

# Disk Manager

## L3.14 Database Storage 数据库存储

1. { OS: mmap 虚存管理  
✓ DBMS dirty pages 事务性写回、预存数据、定期替换策略、线程/进程调度

2. 数据库：一个或多个 db 文件，OS 提供 API 文件读写

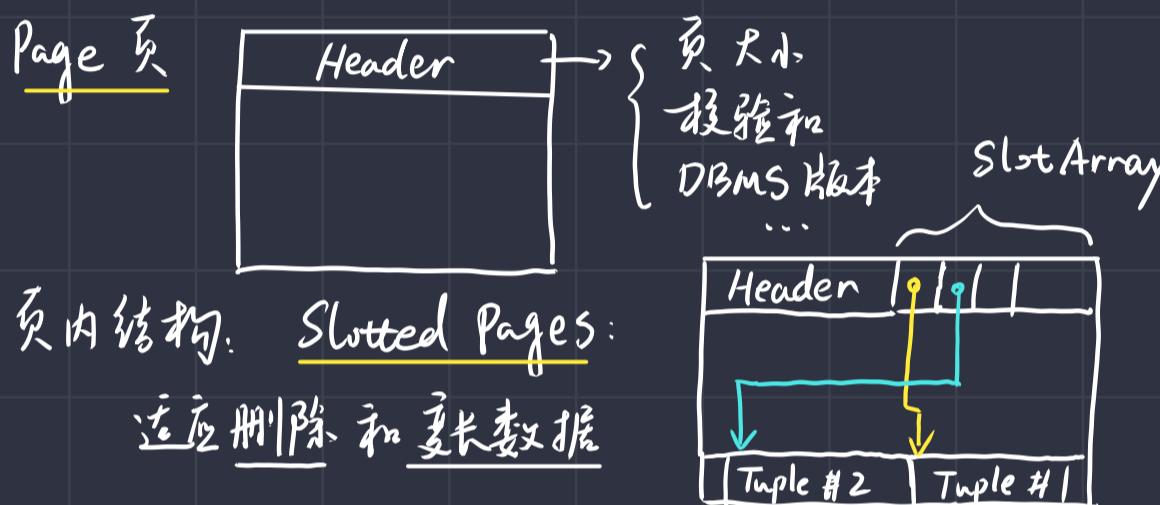
文件：若干 pages → 保存任何东西 (tuple、元数据、索引、日志)

硬件 page · OS page · DB page

3. 垂直文件组织：无序 pages 集合、文件末尾插入

管理方式：Linked List (✗) vs Page Directory (✓)

{ page 位置  
page 利余空间



内存相遇 → 页满

删除时移动数据，更新 Slot Array

(0,3)

Record Identifier 隐藏主键 (file+) page-id + offset/slot

4. 顺序文件组织:

① tuple-oriented · Tuple

元组: 

Header	属性
--------	----

 → 本质是字节序列

不需要存 schema，可能有 NULL bitmap

② log structured 日志

通常在记录 ID 上建索引，定期压缩日志

可能在其他 page

5. Tuple 与数据类型

{ int, float (IEEE 754) like C/C++

变长类型：需要前缀表示大小。

时间：秒数 / 微秒数

精度浮点：numeric/decimal 运算帽

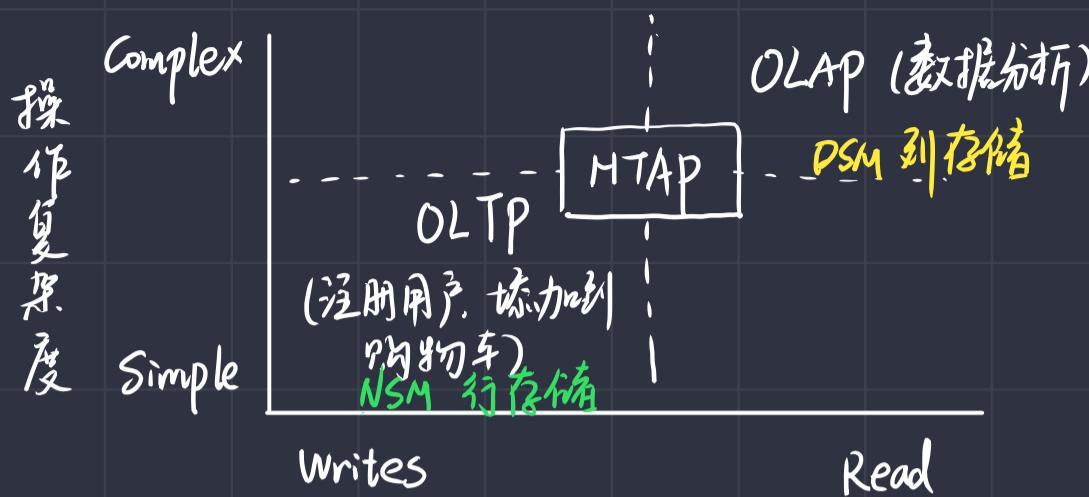
溢出页（一般只读）存大 tuple MySQL: >  $\frac{1}{2}$  size of page

外部存储：大的内容存在外部文件 元数据 → 特定数据库

## 6. Index-organized storage 只使用索引

### L5. 数据存储模型和压缩

1.



2. 列存储: (利于压缩)

如何跟踪每个 tuple 的不同属性: { offset (定长数据) → 常用  
在属性前加上额外 tuple ID → 开销大

3. 压缩 Compression: 因为磁盘 I/O 几乎总是瓶颈

希望压缩得到的特性: ① 定长数据 (不同池中可变长度数据例外)  
② 尽可能进阶压缩

③ 无损 有损压缩只能在应用层

4. 几种压缩方式:

① RLE: 单列中相同值压缩成 (属性值, 起始位置, 数量)  
智能排序 → 提高压缩率

id	v
1	A
2	B
3	A
4	A

→

id	v
1	A
3	A
4	A
2	B

→

id	v
1	(A, 1, 3)
3	(B, 4, 1)
4	
2	

② 位打包 int32 → int8

数值超过最大值特殊标记 → 查找表

③ 位图 (<10)

④ 差值 ⑤ 增量

⑥ Dictionary 字典 (最常见): 用较小代码替换数值中的频繁模式

支持范围查询

Adam	10
Alice	20
Bob	30
Candy	40

like A%

↓  
BETWEEN 10 AND 20

Buffer Pool Manager

L6. Memory Management

Buffer Pool DBMS 从 OS 申请的 memory, 自己管理

1. Buffer Pool Manager 缓冲池管理器

DBMS 维护一个 page table 页表



页表: 记录 page 在内存的位置、Dirty Flag、引用计数(使用该 page 的线程数)  
→ 本质是哈希表，通过页表和 page id 可以确定在哪个 frame 中

## 2. Latch (锁)

page 不在页表中，DBMS 申请 Latch(锁)，表示该 entry 被占用

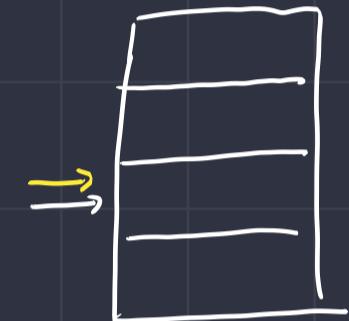
{ locks : 保护数据库 tuple、表等，事务运行时持有锁 ⇒ for 程序开发人员  
latches : OS 中的锁 → mutex ⇒ for 数据库开发

## 3. Page table 和 page directory 区别

内存中。 page id 到 page 位置的映射必须持久化

4. 缓冲池 可以让每个表都有自己的 buffer 池

5. 预读取：早点把接下来的数据读入 树型索引叶子页



扫描共享 → DB2、SQL server

游标共享 → Oracle

By Pass 额外一个内存给一个扫描量很大的线程

DBMS 告诉 OS，不要缓存这些数据 → Direct I/O

## 6. 替换策略 (一些简单的技术)

① LRU : Least recently used 最近最少使用 → 时间戳最早

实现：读写 page 放入队尾，防止 page 被换出

② Clock 时钟：LRU 的近似算法  $1 \rightarrow 0$ , 0 换出  
继续

顺序洪水问题

时间戳 → 预测下次访问的时间

③ LRU-K → 对于单个 page 对应缓存数据的访问进行计数的次数

最近使用过 K 次



↑↑ Maybe MySQL

7. Dirty Page : 脏页换出需要写回到磁盘

后台写：定期扫描 page table，将脏页写入磁盘，标记为干净

Access Method

L7 Hash Tables

DBMS 最重要的数据结构:

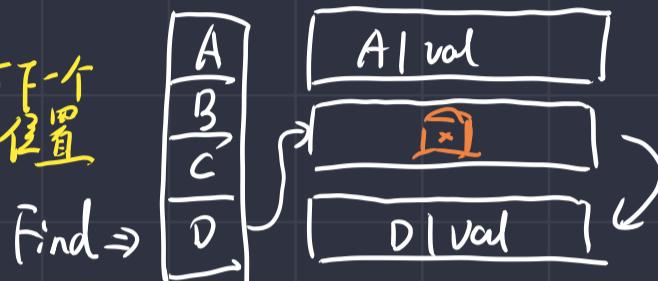
运用在 { Internal meta-data  
Core Data Storage  
Temporary Data Structures  
Table Indexes

1. 哈希函数 ( fast vs. collision rate) DBMS 哈希函数不会暴露在外

• Facebook XXHash (2012) 速度快，碰撞率低，没必要加密

2. 静态哈希方案 (估算空间大小，2倍空间)

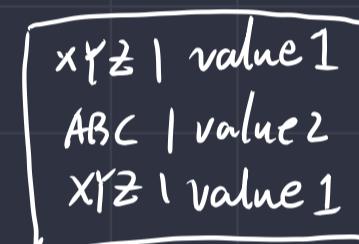
① Linear Probe Hashing 线性探测 → 位置  
删除元素 → 差碑法 ✓ / 移动法



② 一个 key 对应多个 value

维护链表法 (不推荐)

保存冗余 key ✓ (大家都这么做)



③ Cuckoo Hashing 布谷鸟哈希 (一个桶版本，两个桶片版本)

