCTFREE n] [NOSORT]
```

- 二进制搜索算法
  - 见 PPT P26
- 百万索引实体
  - 实体数量：10^6
  - 实体大小：8 bytes
  - 磁盘页面大小：2k bytes
    - 每页实体数：250
    - 磁盘页面总数：4000
  - 磁盘 I / O 平均数量：$log_2 4000 + 1 = 13$
- B 树结构
  - 根节点
  - 内部节点（目录节点）
  - 叶节点
- 节点结构
  - P1 K1 P2 K2 ..... Pm Km P(m+1)
    - Ki 是 key-value 且 K1 < K2 < ... < Km
    - n 是key-value 的最大数量（秩）
  - 叶节点
    - Pi 是对应 Ki 的 row-id
    - P(m+1) 是指向下一个叶节点的指针
    - $floor(\frac{n+1}{2}) \leq m \leq n$
  - 根节点
    - $1 \leq m \leq n$
  - 内部节点
    - Pi 是指向子树 Ti 根节点的指针
    - 对子树 Ti 上的每个节点的 K 都有 $K_{i-1} \leq K < K_i$
- B 树中的百万索引实体
  - 实体数量：10^6
  - 实体大小：8 bytes
  - 磁盘页面大小：2k bytes
    - 每个节点的实体数：250
    - 总叶节点树：4000
    - 总内部节点数：16
    - 根节点：1
  - 磁盘 I / O 的平均数量：B 树深度 + 1 = 4
- B 树的 fan-out：B 树节点上的最大实体数

- B 树的深度：$log_f(N)$，其中 f 为 fan-out，N 为总行数
- B(+) 树的特性
  - 根节点以下的所有节点都至少有一半的实体信息
    - 经过多次删除后未必
  - 根节点至少包含两个实体（一个 key-value）
    - 只有一行被索引时例外（根节点是叶节点）
- B 数算法
  - 见数据结构：搜索，插入，删除
- ORACLE 的 Bitmap 索引
  - 为每一个不同的 key-value 使用一个 bitmap
    - 一个 bitmap 占据了一个 row-id 的列表 空间

Clustered 和 Non-Clustered 索引

- clustered 索引
  - 表中的行顺序与索引顺序相同（按 key-value 排列）

Hash Primary 索引

- 思想
  - 表中的行被通过 hash 函数被放置到伪随机数据页面插槽中，查找时亦同
  - 连续的 key-value 并不相邻，甚至可能在不同的页面上
- 在 hash cluster 中调整 HASHKEYS 和 SIZE
  - 总插槽数：S
  - 一个磁盘块上的插槽数：B
  - 冲突：两个不同的 key-value 被映射到同一个插槽

更新事务

事务

- 一个事务就是将一系列数据库操作打包在一起执行
- 事务能做什么？
  - 返回数据库中的数据
  - 更新数据库以反映真实世界事件的发生
- 事务的执行必须维持数据库状态和企业状态之间的关系

ACID

- A：原子性 Atomicity
  - 一个事务的更新操作要么全部发生要么无一发生
- C：一致性 Consistency
  - 事务不能够破坏规则，从一个一致状态转变为另一个一致状态
- I：隔离性 Isolation
  - 两个事务不能交错，即使是并发，也要像串行一样执行
- D：持久性 Durability
  - commit 结束后，要保证全部事务都被更新

数据库操作

- 读写操作
  - $R_i(A)$：id 为 i 的事务读取数据对象 A
  - $W_j(B)$：id 为 j 的事务写入 B
- 调度
  - 串行调度
    - 按照 T1...Tx 的顺序依次执行
- 调度器
  - 操作入，操作（history）出
  - 保证 history 中的操作和串行调度是等效的
    - 可能会延迟部分操作
    - 可能会丢弃部分事务
- 可串行化调度
  - 调度器任务：寻找到一定的规则使得事务可以交错并且保证串行性
  - 两个事务完全不使用相同的数据对象
    - 操作的顺序是不重要的
  - 使用相同的数据对象
    - $\dots R_1(A) \dots R_2(A) \dots$
      - 可 commute
        - 但如果 $R_1(A) \dots W_3(A) \dots R_2(A)$ 则 R1 R2 不可 commute
    - $\dots R_1(A) \dots W_2(A) \dots$
    - $\dots W_1(A) \dots R_2(A) \dots$
    - $\dots W_1(A) \dots W_2(A) \dots$
      - 以上三种都不可 commute
- 冲突动作
  - 两个 history 中的操作 $X_i(A)$ 和 $Y_j(B)$ 是冲突的当且仅当
    - A 等价于 B
    - $i \neq j$
    - X 和 Y 至少一者是 W
- 对于一个 history 的描述分为三个部分
  - 对操作目的的描述
  - 具体化数据对象
  - 一致性规则
- 前驱图
  - H 的前驱图是由 PG(H) 表示的有向图
  - 顶点对应已提交的事务
