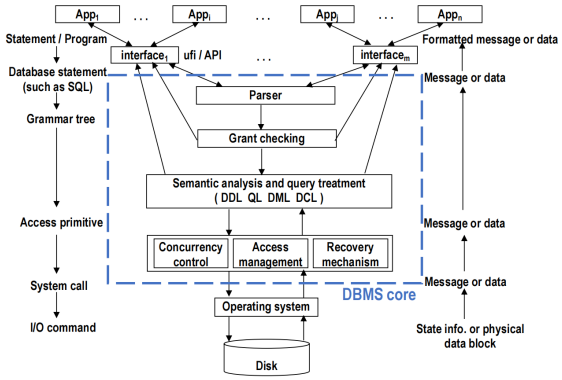
Chapter 4 Database Management Systems

1. The Architecture of DBMS

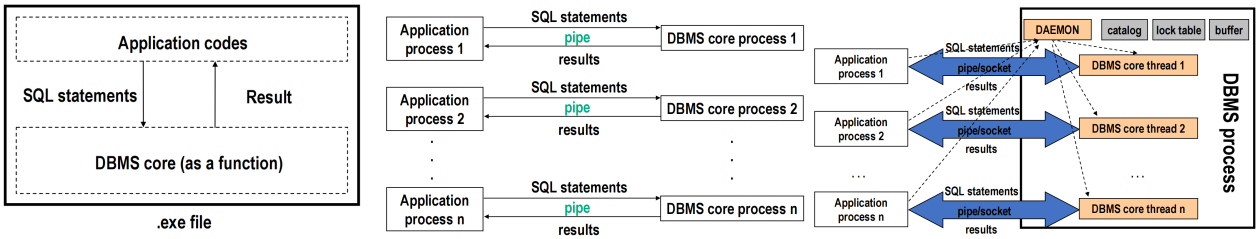
1.1. The components of DBMS core



1.2. The process structure of DBMS

1.2.1. 单进程结构 Single process structure

- 应用程序与DBMS核心编译为单个.exe文件，作为一个单一进程运行。



1.2.2. 多进程结构 Multi processes structure

- 一个应用程序进程对应一个DBMS核心进程

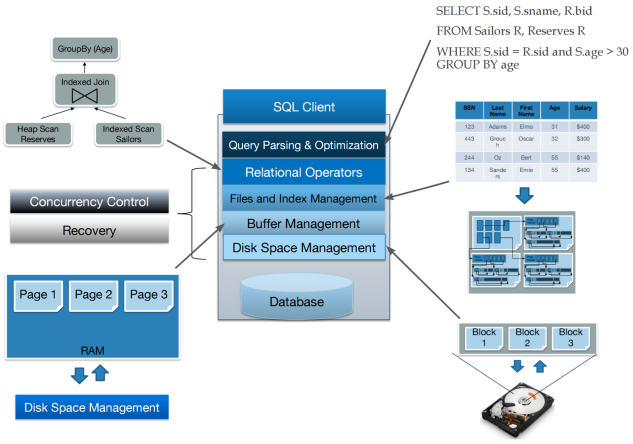
1.2.3. 多线程结构 Multi threads structure

- 只有一个DBMS进程，每个应用程序进程都对应一个DBMS核心线程。

1.2.4. 进程/线程之间的通信协议

1.3. Architecture of a DBMS

- 分层组织
- 每层抽象下层
  - 管理复杂性
  - 性能假设
- 良好系统设计的示例



1.3.1. SQL Client

SQL客户端是与DBMS进行交互的用户界面，允许用户通过SQL发送请求和查询，以获取数据或执行操作。

1.3.2. Parsing & Optimization

- 解析、检查和验证SQL语句的语法和语义正确性，并将其翻译成高效的关系查询计划
- 涉及语法分析、语义分析和查询优化等技术，以确保查询能够以最优的方式执行

1.3.3. Relational Operators

- 通过操作 记录和文件 执行数据流。
- 关系运算符是数据库管理系统中用于执行数据流操作的操作符。

1.3.4. Files and Index Management

- 将表格和记录作为一组逻辑文件中的页面进行组织

- 文件和索引管理是数据库管理系统中用于组织和管理表格和记录的方式，涉及将表格和记录组织为一组逻辑文件中的页面，以便有效地存储和访问数据
- 此过程还包括索引的创建、维护和使用，以提高数据检索的性能

1.3.5. Buffer Management

- 提供“在内存中运行”的错觉
- 将磁盘上的数据块缓存到内存中，并根据需要进行读写，减少磁盘I/O操作，提高数据的访问速度和性能

1.3.6. Disk Space Management

磁盘空间管理是数据库管理系统最低层，管理磁盘上的空间，将页面请求转化为单个或多个设备上的物理字节的过程；负责管理磁盘空间的分配和回收，以确保数据库可以有效地使用可用的磁盘存储空间。

- 目的：
  - 将页面映射到磁盘上的位置
  - 将页面从磁盘加载到内存中
  - 将页面保存回磁盘并确保写入操作
- 上层调用此层：
  - 读取/写入页面
  - 分配/释放逻辑页面
- 请求页面：
  - 请求一系列页面时，最好通过顺序存储在磁盘上的页面来满足
    - 物理细节对系统的较高层级是隐藏的
    - 较高层级可以“安全”地假设下一页的访问是快速的，因此它们会期望顺序页面的扫描速度很快。
- 实现：
  - 直接与存储设备通信
  - 在文件系统上运行
    - 它会在一个干净的磁盘上分配一个单个的大型“连续”文件，并假设顺序/接近的字节访问速度很快
    - 大多数文件系统会优化磁盘布局以实现顺序访问
      - 如果我们从一个空的磁盘开始，这种方法可以给我们带来大致符合我们期望的结果