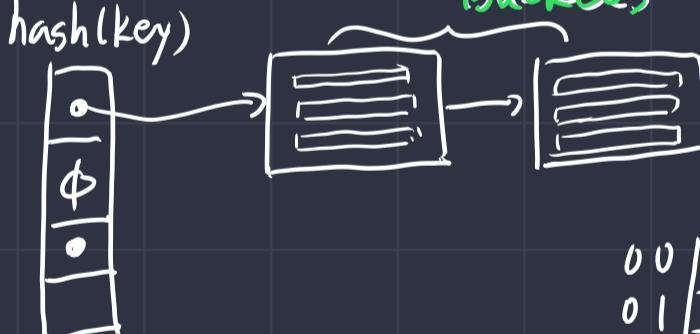
Hash<sub>1</sub>(X) 和 Hash<sub>2</sub>(X)

Hash<sub>1</sub>(X) 被应用选择 Hash<sub>2</sub>(X)

Buckets

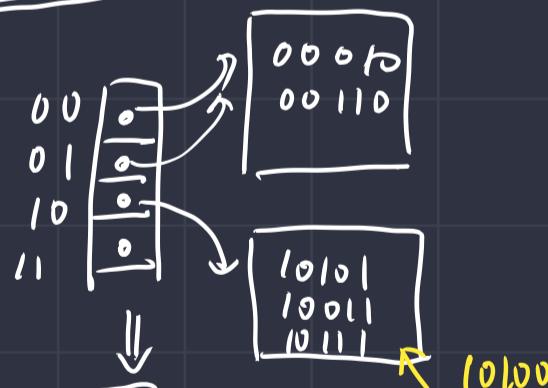
3. 动态哈希方案

① 链式哈希



② 扩展哈希表 Extendible Hashing

一边扩容，一边重置哈希表



③ Linear Hashing 线性散列

维护一个指针指向下一个将被拆分的 bucket

任意 bucket 溢出，将 (\*p) 拆分  
重新哈希

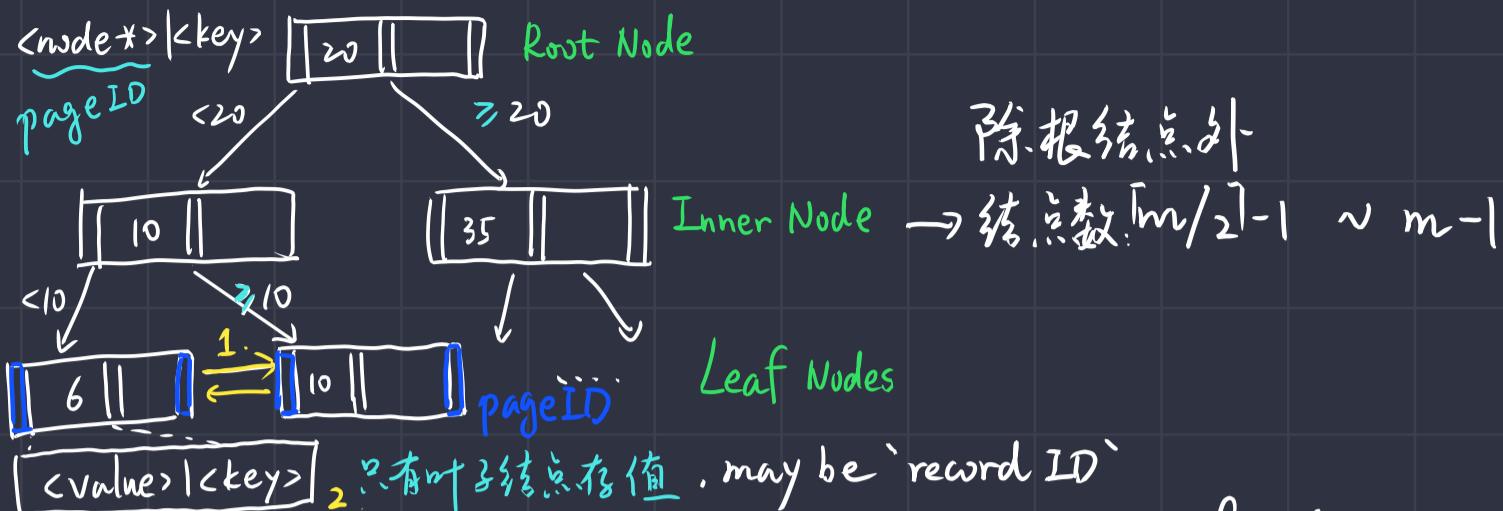


$$\text{hash(key)} = \text{key \% n};$$

$$\text{hash(key)} = \text{key \% 2n};$$

# L8 B+ Tree

## 1. B+ 树结构



Store all **NULL keys** at either first or last leaf nodes

## 1.2. 都使得范围查询更加容易

## 2. B+ 树插入



B+树删除 如果只有  $[M/2]-1$  个 entries 先借, 不行合并, 更新 parent.

## 3. 查询条件 (A, B) 先按A排, 再按B排

重复键: ① 增加 Record ID 到 key → Insert `<6, (page, slot)>`  
 ② overflow page

## 4. Clustered Indexes 聚簇索引 → SCAN 不会导致页频繁切换(有序性)

(CSDN) { 索引和数据在一块  
 最多只有一个聚簇索引}

## 5. B+ 树设计选择

Node Size 存储设备越慢, node size 越大 → HDD: ~1MB  
 → SSD: ~10KB  
 → In Memory: ~512B

Merge threshold 推迟 B+ 树结点合并 PostgreSQL: nbtree

Intra-node search { 线性:  $O(n)$ . 但用 SIMD 向量化比较 (硬件支持)  
 二分查找  
 Interpolation 估计位置 (很困难)

## 6. 优化

1) `robbed || robbing || robot` ⇒ `Prefix: rob`  
`bed || bing || ot`      Sorted keys  
 可能有相同 prefix

2) `K1 | V1 | K1 | V2 | K1 | V3 | K2 | V4` ⇒ `k1 | v1 | v2 | v3 | k2 | v4`

3) 在 innernode, 用 key 的前缀代替 key

4) Pinned 在缓冲池中的页，原指向其的 pageID 可用 pointer 替代

Insert 40

5) BULK INSERT 建立B+树 排序法

6) 延迟修改(分裂结点、代价很高) mod log

B+-trees

7. 索引使用 (PostgreSQL)

`CREATE INDEX idx_email ON emails USING BTREE(email);`

查询计划: EXPLAIN SELECT MIN(email) FROM emails; 仅从索引

L9 索引的线程安全 (volt DB → redis 单线程)

再次区分 DB 中的 locks 和 Latches → 我们现在讨论的

1. Latch Modes: { Read Mode      Write Mode

	R	W
R	✓	✗
W	✗	✗

① 自旋锁

别的线程得到 Read mode latch, 新的线程可以继续获得 read mode latch

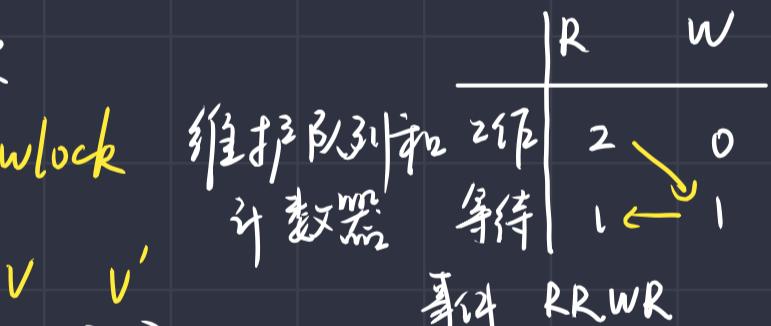
别的线程获得写 mode 的 latch, 新的线程无法获得 write mode latch

std: atomic\_flag latch;  
while (latch-test-and-set(...)) {  
 ...  
}

std: mutex → pthread-mutex → futex

OS 系统调用，开销大

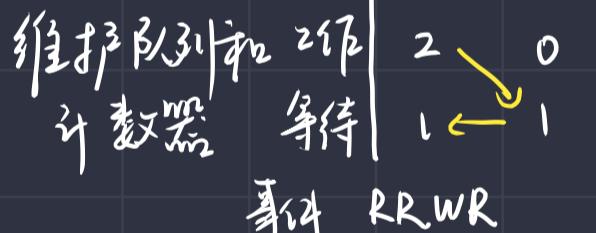
② Blocking OS Mutex



③ Reader-Writer Latches 允许并发读

std: shared-mutex → pthread\_rwlock

尽可能避免 OS



3. --sync-bool-compare-and-swap(&M, 20, 30)

判断 M 和 V 是否相等, true ⇒ M = V'

新值

4. Hash table Latching 按序移动 → 不会有死锁

resize → 锁住哈希表

两种锁 { Page Latches

共享 { Slot Latches

移到下一个 page 解锁 / 移动到下一个 Slot 解锁

5. B+树并发控制

latch coupling/crabbing

{ 得到 latch for parent  
 得到 latch for child  
 神经 latch for parent if "safe" → 无 split 无 merge  
 插入不满 删除多余 half full

Find: Start at root and traverse down the tree:

- Acquire R latch on child,
- Then unlatch parent.
- Repeat until we reach the leaf node.