  - 每当 H 中的 X_i 和 Y_j 发生冲突，则 T_i 和 T_j 间存在一条边
- 定理10.3.4：串行定理
  - 一个 history H 有一个等价的串行执行序列 S(H) 当且仅当前驱图 PG(H) 中没有环

封锁

- 使用封锁技术的前提

    - 在一个事务访问数据库中的数据时，必须先获得被访问的数据对象上的封锁，以保证数据访问操作的正确性和一致性
- 封锁的作用
    - 在一段时间内禁止其它事务在被封锁的数据对象上执行某些类型的操作（由锁类型决定）
    - 同时也表明：持有该封锁的事务在被封锁的数据对象上将要执行什么类型的操作（由系统采用的封锁协议决定）
- 封锁类型
    - 排它锁（X 锁）：又称为 写封锁
    - 共享锁（S 锁）：又称为 读封锁
- 排它锁
    - 特性
        - 只有当数据对象 A 没有被其他事务封锁时，事务 T 才能对 A 施加 X 锁
    - 作用
        - T 自身可以对 A 读写，其他事物全部禁止访问 A
        - T 独占 A，保证了 T 对 A 访问操作的正确性和一致性
    - 缺点
        - 降低了并行性
    - X 锁必须维持到 T 结束
- 共享锁
    - 特性
        - 若 A 没有被其他事务封锁或仅加 S 锁，T 就能对 A 施加 S 锁
    - 作用
        - T 可以读 A，但不能写 A
        - 不同事务所申请的 S 锁可以共存于同一个 A 上，提高了并发性
        - 在 A 上的所有 S 锁被释放前，任何事务都不能写 A
    - S 锁不必维持到 T 结束
- 合适事务
    - 如果一个事务在访问 A 前按照要求申请对 A 的封锁，在操作结束后释放 A 上的封锁，这种事务被称为 合适事务
    - 是保持并发事务正确执行的基本条件
- 2PL
    - Two-Phase Locking
    - 完全看不懂 TODO
    - 定理：由 2PL 事务所构成的任意合法调度 S 都是冲突可串行化的

隔离级别

- 每一个事务都可以自主选择它自己与其他并发事务之间的隔离级别
- 一个事务所选择的隔离级别决定了它在运行过程中（调度器）所采用的封锁策略
- 四种隔离级别
    - READUNCOMMITED：未提交读
        - 当前事务不需要申请任何类型的封锁，所以可能会读到未提交的修改结果
        - 禁止事务以任何方式执行对数据的写操作，以避免写冲突
    - READCOMMITEED：提交读
        - 在读 A 之前需要申请对 A 的 S 锁，在读之后释放
        - 避免读取未提交的修改结果

    - READREPEATABLE：可重复读
        - 在读 A 之前需要申请 S 锁，并将封锁维持到当前事务结束
        - 可以避免其他并发事务对当前事务正在使用的数据对象的修改
    - SERIALIZABLE：可序列化
        - 并发事务以一种可串行化调度策略实现并发，避免相互干扰
- 不管采用何种隔离级别，在写 A 之前都要申请对 A 的 X 锁，持续到当前事务结束
- 封锁时间
    - 短期封锁：持续到操作完成
    - 长期封锁：持续到事务结束

日志

- 储存了每次更新事务的 Before Image 和 After Image
    - Before Image：旧
    - After Image：新
- 日志种类
    - UNDO
        - 只有 Before Image
        - 恢复未提交事务
    - REDO
        - 只有 After Image
        - 恢复已提交的事务
    - UNDO/REDO
        - 两者都有
        - 什么都能恢复
- UNDO
    - 格式
        - <Start T>
        - <Commit T>
        - <Abort T>
        - <T, X, V>
            - 修改了数据库元素 X 的值，X 的旧值时 V
    - 记载规则
        - 如果 T 修改 X，则 <T, X, V> 必须在 X 新值写到磁盘前写到磁盘
        - 如果 T 提交，<Commit T> 必须在 T 改变的所有 DB 元素写到磁盘后再写入磁盘
    - 其他操作
        - Flush Log：将内存中的日志记录全部写入磁盘
        - Output(A)：将 A 的值写入数据库的磁盘
        - D-A：数据库元素 A 在计算机磁盘中的值
        - M-A：A 在内存缓冲中的值
        - t：内存变量
    - 恢复过程
        - 将所有事务划分为两个集合
            - 已提交：有 Start 有 Commit
            - 未提交：有 Start 无 Commit
        - 从日志头部向后扫描整个日志

        - 如果 Commit 已被扫描到，继续扫描吓一跳
        - 否则，由恢复管理其将数据库中 X 的值改为 V
        - 在日志末尾为每个未结束事务写入一条 Abort T 并刷新（Flush Log）
    - 不足