- DBMS的“文件”可能会跨多个文件系统文件、多个磁盘或多个计算机进行存储

1.3.7. Concurrency & Recovery

- 与存储器和内存管理有关的两个贯穿始终的问题
- 并发涉及管理多个并发事务之间的访问冲突和一致性问题
- 恢复涉及到在系统故障或错误发生时，将数据库恢复到一致的状态，以确保数据的完整性和可靠性

2. Database Access Management

最终将数据库访问转换为文件操作，文件结构和提供的访问路径将直接影响数据访问速度，一种文件结构不可能对所有种类的数据访问都有效

2.1. 访问类型 Access types

- 查询文件的所有或大部分记录（>15%）
- 查询一些特殊记录
- 查询部分记录（<15%）
- 范围查询 Scope query
- 记录的更新

2.2. 文件组织 File organization

- 堆文件（heap file）：根据插入顺序存储记录，并顺序检索，这是最基本和最通用的文件组织形式
- 直接文件（direct file）：根据某些属性的值通过哈希函数映射记录地址
- 索引文件：索引+堆文件/簇
- 网格结构文件：适合多属性查询
- 裸设备磁盘（注意文件逻辑块和物理块的差异。可以使用裸设备磁盘来控制操作系统中的物理块）

2.2.1. 许多数据库文件结构

- 无序堆文件 Unordered Heap Files：页上任意放置记录
- 簇堆文件 Cluster Heap File：记录和页是成组出现的
- 排序文件：页和记录是按排序顺序排列的
- 索引文件：B+树、线性哈希、…，可能包含记录或指向其他文件的记录。

2.2.2. 记录格式 Record Formats

- 关系模型 -> 表格中的每个记录具有一些固定的类型
- 假设系统目录存储模式
  - 不需要与记录一起存储类型信息（节省空间）
  - 目录只是另一个表格……
- 目标：记录在内存和磁盘格式上应该是紧凑的，快速访问字段

2.2.3. 定长字段 Fixed-Length Records

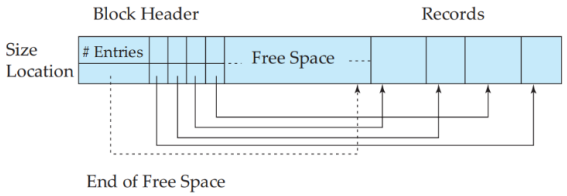
- 简单方法：
  - 从字节 $n*(i-1)$ 开始存储记录 i ，其中 n 是每个记录的大小。
- 记录访问很简单，但记录可能会跨越块 -> 修改：不允许记录跨越块边界。
- 删除记录 i ：
  - 将记录 $i+1$，..，n 移动到 i，...，$n-1$ 的位置
  - 将记录 n 移动到位置 i
  - 不移动记录，将自由列表上所有自由记录链接起来（最后一条“接地”）

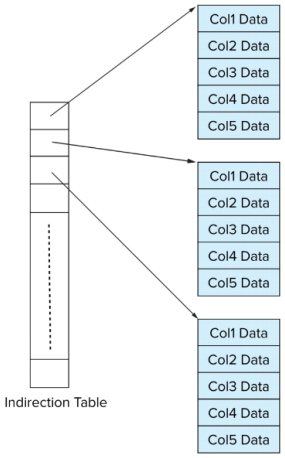
2.2.4. 变长字段 Variable-Length Records

- 在数据库系统中，可变长记录的出现方式有多种
  - 文件中的多种记录类型的存储
  - 允许一个或多个字段具有可变长度的记录类型，例如字符串（varchar）
  - 允许重复字段的记录类型（在一些旧的数据模型中使用）
- 属性按顺序存储
- 用固定大小（偏移量、长度）表示的可变长度属性，实际数据存储在所有固定长度属性之后
- 用空值位图表示的空值
- **插槽页结构 Slotted Page Structure**
  - 插槽页头部（Slotted Page Header）
    - 记录条目的数量
    - 块中可用空间的结束位置
    - 每个记录的位置和大小
  - 记为了保持记录之间的连续性并消除空隙，记录可以在页面内进行移动。但是，当记录移动时，页头中的相应条目必须更新

- 为了处理记录的移动，指针应该指向页头中记录条目的位置，而不是直接指向记录本身。这样，当记录移动时，只需要更新页头中相应条目的位置即可

- **面向列存储 Column-Oriented Storage**
  - 也被称为列式表示
  - 存储关系中的每个属性单独存储
  - 可以无需缓冲管理器直接在内存中存储记录
  - 可以使用面向列的存储在内存中进行决策支持应用
    - 压缩可以减少内存需求





2.3. 索引技术 Index technique

索引是一种数据结构，它能够通过搜索关键字快速查找（lookup）和修改数据条目

- 查找：支持多种不同的操作：相等性、一维范围、二维区域等
- 搜索关键字：关系中的任何子集的列。不需要唯一性：例如（firstname）或（firstname, lastname）