Insert/Delete: Start at root and go down, obtaining W latches as needed. Once child is latched, check if it is safe:  
 → If child is safe, release all latches on ancestors

第一件事，给 root 加上 latch  $\Rightarrow$  性能瓶颈

b. 乐观上锁 大部分其实不会导致 split 和 merge

优化 假设 leaf node 是安全的，一路获取、释放 R latch

{ 确实安全，对叶节点、W-latch，释放  
 split/merge，重新执行，获取、释放 W-latch

## 7. 左右方向的访问模式

并发读不会造成影响

(kill别人较为困难)

横向扫描无法获取下一个节点 latch, 释放 latch to 杀死自己

syscall

## Operator Execution

### L10. Sorting & Aggregation Algorithm

1. 查询计划：操作符组织成树

2. 算法：面向磁盘，数据在内存中处理。数据流

顺序 I/O 好于 随机 I/O

$\Rightarrow$  排序 (ORDER BY, DISTINCT, GROUP BY, 插入 BT 树) S

3. 排序：In-memory sort  $\rightarrow$  Quicksort

4. TOP-N Heap Sort ORDER BY with LIMIT

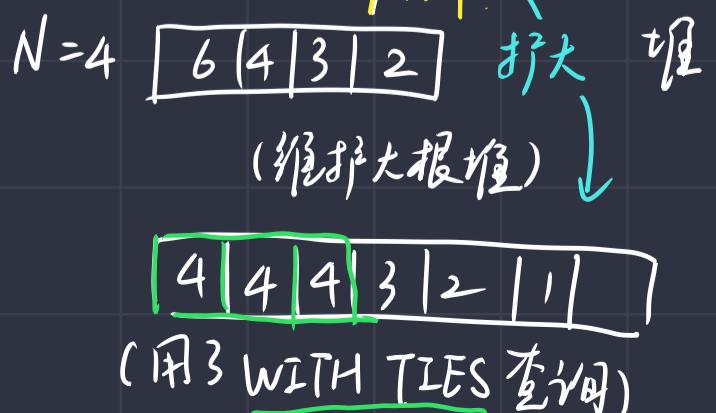
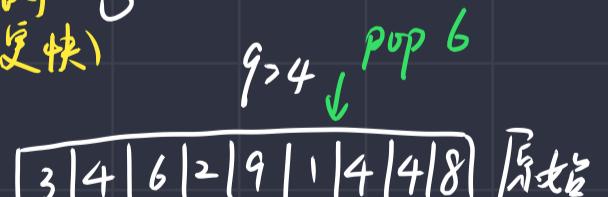
建立内存中的堆

e.g. Select \* from enrolled  
order by sid

Fetch FIRST 4 Rows  
WITH TIES

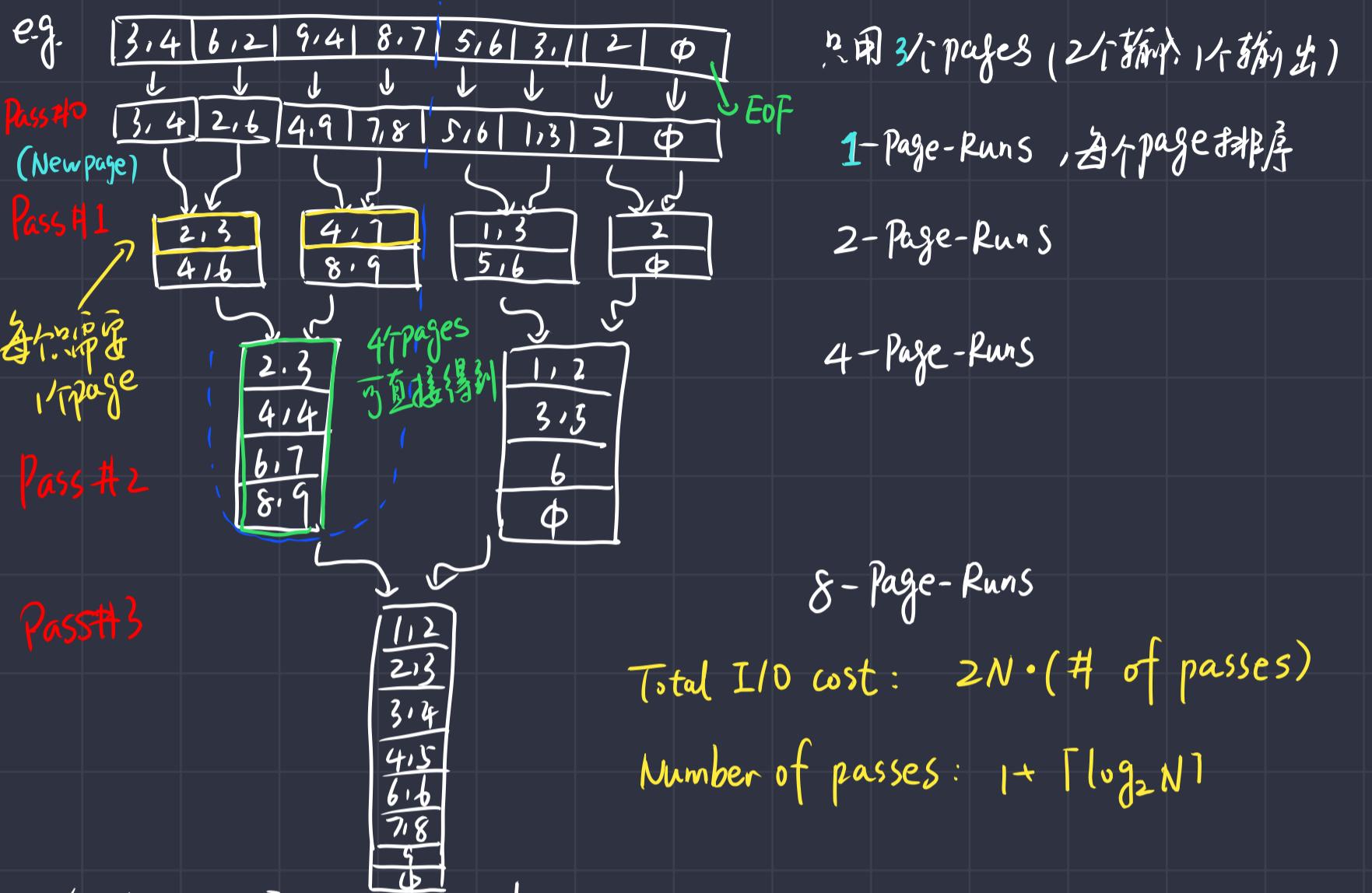
结果

$\pi_{R.id, S.colate}$



5. EXTERNAL MERGE Sort { #1 Sorting  
排序-多个表(非常大) → 分治 { #2. Merging

2路外部归并排序 (数据有  $N$  个 pages,  $B$  个缓冲池 pages)



6. 优化: 大于 3 个 buffer pool pages

直接得到 4 pages 排序, 多路归并

Pass 0 Use  $B$  buffer pages, Produce  $\lceil N/B \rceil$  sorted runs of size  $B$

Pass 1, 2, 3: Merge  $B-1$  runs  $\# \text{Passes} = 1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil$

7. Double buffering: CPU 和 I/O, 有一个不干话

在合并时, CPU 会 fetch 接下来要用的 pages 到 second buffer

8-比较函数 (不同数据类型不同 cmp-) hardcoded C++ template

VARCHAR: 比较 prefix, 失败再比较 string

9. 使用 B+ 树排序: traversing (遍历) the leaf pages of the tree

① Clustered B+ TREE: 顺序, 无比较 从最左叶节点开始往右

② Unclustered B+ TREE: (bad idea) 反复页面切换

10. Aggregations: (对 tuple 的某些值做统计) 聚合查询

两种方法: { Sorting Filter → Remove Columns → Sort  
Hashing }

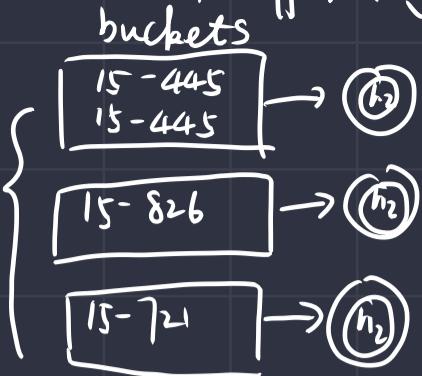
有时不需要对数据排序 → Hashing 利用一个临时哈希表 DISTINCT, Group by

B-1 buffers 用于 partitions | 1 buffer 用于 input data

#1. PARTITION  $h_1$

#2. REHASH  $h_2$

GROUP BY 求均值: Phase #1



Hash Table	
key	value
15-445	(2, 7.32)
15-826	(1, 3.33)
15-721	(1, 2.89)

$$\frac{7.32}{2} = \text{AVG}$$

## L11. Join Algorithm

Select R.id, S.cdate

FROM R, S  
Where R.id = S.id  
And S.value > 100

1. Early Materialization:

R.id	R.name	S.id	S.value	S.cdate

Late Materialization:

R.id	R.RID	S.id	S.RID

Record IDs 与以换取 total I/Os

对 Column Stores 友好

## 2. 代价分析 (I/O)

(无需拷贝查询不需要的数据)

假设:  $\rightarrow M$  pages in table R,  $m$  tuples in R

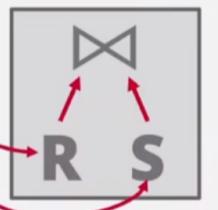
$\rightarrow N$  pages in table S,  $n$  tuples in S

Join 算法: { Nested Loop Algorithm  
Sort-Merge Algorithm  
Hash Join

## 3. Nested Loop Join Algorithm 嵌套循环

Simple:

```
foreach tuple r ∈ R:  
  foreach tuple s ∈ S:  
    if r and s match then emit
```



Cost:  $M + (m \cdot N)$

如果交换一下:  $N + (n \cdot M)$  可能更优

Block: 将 table 为 Outer table (指 pages)

```
foreach block  $B_R \in R$ :  
  foreach block  $B_S \in S$ :  
    foreach tuple r ∈  $B_R$ :  
      foreach tuple s ∈  $B_S$ :  
        emit, if r and s match
```

Cost:  $M + (M \cdot N)$

考虑缓冲池大小为 B,

一个用于 output, 一个用于 inner table.

$$\text{Cost} = M + \lceil \frac{M}{B-2} \rceil \cdot N$$

Index -

```
foreach tuple r ∈ R:  
  foreach tuple s ∈ Index( $r_i = s_j$ ):  
    emit, if r and s match
```

$$\text{Cost} = M + (m \cdot C)$$

使用索引

## 4. Sort-Merge Join

算法：

```

sort R, S on join keys
cursorR ← Rsorted, cursorS ← Ssorted
while cursorR and cursorS:
    if cursorR > cursorS:
        increment cursorS
    if cursorR < cursorS:
        increment cursorR backtrack cursorR if necessary
    elif cursorR and cursorS match:
        emit
        increment cursorS
    
```

Phase #1 : Sort (按连接时的属性, e.g. id 排序)  
 Phase #2 : Merge  
 过程:

R(id, name)	
id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface

S(id, value, cdate)		
id	value	cdate
100	2222	10/4/2023
100	9999	10/4/2023
200	8888	10/4/2023
400	6666	10/4/2023
500	7777	10/4/2023

代价: Sort Cost( $R$ ) :  $2M \cdot (1 + \lceil \log_{B-1} \lceil M/B \rceil \rceil)$   
 Sort Cost( $S$ ) :  $2N \cdot (1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil)$   
 Merge Cost =  $M+N$

最坏情况(值都相同): Cost =  $(M \cdot N) + (\text{Sort cost})$

## 5. Hash Join

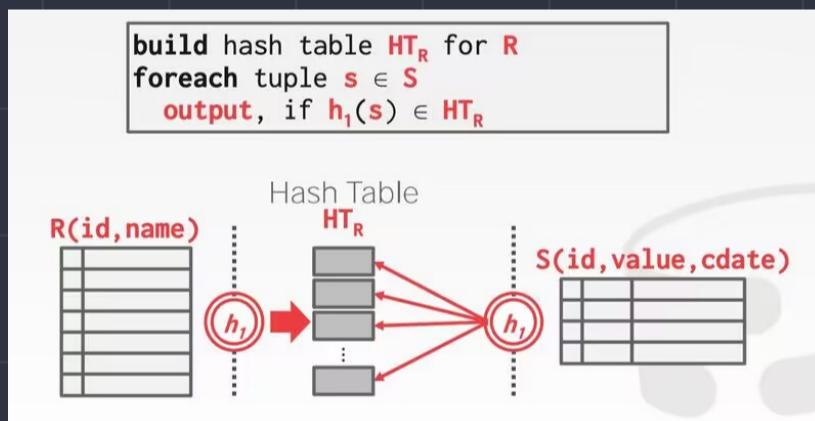
almost always better

Join 值相等, 经过哈希后也相等

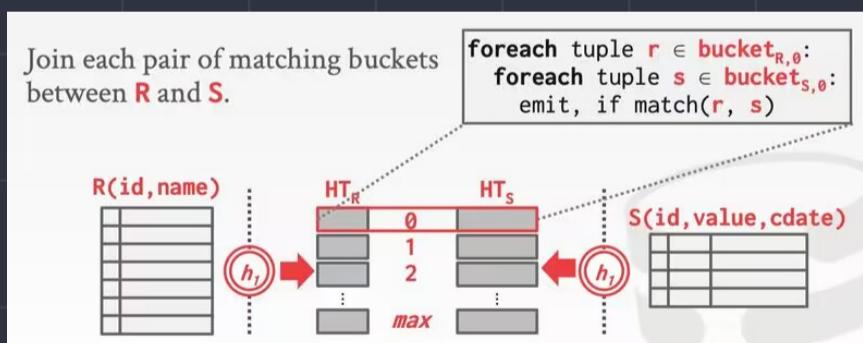
Bloom Filter → in CPU cache  
 判断 key 是否存在

缺点: Hash Table  
 可能无法放进内存,  
 In memory 和 disk 之间  
 来回移动

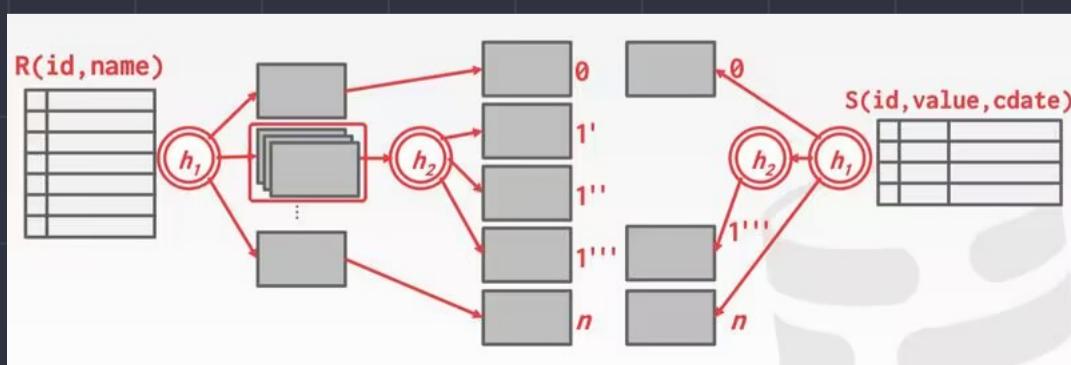
### ①朴素哈希 Join



### ② Grace Hash Join



### ③ Recursive partition, 再来一个 hash



Partition Phase

Hash Join 代价  $3(M+N) <$  Read, Write Both tables  $2(M+N)$   
 Read Both tables  $(M+N)$  Probing Phase

## L12. Query Execution I

1. Processing Model : 系统如何执行查询计划

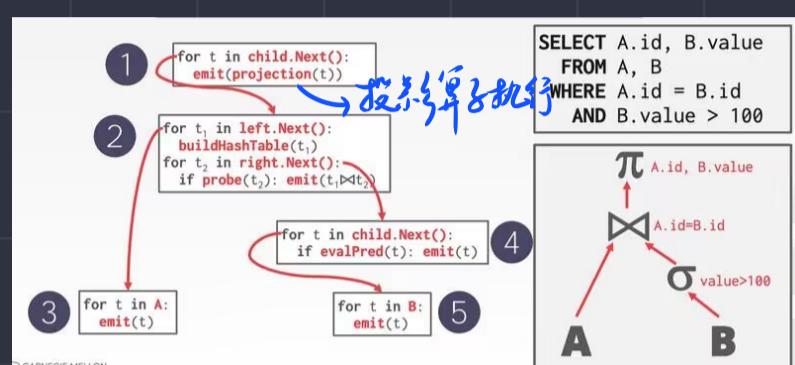
① Iterator Model 迭代模型 (火山管道模型)  $\Rightarrow$  几乎被用在每个DBMS

每步 operator 都实现一个 next 函数, 每次调用返回一个 tuple 或 null

operator 本身是一个循环, 每次调用 child 的 next 函数取一条数据

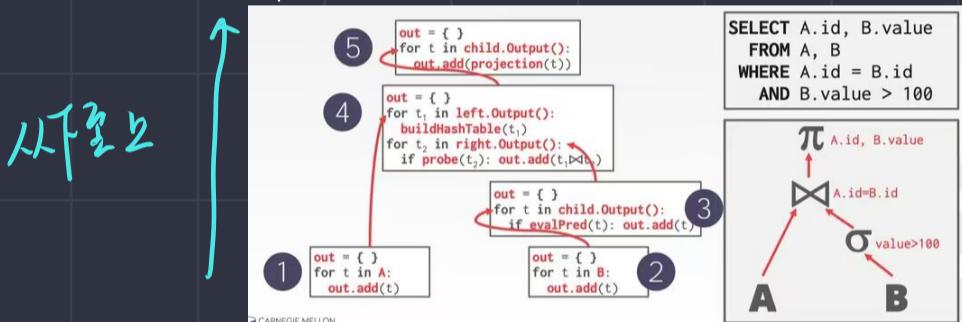
进行操作

从上到下



② Materialization Model 物化模型

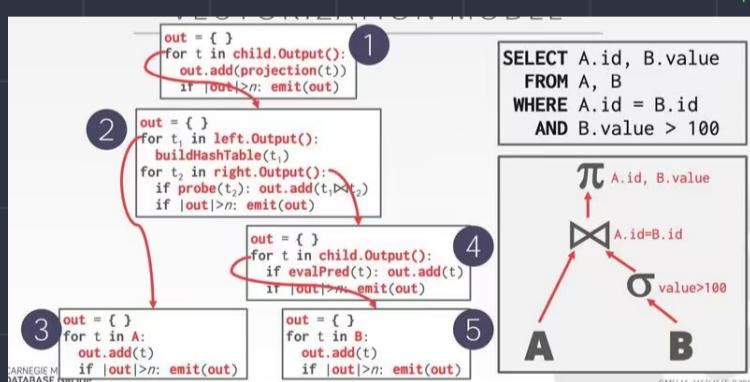
operator 处理完所有输入后, 将结果一次性输出(数组)



③ Vectorization Model 向量化模型 (以上两种模型的升级)

operator 实现了一个 next 函数, 每个 next 函数返回一批数据  $\Rightarrow$  OLAP 友好

Intel 处理器可以向量化运算  $\rightarrow$  并行



Batch operator 调用次数

使用有: VectorWise, SQL Server, Oracle, DB2 ...

Sequential Scan

Index Scan

Mult:Index / "Bitmap" Scan

2. Access Method 访问方法:

DBMS 从数据表中获取数据的方式 (关系代数中没定义)

① Sequential Scan 全表顺序扫描.

```
for page in table.pages:
    for t in page.tuples:
        if evalPred(t):
            // Do Something!
```