        - 在事务改变的所有数据写入磁盘前不能提交该事务
        - 需要执行许多写操作，增加了事务提交的时间开销
            - 思考：在一个事务T被提交后，能否允许将事务T的修改结果暂时保存在内存中，在需要的时候再写入磁盘，以减少磁盘 I/O 的次数？
- REDO
    - 格式
        - <T, X, V> 中 V 记录更新后的值
        - 其余同 UNDO
    - 记载规则
        - 在修改磁盘上任何数据库元素 X 之前，要保证所有与 X 这一修改有关的 <T, X, V> 和 Commit T 都必须出现在磁盘上
            - 体现在 Output 前的 Flush Log
    - 恢复过程
        - 确定所有已提交的事务
        - 从日志首部开始扫描，对遇到的 T, X, V
            - 如果 T 未提交，继续扫描
            - 如果 T 已提交，则 X 写入新值 V
        - 对每个未完成的 T，日志尾部写入 Abort T 并刷新日志
    - 与 UNDO 主要区别
        - 恢复的目的不同
        - Commit T 写入日志的时间不同
            - UNDO 是在 T 的所有 Ouput 结束后写入
            - REDO 是在 Commit 被 Flush 后才能 Output
        - 在 T, X, V 中的 V 不同
    - 不足
        - 要求在日志记录刷新前将修改的数据保存在内存缓冲区，可能增加事务需要的平均缓冲区数量
- UNDO/REDO
    - 格式
        - <T, X, V, W>：V 是更新前的值
        -
    - 记载规则
        - 由于 T 而修改的 X 写入前，更新记录必须出现在磁盘上
            - 只要确信记录已经 Flush 就可以 Output，甚至可以不管 T 是否提交
        - 在每一条 Commit T 后必须紧跟一条 Flush Log
            - 为了确保在日志中写入 Commit T 的事务 T 确实被提交
    - 恢复
        - 根据 Commit T 是否出现在磁盘中来决定 T 是否被提交，然后
            - 从后往前撤销所有未提交事务
            - 从前往后重做所有已提交事务

检查点

- Commit-Consistent 检查点
  - 在检查点完成前，没有新的事务可以开始
  - 等到所有当前活跃的事务被提交或中止，并且在日志中写入了 Commit 或 Abort 记录
  - 将日志记录刷新到磁盘
  - 写一个特殊的日志记录 CKPT 到磁盘中，检查点完成
- Cache-Consistent 检查点
  - TODO
- Fuzzy 检查点
  - TODO
- UNDO 日志中的检查点
  - 在进行故障恢复时，只要逆向扫描到第一条 CKPT 就可以结束故障恢复工作
- 非静止检查点
  - 在设置检查点的过程中，允许新的事物进入系统
  - 步骤
    - 写入日志记录 <Start CKPT(T1…Tk)>，并刷新日志
    - 等待 T1…Tk 中每一个事务的提交或中止，但允许开始执行其他新的事务
    - 当 T1…Tk 全完成，写入 <End CKPT> 并刷新日志
  - 恢复
    - 从日志尾部向后扫描日志文件进行故障恢复
    - 如果先遇到 End CKPT，就继续向后扫描，直到出现对应的 Start CKPT 就可以结束故障恢复工作，这之后的日志记录是没有用的，可以被抛弃
    - 如果先遇到 Start CKPT，需要撤销两类事务的操作
      - 在 Start CKPT 之后启动的事务
        - 在扫描到 Start CKPT 时，这类事务的操作已被撤销
      - T1…Tk 中在系统崩溃前尚未完成的事务
        - 继续向后扫描，直至其中未完成事务的访问操作全部撤销
- REDO 日志中的检查点
  - 插入非静止检查点步骤
    - 写入 Start CKPT(T1…Tk) 并刷新日志
      - 同时获得当时所有已提交事务的标识符集合 S
    - 将集合 S 中的事务已经写到内存缓冲区但还没有写到磁盘的数据写入磁盘
    - 写入 End CKPT 并刷新日志
  - 恢复
    - 寻找最后一个被记入日志的检查点记录
      - 如果是 End CKPT（记为 Et），假设与之对应的检查点记录是 Start CKPT(T1..Tk)（记为 St），并找到最早出现的 Start Ti（记为 ti），则针对事务 T1…Tk 以及 St 之后开始的事务，重做其中已经被提交的事务
      - 如果是 Start CKPT（记为 St1），继续寻找前一个 End CKPT（记为 Et2），以及与 Et2 对应的 Start CKPT（记为 St2）
        - 该情况就如同 Et2 是日志文件中最后一条检查点一样进行恢复
- UNDO/REDO 日志中的检查点
  - 插入非静止检查点步骤

- 写入 Start CKPT(T1…Tk) 并刷新日志
- 将所有被修改过的缓冲区写到数据库磁盘中
- 吸入 End CKPT 并刷新日志