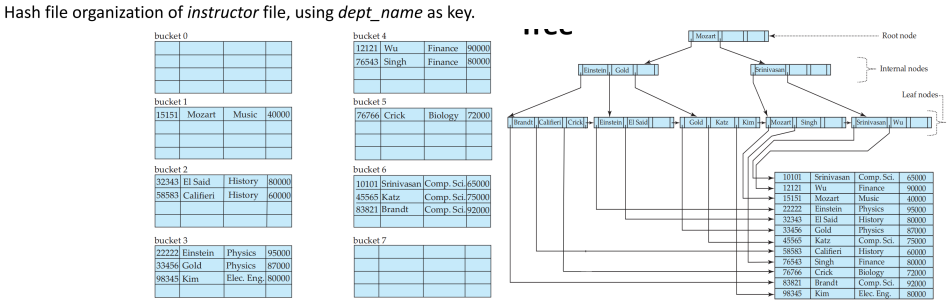
2.3.1. 基本概念

- 索引机制用于加快对所需数据的访问速度
- 搜索关键字（Search key）：用于在文件中查找记录的一组属性的属性
- 索引文件由记录组成（称为索引条目），其形式：[search-key] [pointer]
- 索引文件通常比原始文件小得多
- 两种基本的索引类型：
  - 有序索引 Ordered indices：搜索关键字按排序顺序存储
  - 哈希索引 Hash indices：使用哈希函数将搜索关键字均匀分布在“桶”中

2.3.2. B+ 树

- B+树是一种满足以下属性的根节点树：
  - 从根节点到叶节点的所有路径长度都相同
  - 每个节点如果不是根节点或叶节点，则具有 $\lceil n/2 \rceil \sim n$ 个孩子节点。
  - 叶节点具有 $\lceil (n-1)/2 \rceil \sim (n-1)$ 个值。
  - 特殊情况：
    - 如果根节点不是叶节点，则至少有两个孩子节点。
    - 如果根节点是叶节点（即树中没有其他节点），则它可以具有0到n–1个值。
- 节点结构
  - $P_1 \mid K_1 \mid P_2 \mid ..... \mid P_{n-1} \mid K_{n-1} \mid P_n$
    - K_i 是搜索关键字
    - P_i 是指向子节点（非叶节点）或记录指针或记录桶（叶节点）的指针
  - 节点的搜索关键字按顺序排列，如 $K_1 < K_2 < K_3 < ... < K_{n-1}$ （假设初始时没有重复的关键字，稍后处理重复关键字）
- 提供三种存储路径，且可以动态的随着数据变化而消长且始终保持平衡
  - 通过索引集进行树形搜索
  - 通过顺序集进行顺序搜索
  - 先通过索引集找到入口在沿顺序集顺序搜索

2.3.3. Hashing

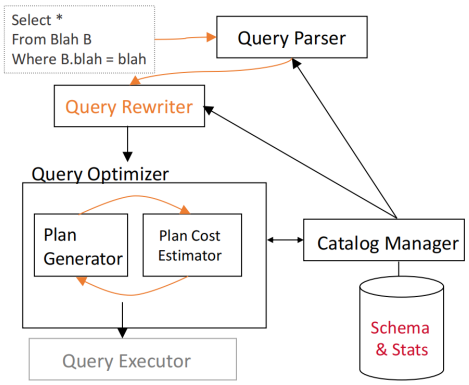


2.3.4. 动态哈希

- 定期重哈希 Periodic rehashing
  - 如果哈希表中的条目数量增加到哈希表大小的1.5倍
    - 创建一个新的哈希表，大小为前一个哈希表的两倍
    - 将所有条目重哈希到新表中

- 线性哈希 Linear Hashing
  - 以增量方式进行重哈希
- 可扩展哈希 Extendable Hashing
  - 对基于磁盘的哈希，具有多个哈希值共享的桶
  - 哈希表中的条目数量成倍增加，而桶的数量不加倍

3. Query Optimization



- 查询解析器 Query Parser：检查正确性、权限；生成解析树；straight forward
- 查询重写器 Query Rewriter：将查询转换为标准形式（平滑视图、子查询拆分为更少的查询块）、许多开源DBMS的弱点
- 基于成本的查询优化器：一次优化一个查询块（选择、投影、连接、分组/聚合、排序）；使用目录统计信息找到每个查询块的最小“成本”计划。

3.1. Overview

3.1.1. 基本逻辑

- 查询块可以转换为关系代数
- 关系代数转换为树形结构
- 每个操作符都有实现选择
- 操作符还可以以不同的顺序应用

3.2. 代数优化

3.2.1. 关系代数等价性

- Selections
  - $\sigma_{c1\wedge\ldots\wedge cn}(R) \equiv \sigma_{c1}(\ldots(\sigma_{cn}(R))\ldots)$ (级联)
  - $\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$ (交换)
- Projections
  - $\pi_{a1}(R) \equiv \pi_{a1}(\ldots(\pi_{a1,\ldots,an-1}(R))\ldots)$ (级联)
- Cartesian Product
  - $R \times (S \times T) \equiv (R \times S) \times T$ (结合)
  - $R \times S \equiv S \times R$ (交换)
- Join 的结合与交换

  - Consider R(a,z), S(a,b), T(b,y)
    - $(S \bowtie_{b=b} T) \bowtie_{a=a} R \not\equiv S \bowtie_{b=b} (T \bowtie_{a=a} R)$ (*not* legal!!)
    - $(S \bowtie_{b=b} T) \bowtie_{a=a} R \not\equiv S \bowtie_{b=b} (T \times R)$ (*not* the same!!)
    - $(S \bowtie_{b=b} T) \bowtie_{a=a} R \equiv S \bowtie_{b=b \wedge a=a} (T \times R)$ (the same!!)