$\rightarrow$  需要一个游标 cursor 记录迭代器那

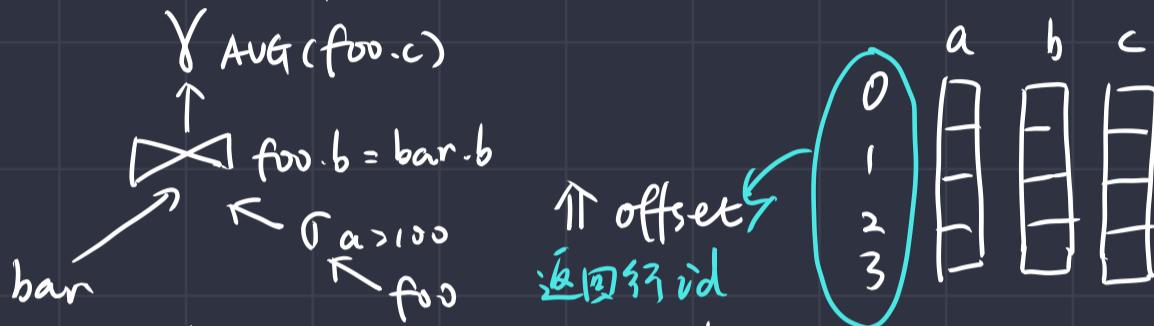
优化：a. Prefetching 预加载 b. 缓冲池 c. 并行执行 (多算子查表)

d. Zone Maps 为每个 page 计算好属性值的统计 → 先检查 zone map

Select * from table	val	Zone Map	缺点：
where val > 600;	$\begin{matrix} 100, 200, 300 \\ 400, 400 \end{matrix}$		① 占用空间
in SQL Server, Oracle, DB2, MySQL 等		page MAX 为 400, 不用访问	② 更新麻烦
		type   val	
		MIN   100	
		MAX   400	
		Avg   280	
		Sum   1400	
		Count   5	

e. Late Materialization 延物化

到存储 DBMS 中，operator ? 选取查询需要的数据，仅需传递 offset (行 id)



f. Heap Clustering : 使用聚簇索引时，tuple 在 page 中顺序排列。

若查询语句被索引的属性 ⇒ 跳跃访问

② Index Scan 抓指索引

DBMS 如何选择 index =>  
排序优化，避免页面抖动，按 page id  
排序

{ 索引、查询条件包含哪些属性  
attribute 的定义域  
索引是否包含输出列  
谓词压缩  
索引是否是 unique key }

越早过滤掉越多 tuples 越好  
多个索引选择一个

③ Multi-index Scan 多索引扫描 Postgres 称为 Bitmap Scan

如果有多个索引可以部署，取并集 merge record ids → 更快

3. 修改查询：要检查约束、更新索引

索引 ⇒ 避免对数据重复处理  
更新游标记录是否处理过

跟踪每个查询修改的 record ID → keep in memory

4. 表达式计算

expression tree ← WHERE clause

SQL parser → AND  
= / / +  
Attribute(R.id) S.id S.value 100

存在很大效率问题，例如每次计算  $? + 1$

WHERE 1 = 1 JIT 高效数据库会干掉这种语句

# L13 Query Execution II 并行执行

1. 并行执行：{ 提升吞吐量 潜在降低总拥有成本  
降低延迟 }

并行分布式：{ Parallel：运行在多核CPU上，通信成本低  
Distributed：节点距离远，通信成本高。部署依赖于访问单个节点。 }

2. Process Model：系统如何组织并发查询

① Process (进程) per worker

依赖OS dispatcher，使用共享内存。进程crash不会破坏整个系统

使用Process Pool



② Thread per DBMS worker 线程 → 效率高，上下文切换代价低

线程调用由DBMS自行负责（DBMS比OS有更多知识），进程中内存共享

缺点：一个thread crash了，整个系统崩溃

e.g. DB2, MySQL, Oracle (2014) 等

DBMS 需要调度 Scheduling：  
执行计划切分成多少 task

每个task占用多少CPU。哪些CPU执行哪些task等

补充：SQLOS 用户级别的OS层，在DBMS内部运行并管理硬件资源

③ Embedded DBMS 嵌入式DBMS

DBMS 和应用程序在相同地址空间，应用程序负责线程、调度

3. 并发执行

① Inter-query Parallelism 并行执行多个SQL查询

{ 多个查询只读 ⇒ 没问题 }

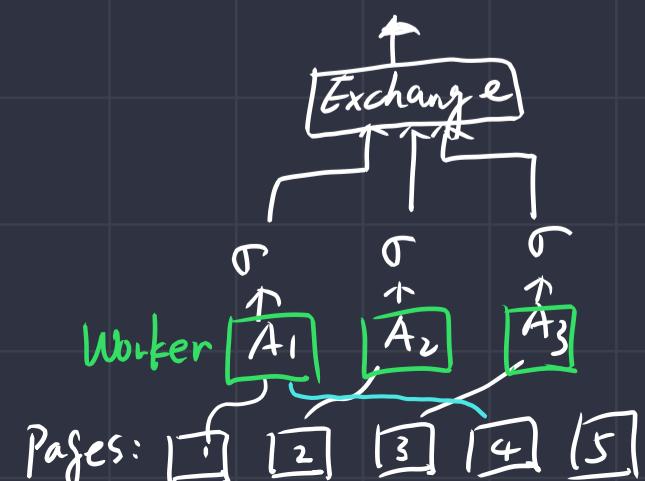
{ 并发查询存在更新 ⇒ 并发控制一章看 }

② Intra-query Parallelism 并行执行单个查询

Approach #1: Intra-operator (Horizontal 水平)

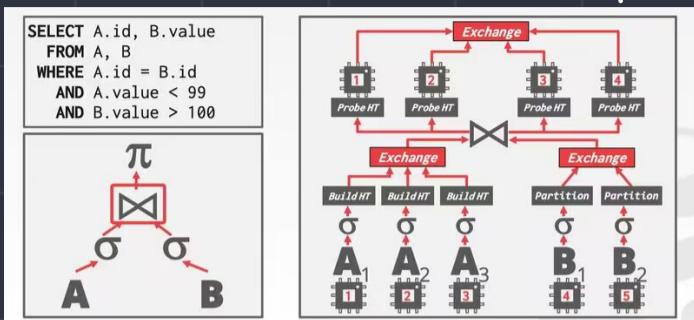
{ 将数据拆成若干子集，并行执行 }

{ 使用 exchange operator 合并子集处理结果 }



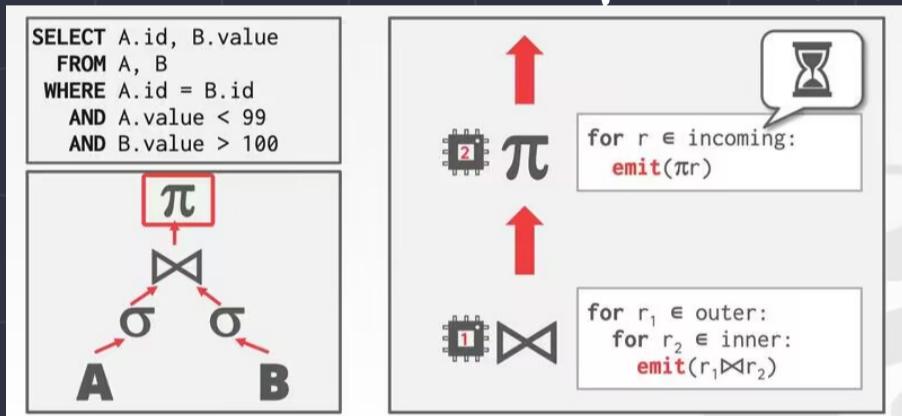
exchange 算子分类:

Gather	$\rightarrow 1$
Distribute	$1 \rightarrow \text{多个}$ (拆)
Repartition	$\text{多个} \rightarrow \text{多个}$ (再划分)

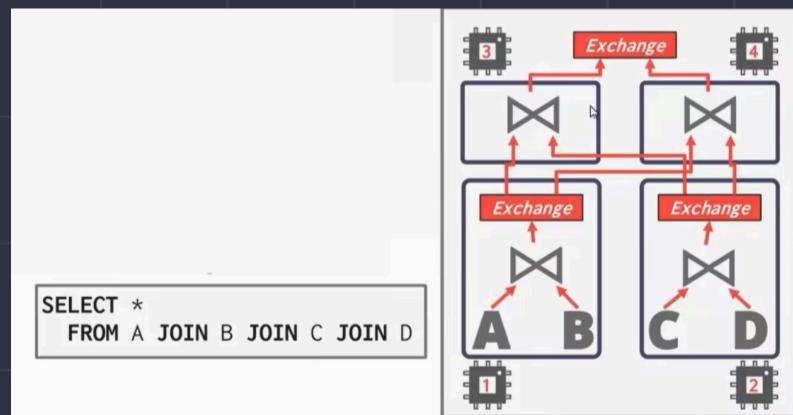


Approach #2: Inter-operator (垂直拆分) 流式处理  
生产者消费者问题 1) 把不同算子交给不同 worker 处理  
2) 一般无须等待前一步完成

常见用在流处理系统: Spark, Mifi, Kafka 等



数据流动方向:  
↑ 消费者  
↓ 生产者



Approach #3 混合模式: 同时存在两种并发

#### 4. 磁盘 I/O 并发

数据库性能瓶颈  $\Rightarrow$  磁盘 I/O

可以将数据库表存放在一个磁盘，一个表切分在不同磁盘

RAID 0 / RAID 1

#### 5. 分表

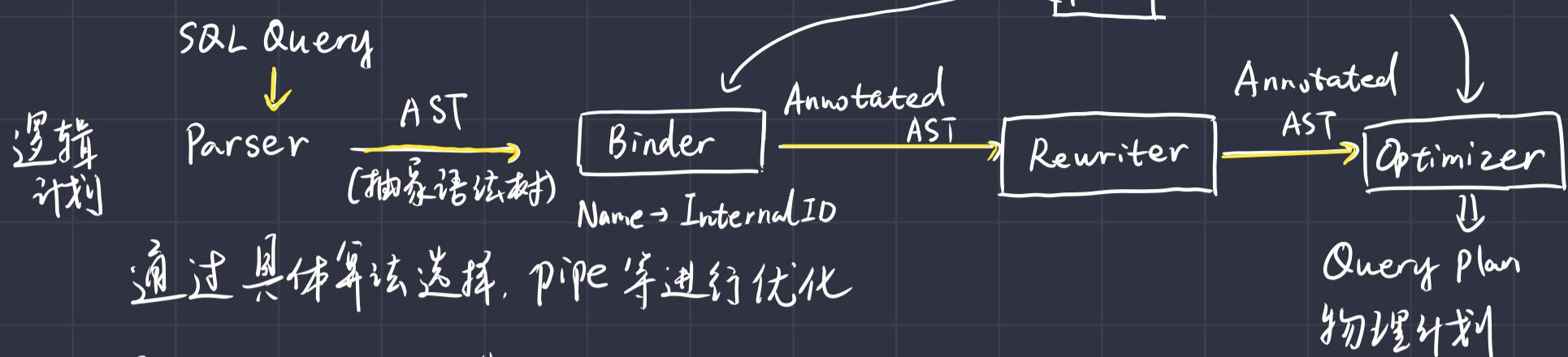
将 logical table 打分成多个 physical segments 分开存储

理想情况: partitioning 对应用透明 (不一定能实现)