3.2.2. 常见的启发式搜索 Some Common Heuristics

- Selections：一旦你有了相关的列，就立即应用选择

$$\pi_{sname}(\sigma_{(bid=100 \wedge rating > 5)}(Reserves \bowtie_{sid=sid} Sailors))$$
$$\pi_{sname}(\sigma_{bid=100}(Reserves) \bowtie_{sid=sid} \sigma_{rating > 5}(Sailors))$$

- Projections：只保留评估下游操作符的列

$$\pi_{sname}\sigma_{(bid=100 \wedge rating > 5)}(Reserves \bowtie_{sid=sid} Sailors)$$
$$\pi_{sname}(\pi_{sid}(\sigma_{bid=100}(Reserves)) \bowtie_{sid=sid} \pi_{sname,sid}(\sigma_{rating > 5}(Sailors)))$$

- 避免笛卡尔积
  - 如果有选择，使用 theta-join 而不是交叉积
  - 考虑 R(a,b)，S(b,c)，T(c,d)：优先使用 (R ⋈ S) ⋈ T 而不是 (R x T) ⋈ S

3.3. 依赖存取路径的优化

3.3.1. 选择操作的启发式选用存取路径

- 对于小关系，不必考虑其他存取路径，直接用顺序扫描
- 如果无索引或散列等存取路径可用，或估计中选的元组数在关系中占有较大的比例(如大于15%），且有关属性上无族集索引，则用顺序扫描
- 对于主键的等值条件查询，最多只有一个元组可以满足此条件，应优先采用主键上的索引或散列

- 对于非主键的等值条件查询，需要估计中选的元组数在关系中所占的比例：如果比例较小(如小于15%），可用无序素引，否则只能用簇集索引或顺序扫描
- 对于范围条件，一般先通过索引找到范围的边界，再通过索引的顺序集沿相应方向搜索。例如，对于条什 AGE⩾20，可先找到 AGE=20 的顺序集结点，再沿顺序集向右搜索。若中选的元组数在关系中所占比例较大，且无有关属性的簇集索引，则宜采用顺序扫描
- 对于用 AND 连接的合取选择条件，若有相应的多属性索引，则优先采用多属性索引。否则，可检查诸条件中有无多个可用二次索引检索的，若有，则用预查找法处理，即通过二次索引找出满足各条件的 tid 集合，再求这些tid 集合的交集；然后取出交集中 tid 所对应的元组，且在取这些元组的同时，用合取条件中的其余条件检查。凡能满足所有其余条件的元组，即为所选择的元组。如果上述途径都不可行，但合取条件中有个别条件具有规则(3，4，5)中所述的存取路径，则可用此存取路径选择满足此条件的元组，再将这些元组用合取条件中的其他条件筛选。若在诸合取条件中，没有一个具有合适的存取路径，那只有用顺序扫描
- 对于用OR连接的析取选择条件，尚无好的优化方法，只能按其中各个条件分别选出一个元组集，再求这些元组集的并。并是开销大的操作，而且在OR连接的诸条件中，只要有一个条件无合适的存取路径，就不得不采用顺序扫描来处理这种查询。因此，在查询语向中，应尽量避免采用析取选择条件
- 有些选择操作只要访问素引就可得到结果，如查询索引属性的最大值、最小值、平均值等。在此情况下，应优先利用索引，避免访问数据

3.3.2. 连接操作的实现和优化

3.3.2.1. 嵌套循环法

- 对于 $R \bowtie S$ ，对于每个 $R(i)$ 与所有 $S(j)$ 比较， R 为外循环， S 为内循环
- 优化：每次取一个块的 $R(i)$

3.3.2.2. 利用索引或哈希匹配

- 在内关系循环时，如果有合适的存取路径（如连接属性上的散列或索引），可以取代顺序扫描以减少IO
  - 有簇集索引或哈希散列时最好
  - 无序索引也还行
- 但如果每次匹配元组数占比较大时，无序索引还不如顺序扫描

3.3.2.3. 排序归并法

- 如果 R, S 按连接属性排序，则可按序比较 $R.A$ 和 $S.B$ 找出所有匹配的元组。在此方法中，R 和 S 都只需扫描一次

3.3.2.4. 散列连接法

- 由于连接属性 $R.A$ 和 $S.B$ 应具有相同的域，因此可以用 A, B 作为 R, S 的散列键，用相同的散列函数，把 R, S 散列到同一散列文件中
- 符合连接条件的 R 和 S 的元组必然位于同一桶中，但是同一桶中的 R, S 元组未必都满足连接条件
- 由于桶中的元组一般不会很多，在匹配时可用嵌套循环法

3.3.2.5. 启发式规则

- 如果两个关系都已按连接属性排序，则优先用排序归并法
    - 如果两个关系中已有一个关系按连接属性排序，另一个关系很小，也可考虑对此关系按连接属性排序,再用排序归并法连接。
- 如果两个关系中有一个关系在连接属性上有索引(特别是簇集索引)或散列，则可令另关系为外关系，顺序扫描，并且利用内关系上的索引或散列寻找其匹配元组，以代替多遍扫描
- 如果应用上述两规则的条件都不具备，两个关系都比较小，可以应用嵌套循环法
- 如果（1，2，3）均不适合，可以用散列连接法

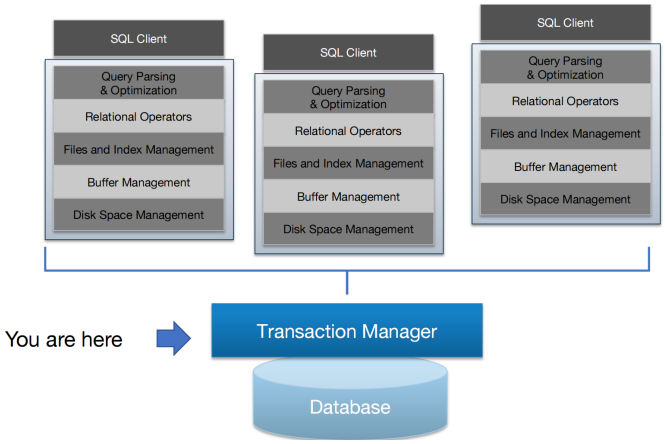
3.3.3. 投影操作

- 投影操作一般与选择、连接等操作同时进行,不需要附加的IO开销
- 如果在投影的属性集合中没有主键,则投影结果中可能出现重复元组
    - 一般须将投影结果按其全部属性排序，使重复元组连续存放，以便于发现和消除
    - 散列法消除：将投影结果按某一属性或多个属性散列成一个文件，当一个元组被散列到一个桶中时，可以检查是否与桶中已有元组重复

3.3.4. 组合操作

- Materializing Inner Loops：可以设置临时文件，节省各种开销

4. Transaction Management



4.1. Transactions 事务

4.1.1. Concept and Implementation

- 一系列需要作为一个原子单元执行的多个操作
- 应用视图（SQL视图）
  - 开始事务
  - 一系列SQL语句
  - 结束事务

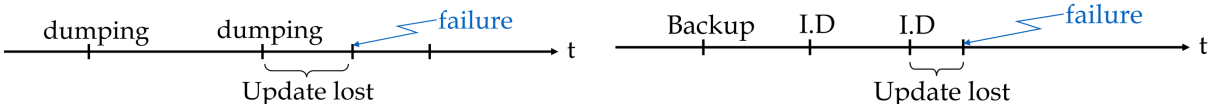
4.1.2. ACID: 事务的高级属性

- 原子性（Atomicity）：事务中的所有操作要么全部执行，要么全部不执行
- 一致性（Consistency）：事务对数据库的作用使数据库从一个一致状态转变到另一个一致状态
- 隔离性（Isolation）：每个事务的执行与其他事务隔离。
- 持久性（Durability）：如果事务提交，其效果将持久存在。

4.2. Recovery

- 恢复机制在DBMS中的主要作用有：
  - 减少故障发生的可能性（预防）
  - 故障发生后的恢复（解决）
- 在发生故障后将数据库恢复到一致的状态。冗余是必要的；应该检查所有可能发生的故障。

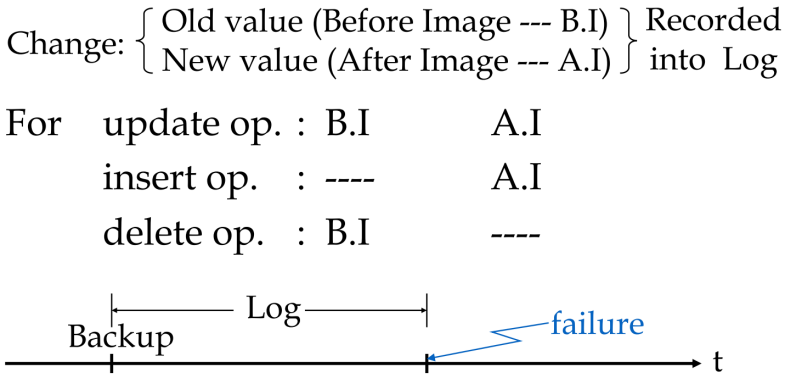
4.2.1. 定期备份 Periodical dumping



- 变种：备份+增量转储（Backup + Incremental dumping）
  - 在这种方法中，除了定期进行完整的备份外，还会进行增量备份，只备份数据库中发生更新的部分（I.D）
- 这种方法（包括变种）实现起来比较容易，开销也较低，但是一旦发生故障，部分更新可能会丢失。因此，它通常用在文件系统或小型DBMS中。

4.2.2. 备份+日志 Backup + Log

日志：记录自上次备份拷贝以来数据库的所有更改。

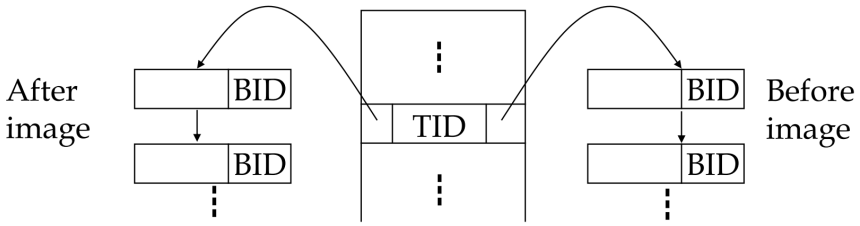


- 如果某些事务只完成了一部分操作（即半完成状态），可以使用事务日志中的回滚信息（B.I）来撤销这些操作，将数据库恢复到事务开始之前的状态。
- 如果某些事务已经完成了所有操作，但是尚未及时将结果写入数据库，可以使用事务日志中的重做信息（A.I）来重新执行这些操作，将结果写入数据库中

4.2.3. 支持恢复的一些结构

- 恢复信息（如日志）应该存储在非易失性存储器中。存储以下信息：
  - 提交列表（Commit list）：记录已经提交的事务的事务标识符（TID）的列表。
  - 活动列表（Active list）：记录正在进行中的事务的事务标识符（TID）的列表。
  - 前像文件（Before image）：
    - 可以看成一个堆文件
    - 每个物理块有个块标识符BID（block identifier），BID由TID、关系名和逻辑块号组成。逻辑块号在关系中是唯一的。
    - 如果一个事务需要回卷，可以在前像文件中找出该事务的所有前像块，按照逻辑块号写入到关系的对应块，从而消除该事务对数据库的影响。