{ Hash Partitioning  
Range Partitioning  
Predicate Partitioning

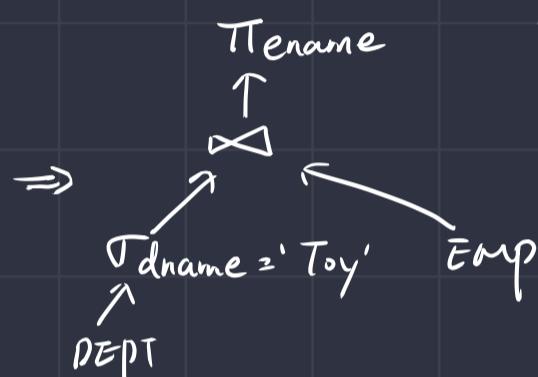
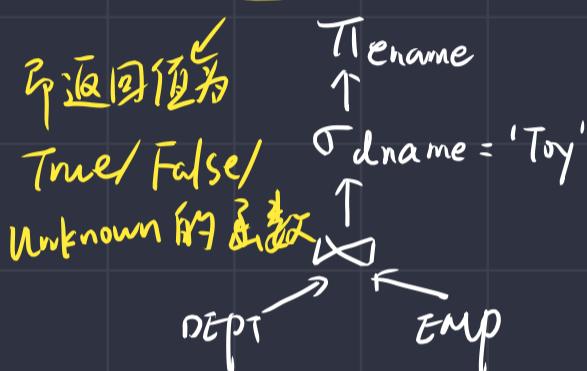
# L14 查询优化

## 1. SQL → 查询流程:



## 2. Query Rewriting (基于启发式和规则)

### ① **谓词下移** Predicate Pushdown



- 1) 不一定优化性能,  $\bowtie$  后元组数量很少的情况
- 2) 越早过滤越好  $X=Y \text{ AND } Y=3 \Rightarrow X=3 \text{ AND } Y=3$

$$\pi_{ename}(\sigma_{dname='Toy'}(DEPT \bowtie EMP)) \Rightarrow \pi_{ename}(EMP \bowtie \sigma_{dname='Toy'}(DEPT))$$

### ② 替换笛卡尔积



### ③ 投影 (Projection) 下移



### ④ 等价变换

$$\begin{cases} R \bowtie S = S \bowtie R \\ (R \bowtie S) \bowtie T = R \bowtie (S \bowtie T) \end{cases}$$

$$\sigma_{P_1 \wedge P_2 \wedge P_3}(R) = \sigma_{P_1}(\sigma_{P_2}(\sigma_{P_3}(R)))$$

## 3. Cost-Based Search

查询优化是一个 NP hard 问题

SYSTEM R OPTIMIZER { 自底向上  
自顶向下 }

嵌套子查询 → 层次化

Select name from sailors as S

where EXISTS (

Select \* from reserves as R 层次化

where  $S.sid = R.sid$

and  $R.day = '2018-10-15'$ );

Select name from sailors As S

reserves As R

where  $S.sid = R.sid$

and  $R.day = '2018-10-15'$ ;

嵌套子查询：解耦

如查询子条件为  $S.rating = (\text{SELECT MAX}(S2.rating) FROM sailors S2)$

重写定值：Select \* from A where  $l=0$ ;  $\times$

4. cost model { Choice #1 : Physical Cost

Choice #2 : Logical Cost

Selection Cardinality DBMS 保存一些统计信息  $\Rightarrow$  估计查询成本

① 统计次数、最值信息



还可以分为若干 buckets

利用最值评估： $\text{sel}(A>, a) = \frac{A_{\max} - a}{A_{\max} - A_{\min}}$

② 采样：根据采样表进行估计

L15 并发控制

→ 一系列 read to write

1. Transactions 事务：DBMS 状态变化的基本单位，由一条或多条 SQL 语句构成

事务的 ACID 特性：

{ A: Atomicity: "all or nothing" 原子性  
C: Consistency: "it looks correct to me" 一致性 前后一致 < 100ms  
I: Isolation "as if alone" 隔离性  
D: Durability: "survive failures" 持久性

2. 原子性：事务执行的两种结果：Commit 和 Abort

保证原子性：① Logging (常见) ② Shadow Paging (复制一份，Commit 后才可见) (少见)

3. 隔离性：有两个事务（假设  $A = B = 1000$ ）

$T_1$ : BEGIN  
 $A = A - 100$   
 $B = B + 100$   
COMMIT

$T_2$ : BEGIN  
 $A = A * 1.06$   
 $B = B * 1.06$   
COMMIT

两种执行顺序（均为正确）：

$$T_1 \rightarrow T_2: A + B = 2120$$

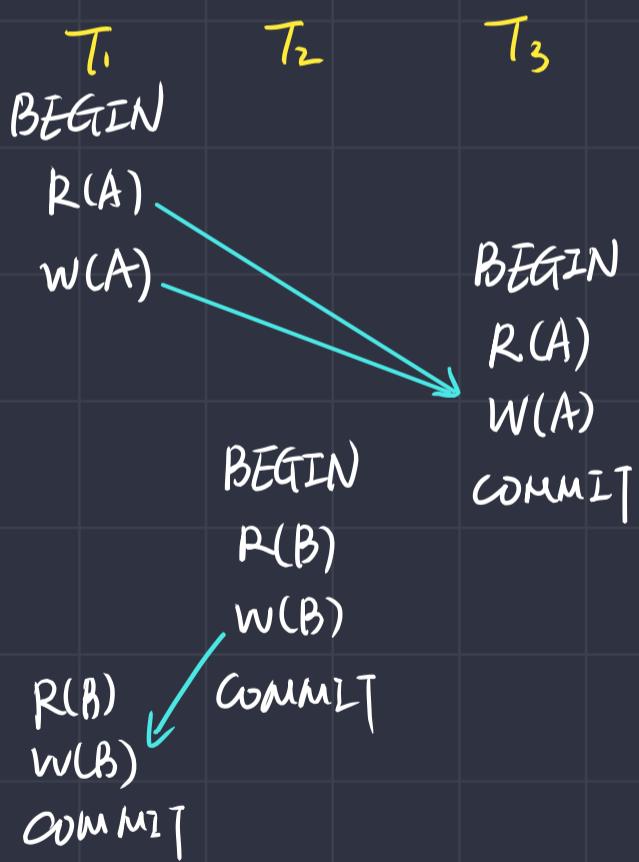
$$T_2 \rightarrow T_1: A + B = 2120$$

如何两个事务重叠运行，只有与某一种串行调度结果一致，才是正确的

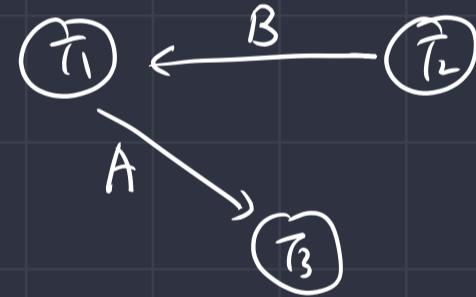
$\Rightarrow$  可串行化 Serializable Schedule

冲突操作：对同一个对象 R-W, W-R, W-W  $\Rightarrow$  不可重复读、脏读、覆盖写

冲突可串行化（交换法）、依赖图：



Dependency Graph



串行化：( $T_2, T_1, T_3$ )