- 后像文件（After image）：
  - 结构与前像文件类似
  - 在恢复时，可按提交事务表中的事务次序，按逻辑块号，写入其后像。这相当于按提交的次序重做各个事务。



4.2.4. 提交规则和提前记录规则

- 提交规则（Commit Rule）：
  - 在事务提交之前，A.I（后像文件）必须被写入非易失性存储介质。这意味着，在将事务标记为已提交之前，必须确保已将事务的操作持久化，以保证事务的持久性和一致性。
- 提前记录规则（Log Ahead Rule）：
  - 如果在事务提交之前，A.I已经被写入数据库，则B.I（前像文件）必须首先被写入日志。这是为了确保在事务提交之前，对已经应用到数据库的操作进行了记录，以便在恢复时能够正确地执行撤销操作。
- 恢复策略：
  - 撤销（undo）和重做（redo）的特性是幂等的：
    - undo(undo(undo ··· undo(x) ···)) = undo(x)
    - redo(redo(redo ··· redo(x) ···)) = redo(x)

4.2.5. 更新策略 Three kinds of update strategy

- **commit 前将A.I写入数据库**
  - 将该事务的事务标识符（TID）添加到活动列表（active list）中，表示该事务正在进行中。
  - 根据LAR，B.I 必须首先被写入日志，以确保对已经应用到数据库的操作进行记录。
  - 接下来，将 A.I 同时写入数据库和日志：确保事务的持久性和一致性，同时保留了日志中的操作记录。
  - 当事务完成并准备提交时，将其添加到提交列表（commit list）中，表示该事务已经成功提交，
  - 将该事务的TID从活动列表中删除

TID →active list
B.I →Log (Log Ahead Rule)
A.I →DB, Log
⋮
TID →commit list
delete TID from active list

Commit list	Active list	
	✓	Undo, delete TID from active list
✓	✓	delete TID from active list
✓		nothing to do

- **commit 后将A.I写入数据库**
  - 将该事务的TID添加到活动列表中，表示该事务正在进行中。
  - 根据提交规则，A.I必须先被写入日志，以确保在提交之前将操作持久化。
  - 将该事物的TID添加到提交列表（commit list）中，表示该事务已经成功提交。
  - 将该事务的A.I写入数据库：这样可以保证事务操作的持久性和一致性。
  - 将该事务的TID从活动列表中删除

TID →active list
A.I →Log (Commit Rule)
⋮
TID →commit list
A.I →DB
delete TID from active list

Commit list	Active list	
	✓	delete TID from active list
✓	✓	redo, delete TID from active list
✓		nothing to do

- **commit 与将 A.I 写入DB同时进行**
  - 将该事务的TID添加到活动列表中，表示该事务正在进行中。
  - 根据提交规则，A.I必须先被写入日志，以确保在提交之前将操作持久化。同时，根据LAR，B.I也需要被写入日志，以确保对已经应用到数据库的操作进行记录。
  - 将该事务的A.I部分写入数据库，这表示部分操作已经完成。
  - 将该事务的TID添加到提交列表
  - 将该事务的剩余A.I写入数据库
  - 将该事务的TID从活动列表中删除

TID →active list
A.I, B.I →Log (Two Rules)
A.I →DB (partially done)
⋮
TID →commit list
A.I →DB (completed)
delete TID from active list

Commit list	Active list	
	✓	Undo, delete TID from active list
✓	✓	redo, delete TID from active list
✓		nothing to do

4.3. Concurrency Control： 提供隔离性

4.3.1. 事务调度

- 一个调度计划是对一个或多个事务的数据上的一系列操作的顺序。
- 操作：开始Begin、读取Read、写入Write、提交Commit和终止Abort。
  - e.g. R1(A) W1(A) R1(B) W1(B) R2(A) W2(A) R2(B) W2(B)
- 根据约定，我们只包括已提交的事务，并省略开始和提交操作。

4.3.2. 串行等价性 Serial Equivalence

- 串行调度（Serial schedule）：每个事务从开始到结束运行，中间没有其他事物的操作
- 两个调度是等价的：
  - 涉及相同的事务
  - 每个单独事务内的操作顺序相同
  - 两个调度结束后数据库状态相同

4.3.3. 可串行化 Serializability

- 定义：调度 S 等同于某个串行调度（Serial schedule），则 S 是可串行化的

T1: Transfer $100 from A to B	T2: Add 10% interest to A & B
begin	
read(A)	
A = A - 100	
write(A)	
read(B)	
B = B + 100	
write(B)	
commit	
	begin
	read(A)
	A = A * 1.1
	write(A)
	read(B)
	B = B * 1.1
	write(B)
	commit

串行调度

T1: Transfer $100 from A to B	T2: Add 10% interest to A & B
begin	
read(A)	
A = A - 100	
write(A)	
	begin
	read(A)
	A = A * 1.1
	write(A)
read(B)	
B = B + 100	
write(B)	
commit	
	read(B)
	B = B * 1.1
	write(B)
	commit

可串行化调度

4.3.4. 冲突操作（难以检查属性“ 两个调度结束后数据库状态相同”）

- 两个操作是操作冲突：
  - 由不同的事务执行的
  - 在同一个对象上执行
  - 其中至少一个是写入操作

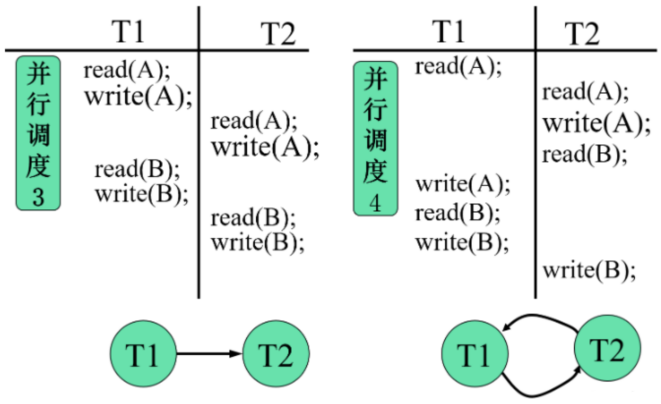
- 非冲突操作的顺序对数据库的最终状态没有影响

4.3.5. 冲突可串行化调度 Conflict Serializable Schedules

- 冲突等价（conflict equivalent）：两个调度
  - 它们涉及相同事务的相同操作
  - 每对冲突操作以相同的方式排序
- 冲突可串行化（conflict serializable）： S 与某个串行调度是冲突等价的
  - 同时可以得到 S 也是可串行化的
- 注意：一些可串行化调度不是冲突可串行化的
  - 例子：串行化调度 $S = R_1(x)W_1(x)W_2(x)W_3(x)$
    - $S_1 = R_1(x)W_2(x)W_1(x)W_3(x)$ 显然是可串行化的
    - 但由于 $W_1(x)$与$W_2(x)$ 是冲突操作且交换了顺序，故不是冲突可串行化的
  - 检测冲突可串行化（结果为False），即认为该调度是不可串行化的，产生FN（假阴性）
  - 但可以额节省成本

4.3.6. 冲突依赖图 Conflict Dependency Graph

- 用于判断是否冲突可串行化
  - 当且仅当调度表的依赖图是无环图时，该调度是冲突可串行化的。
- 每个事务有一个节点
  - 边从 T_i 到 T_j 含义
    - T_i 的操作 O_i 与 T_j 的操作 O_j 发生冲突
    - O_i 在调度中出现的比 O_j 早。



4.4. Locking Protocol 锁协议

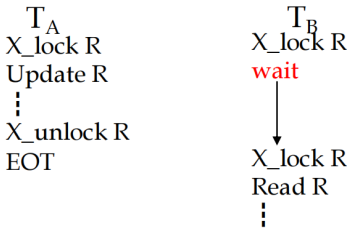
锁定方法是并发控制中最基本的方法，可能存在多种锁定协议。

4.4.1. X locks 排他锁

- 只使用一种类型的锁，既用于读取也用于写入。

Compatibility matrix : NL－no lock　　X－X lock
Y －compatible　N－incompatible

	NL	X
NL	Y	Y
X	Y	N



- 问题：可能导致连续回卷（T1挂了T2要回卷）
  - 一般要保持到EOT后（图中存在的问题）

4.4.2. (S,X) locks

- 共享锁（S）：读访问锁——允许多个事务同时访问共享资源，但不允许任何事务进行修改
- 排他锁（X）：写访问锁——独占锁，只允许一个事务持有，且其他事务不能同时持有共享锁或排他锁

(2) (S,X) locks

S lock --- if read access is intended.

X lock --- if update access is intended.

	NL	S	X
NL	Y	Y	Y
S	Y	Y	N
X	Y	N	N

- (S,X)锁的组合使用场景如下：
  - 某个对象加了S锁，其他事务可以同时持有S锁，以实现并发读取操作——提高并发度
  - 某个对象加了S锁，其他事务申请X锁需等待——保持一致性
  - 某个对象加了X锁，其他事务无法同时持有共享锁或排他锁

4.4.3. (S,U,X) locks

- U锁 — 更新锁。
- 更新：首先获取U锁，此时其他读可以同时拥有S锁；读过程完成后将其升级为X锁。
- 目的：缩短排它时间，以提高并发度，减少死锁。

	NL	S	U	X
NL	Y	Y	Y	Y
S	Y	Y	Y	N
U	Y	Y	N	N
X	Y	N	N	N

4.4.4. 锁管理 Lock Management

- 由锁管理器处理的锁定和解锁请求
- LM维护一个哈希表，以被锁定的对象的名称作为键，为当前持有的每个锁保留一个条目
- 条目包含
  - 授予集 Granted set：当前授予访问该锁的行为所属事务集合
  - 锁模式 Mode：持有的锁的类型（共享或独占）
  - 等待队列：锁请求队列。

	Granted Set	Mode	Wait Queue
A	{T1, T2}	S	T3(X) ← T4(X)
B	{T6}	X	T5(X) ← T7(S)

	Granted Set	Mode	Wait Queue
A	{T1, T2}	S	T2(X) ← T3(X) ← T4(X)
B	{T6}	X	T5(X) ← T7(S)

- 当锁请求到达时：
  - 是否 Granted Set 或等待队列中的任何事务需要冲突的锁?
    - 如果不需要，则将请求者放入授予集并让其继续进行
    - 如果需要，则将请求者放入等待队列（通常为FIFO）
- 锁升级：
  - 具有共享锁的事务可以请求升级为排他锁

4.4.5. DeadLock & LiveLock

- 死锁：等待循环，没有事务能够获得完成所需的所有资源。
- 活锁：虽然其他事务在有限的时间内释放了资源，但有些事务过长时间无法获得所需资源。
- 死锁检测——超时
  - 超时：如果一个事务等待了某些指定的时间，那么就假设发生了死锁，并应终止该事务。
  - 通过等待图 $G =< V, E >$ 检测死锁
    - V ：数据库中事务的集合 $\{T_i | T_i$是数据库中的事务$(i = 1, 2, ..., n)\}$ 。
    - E ： $\{< T_i, T_j > | T_i$等待$T_j (i \neq j)\}$ 。