## L16. 两段锁并发控制

基于锁的悲观控制方法

1. Basic Lock type 数据库有数十种不同的锁  $\Rightarrow$  组成锁的 相容矩阵

{ X-lock 排它锁  
S-lock 共享锁

	S	X
S	✓	✗
X	✗	✗

锁升级：S-lock  $\rightarrow$  X-lock

# Lock Manager (一个内存中的数据结构，哈希表)

维护哪个事务持有哪把锁，哪些事务在等待

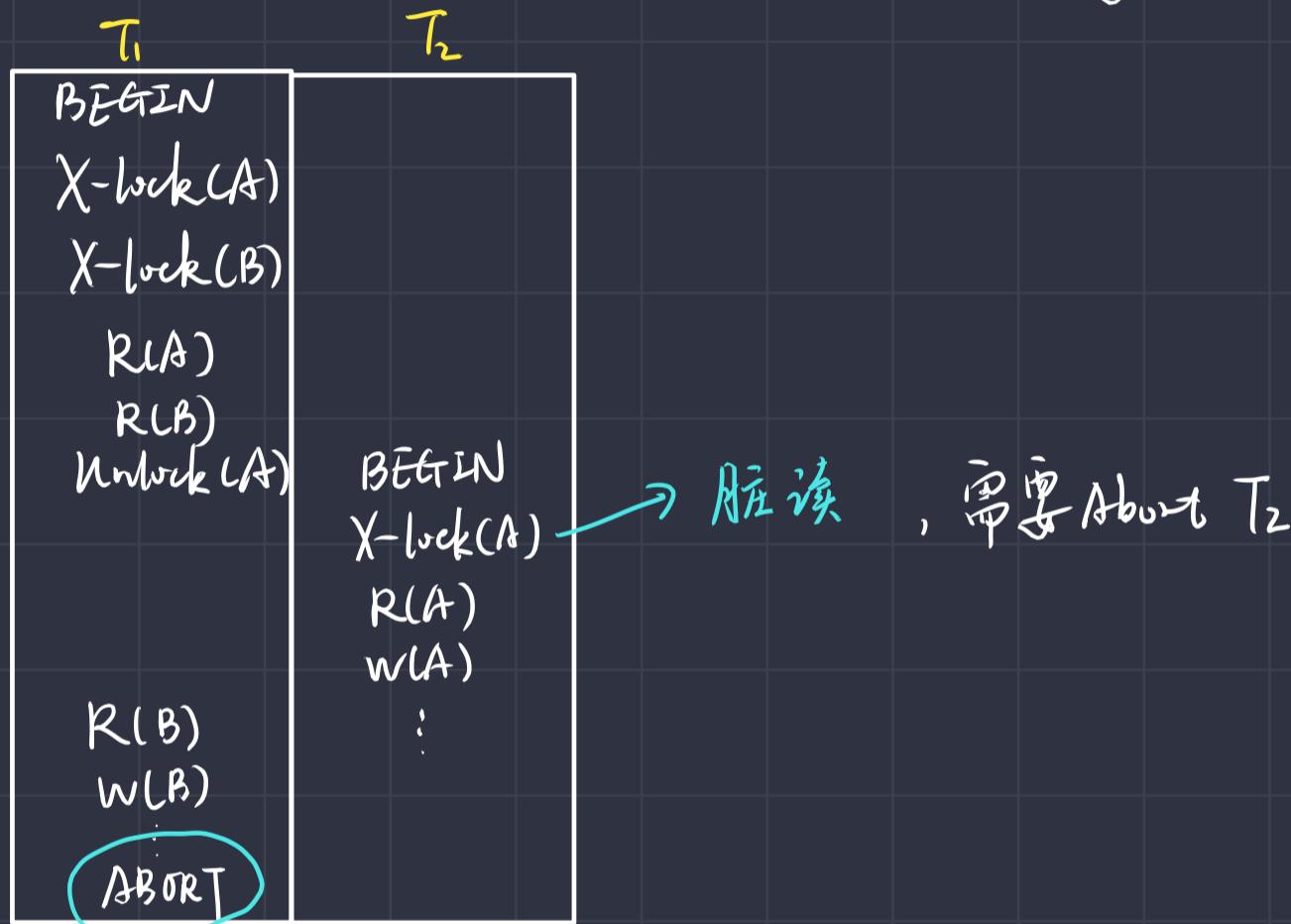
仅加锁不能得到正确的冲突可串行化调度

## 2. 两阶段锁 Two-Phase locking (2PL)

{ Phase #1: Growing      释放锁后无法再申请锁

{ Phase #2: Shrinking

可以保证冲突可串行化，会受到 cascading aborts 级联中断



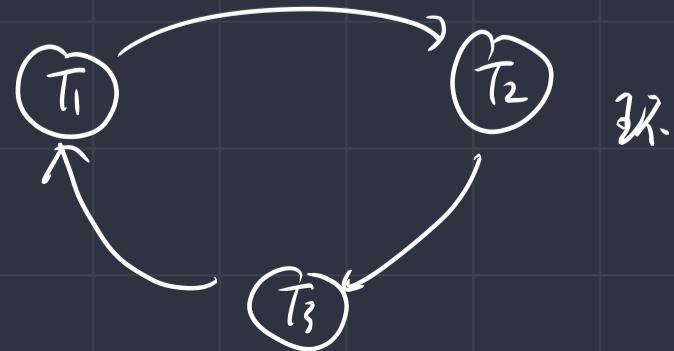
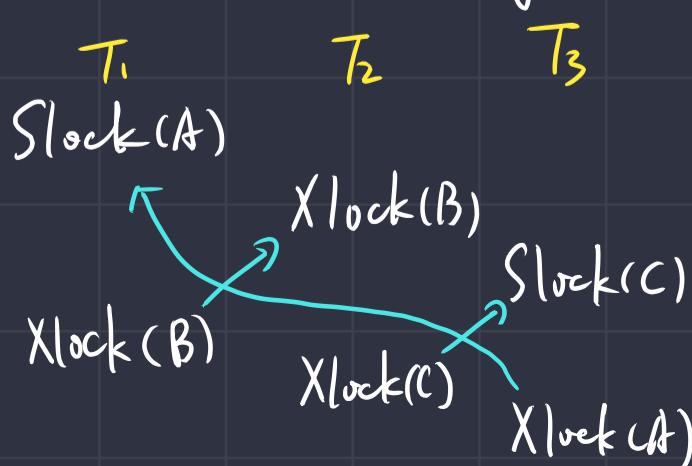
Strong STRICT TWO-Phase lock : 只在结束时释放锁

## 3. 死锁 遵守 2PL 仍会死锁

Approach #1: Dead Lock Detection 后台线程处理

Approach #2: Dead Lock Prevention

等待图 (由 Lock Manager 构建)      Waits-For Graph



代价优化：更频繁去检测大小为 2 的 cycle

死锁处理：选择一个事务撤销，Roll back Length → Save Points

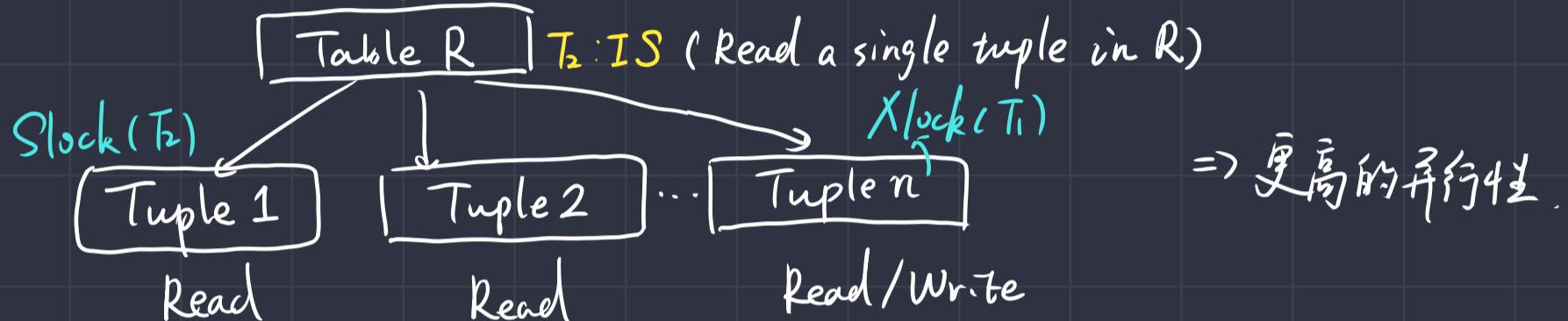
死锁预防：Wait-Die / Wound-Wait

4 Lock Granularities 如果没有层次结构，更新+检索数据，lock manager 使用较少 在此之后加锁

数据库层次结构： Database → Table → Page → Tuple → Att } 摆樣了

意向锁：下一层结构中要加锁：IS . IX , SIX

T<sub>3</sub>: S-lock (等待) T<sub>1</sub>: SIX



意向锁兼容矩阵：

	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	✗
IX	✓	✓	✗	✗	✗
S	✓	✗	✓	✗	✗
SIX	✓	✗	✗	✗	✗
X	✗	✗	✗	✗	✗

通常应用程序员不会手动控制 Lock，但数据库提供了这些锁的非标准SQL

## L17. 时间戳顺序并发控制 (悲观)

Each txn 被指派一个 unique fixed timestamp (自动增加) TS(T<sub>i</sub>)

### 1. BASIC TIO

对每个对象标记最后一个事务成功读/写时间戳：R-TS(X), W-TS(X)

① Read: Don't read stuff from future

If TS(T<sub>i</sub>) < W-TS(X) :

Above T<sub>i</sub>, restart it with a new TS

else : Allow T<sub>i</sub> Read X

$$R-TS(X) = \max(R-TS(X), TS(T_i))$$

局部 copy X 保证 T<sub>i</sub> 的可重复读

② Write: If  $TS(T_i) < R-TS(X)$  or  $TS(T_i) < W-TS(X)$

Abort and restart  $T_i$

Else:

Update  $W-TS(X)$  允许写

局部 copy  $X$  保证可重复读

例子(需要 Abort)

$T_1$	$T_2$
BEGIN $TS=1$	
R(A)	BEGIN $TS=2$
	W(A)
(W(A))	COMMIT
R(A)	
COMMIT	

Database

Object	R-TS	W-TS
A	1	2
B	0	0

违背  $TS(T_i) < W-TS(A)$

DBMS 需要 Abort  $T_i$  也可能升级颗粒粒度

拓展: Thomas Write Rule

(没有人用这个方法写分布式管理系统)

缺点: 频繁、复制数据产生成本

## 2. 乐观控制法 OCC

冲突是短暂且少见的

(有点像 Git)

Phase #1: Read Phase: 追踪记录每个事务读写集合，并存储到私有空间

Phase #2: Validation Phase: 事务提交时，检查冲突。

Phase #3: Write Phase: 检验成功→合并；否则中止并重启事务

例子: Schedule

$T_1$	$T_2$
BEGIN	BEGIN
READ R(A)	READ R(A)
	VALIDATE
	WRITE (失败)
	COMMIT
W(A)	
VALIDATE	
WRITE	
COMMIT	

Database

Object	Value	W-TS
A	123	0
-	-	-

$TS(T_1) = 1$

$T_1$  Workspace

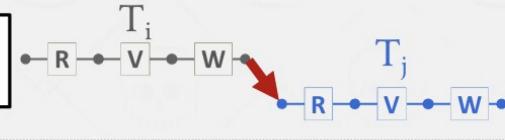
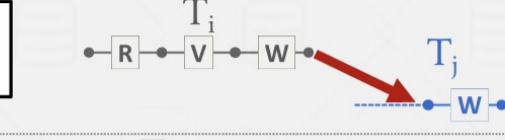
Object	Value	W-TS
A	123	0
-	-	-

$T_2$  Workspace

Object	Value	W-TS
A	123	0
-	-	-

本地副本

若  $TS(T_i) < TS(T_j)$  则三个情况其一必须被满足 VALIDATE 阶段

OCC: VALIDATION ( $T_i < T_j$ )			
	$R \rightarrow W$	$W \rightarrow R$	$W \rightarrow W$
Case 1		✓	✓
Case 2		✓	WriteSet( $T_i$ ) $\cap$ ReadSet( $T_j$ ) = $\emptyset$
Case 3		✓	WriteSet( $T_i$ ) $\cap$ ReadSet( $T_j$ ) = $\emptyset$
		WriteSet( $T_i$ ) $\cap$ WriteSet( $T_j$ ) = $\emptyset$	WriteSet( $T_i$ ) $\cap$ WriteSet( $T_j$ ) = $\emptyset$

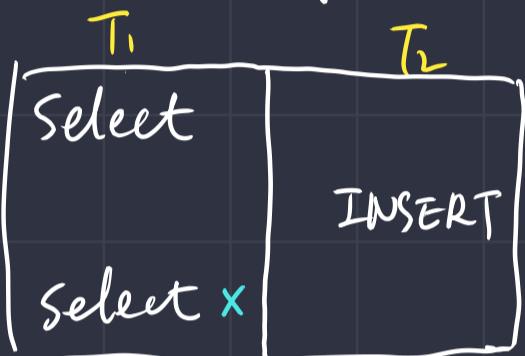
$T_i$  检查 All  $T_j$  ( $T_i < T_j$ ) in the future

缺点：本地副本复制代价，validate 阶段 Abort 代价

### 3. dynamic database

插入记录的幻读问题：

新生成的记录 不会被锁住

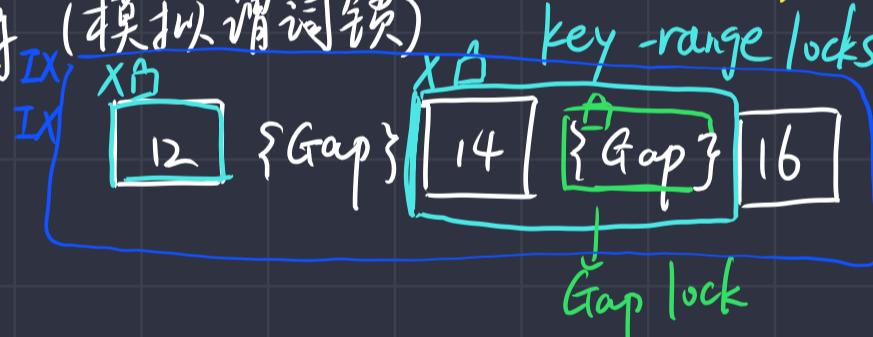


冲突可串行化保证事务可序列化前提  $\Rightarrow$  数据集合固定

Approach #1: Re-Execute Scans 在提交时重新查一遍 不一致  $\rightarrow$  Abort

Approach #2: Predicate Locking

Approach #3: Index Locking (模拟谓词锁)  $\xrightarrow{\text{同一个方向}}$  key-range locks  
 为 B+ 树中的 key 和 Gap



### 4. 隔离级别 Isolation level

可序列化要求比较严格

	Dirty Read	Unrepeatable Read	Phantom (幻读)
Serializable	No	No	No
Repeatable reads	No	No	Maybe
Read committed	No	Maybe	Maybe
Read Uncommitted	Maybe	Maybe	Maybe

Obtain all locks first; plus index locks, plus strong strict 2PL

Same as above, but S locks are released immediately

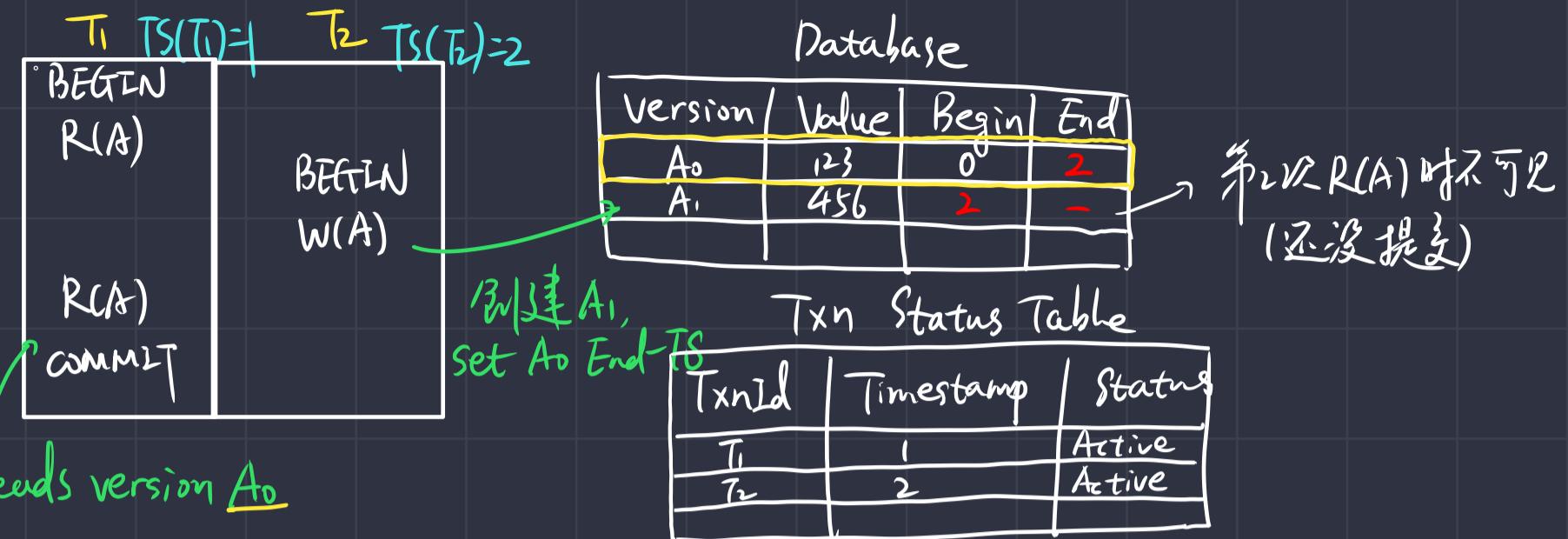
no S locks

## L18. MVCC Multi-Version Concurrency Control (广泛使用)

1. Writers do not block readers

Reader do not block writers

事务修改某数据后，DBMS 将为其创建一个新的版本



2. MVCC 并不只是一个并发控制协议，并发控制协议只是他的一部分

MVCC Design {  
 并发控制 : T10, OCC, 2PL  
 Version Storage 版本链  
 Garbage Collection  
 Index Management  
 Deletes

3. 版本存储: ① APPEND-ONLY 物理版本存在相同表空间中

Approach #1: Oldest to Newest

Approach #2: Newest to Oldest

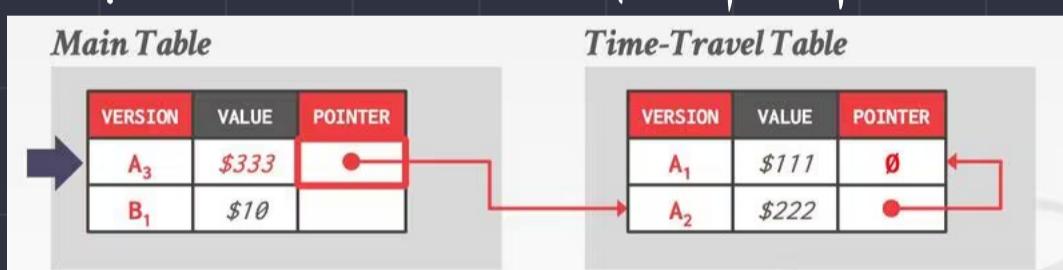
② Time-travel

将当前版本插入到 time-travel table，并更新 pointer.

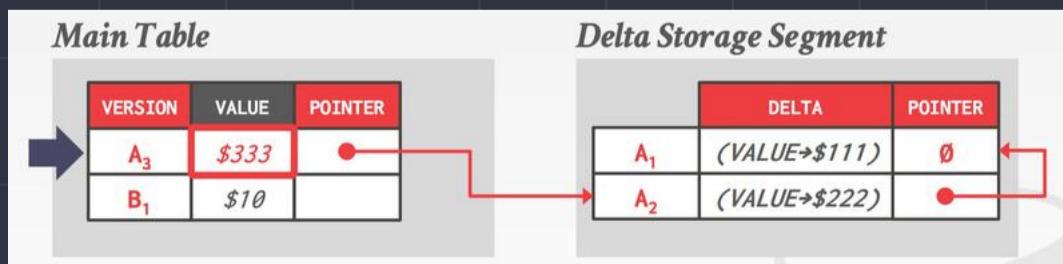
将新版本写入 Main Table 并更新指针

Tuple pointer		
A <sub>0</sub>	\$111	•
A <sub>1</sub>	\$222	•
B <sub>1</sub>	\$10	∅
A <sub>2</sub>	\$333	∅

单链表  
 (可跳转 page)



③ Delta Storage 存储变化信息



4. 回收收集 } 没有 active txn 看到的版本  
 } 被 Abort by txn 创建的版本

## Approach #1 : Tuple-level

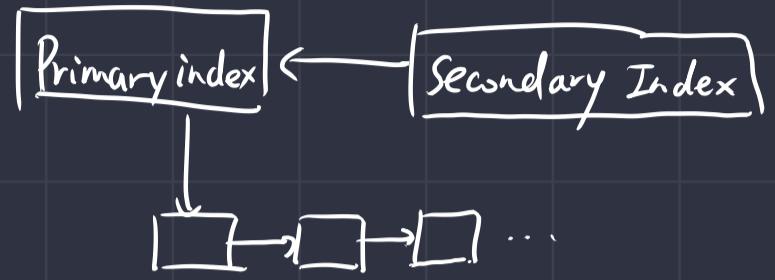
Vacuum 守护线程会周期性地检查每条数据不同版本  
结束时间 < 当前活跃 txn 最小时间戳  $\Rightarrow$  删除

## Approach #2: Transaction-level: 由 DBMS 处理

### 5. Index Management

主键索引 直接指向版本链 头节点

二级索引 { 存储主键值 / Tuple Id



### L9. Database logging 日志

1. Failure 分类

易损故障	{ 逻辑错误 e.g. 约束
	内部错误 e.g. 死锁
系统故障	{ 软件故障
	硬件故障 e.g. 断电
存储介质故障	

### 2. Buffer Pool Policies

Undo: 将中止或未完成的事务中 已经执行的操作回退

Redo: 将 提交的易损执行的操作重做

Steal Policy: DBMS 是否允许一个未提交事务修改持久化存储中的内容

Force Policy: DBMS 是否强制要求一个提交完毕事务的数据改动都反映在持久化存储中

### 3. Write ahead log Steal + No-force (WAL)

DBMS 必须先将操作 日志化到独立的日志文件，然后才能修改其真正的数据页

log entry:  $\rightarrow$  Transaction Id

$\rightarrow$  Object Id

$\rightarrow$  Before Value (UNDO)

$\rightarrow$  After Value (Redo)

当使用 append-only MVCC

时不需要

#### 4. Logging schemes

{ Physical e.g. g.t diff Before & After  
Logical < Ti Query = "UPDATE foo SET val=XYZ WHERE id=1" >  
Physiological