        - 如果图中存在循环，则发生了死锁。
        - 检测时机：1）每一个新的等待事件发生；2）定期检测
    - 处理：
        - 循环等待中选择一个事务牺牲（最迟交付、占有锁最少、卷回代价最小...）
        - 回卷牺牲的事务，释放锁和资源
        - 将锁让给等待的事务
        - 重启牺牲的事物 （自动或手动）
- 避免死锁 DeadLock Avoidance
    - 在事务初始阶段请求所有锁。
    - 按资源指定顺序请求锁。
    - 一旦发生冲突就中止事务。
    - 事务重执：每个事务都被唯一地加上时间戳。如果 T_A 需要锁定已经被 T_B 锁定的数据对象，则使用以下方法之一
        - 等待–死亡 Wait–die：如果 T_A 比 T_B 老，则 T_A 等待，否则重执一段时间（即被中止并自动以原始时间戳重新尝试）。
        - 击伤–等待 Wound–wait：如果 T_A 比 T_B 年轻，则 T_A 等待，否则“伤害” T_B ，即 T_B 被中止并自动以原始时间戳重新尝试。

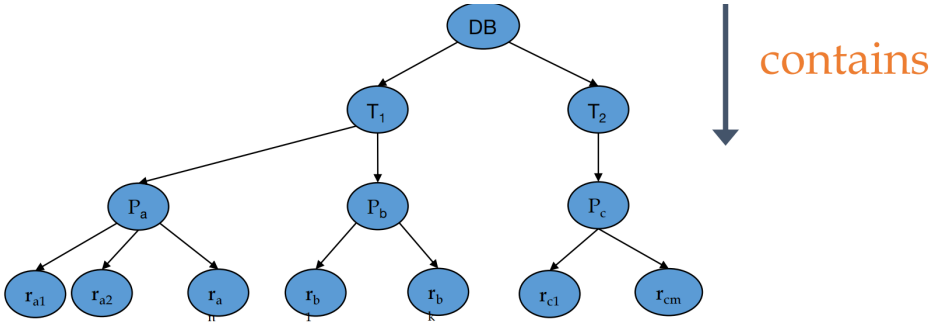
```
# ts 表示事务开始时间戳（越大表示事务越小）
# TA后来，想要拿到TB已经占有的资源
# wait-die：老 等 小
if TA.ts < TB.ts:
    TA.wait()
else
    TA.rollback()           # die
    TA.restartWithSameTs()

# wound-wait：小 等 老
if TA.ts > TB.ts:
    TA.wait()
else
    TB.rollback()           # wound
    TB.restartWithSameTs()
```

4.4.6. （不太考）锁定粒度 Lock Granularity

- 多粒度锁定 Multiple Locking Granularity
    - 允许数据项具有不同的尺寸

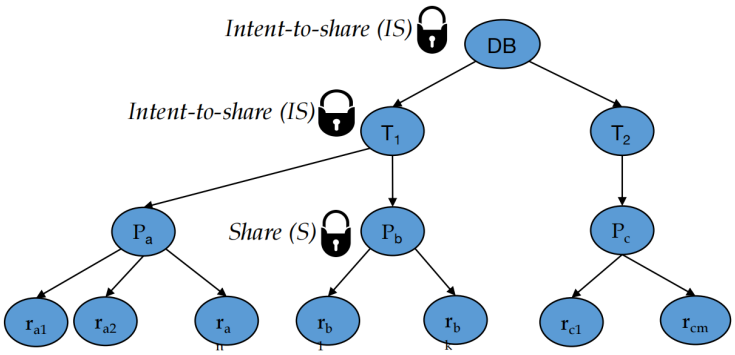
- 定义数据粒度的层次结构，小的嵌套在大的内部
  - 可以以树形结构进行图形表示。
  - 显式锁定（explictly）：该数据对象被锁
  - 阴式锁定（implicitly）：本身未被锁，但上级被锁了



- 锁粒度（进行锁操作的树中的级别）：
  - 精细粒度 Fine granularity（树中较低级别）：高并发，大量的锁
  - 粗粒度 Coarse granularity（树中较高级别）：锁数量较少，并发度小
    - 如果不需要粗粒度锁住的所有下级数据对象，则丢失潜在的并发性。

4.4.7. （不太考）意图锁（新的锁模式和协议）

- 允许事务在每个级别上锁定，使用新的“意图”锁：
- 在获得S或X锁之前，事务必须在粒度层次结构的所有祖先上具有适当的意图锁。



- 额外的3种锁模式：
  - IS 意向共享锁：一个数据对象加了IS锁，表示它的某些子孙加了或拟加S锁。
    - 某元组加了S锁，则其祖先(关系和数据库)都得加上 IS 锁，以防止其他事务在数据库和关系这一级加锁(如X锁)，导致其隐式锁和S锁冲突
  - IX 意向排他锁：一个数据对象加了IX锁，表示它的某些子孙加了或拟加X锁。
  - SIX：事务既希望以共享方式访问资源，又希望在更细粒度的层次上获取排他锁

        - 典型场景：要读整张表，但只修改部分元组
- 申请锁与释放锁规则：
    - 每个事务锁申请从根节点开始（自上向下）
        - 要获得节点上的S或IS锁，必须持有父节点的IS或IX锁
        - 要获得节点上的X或IX或SIX锁，必须持有父节点的IX或SIX锁
    - 必须按照从下向上的顺序释放锁
    - 同时还要遵循2阶段和锁兼容矩阵规则

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false