Relational Database System Implementation

CS122 – Lecture 2

Winter Term, 2018-2019

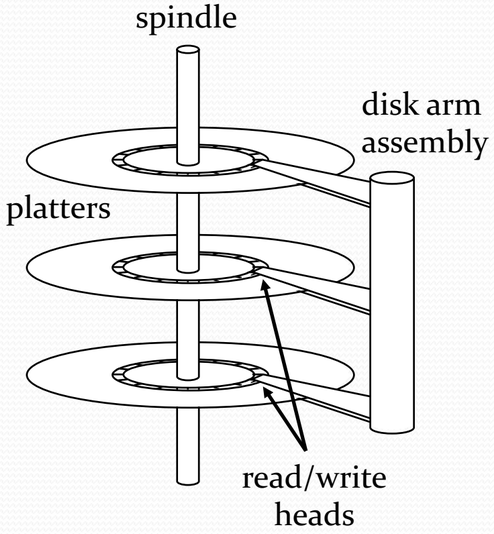
Relational Database System Implementation

CS122 – Lecture 2

2018–2019冬季学期

Magnetic Disks

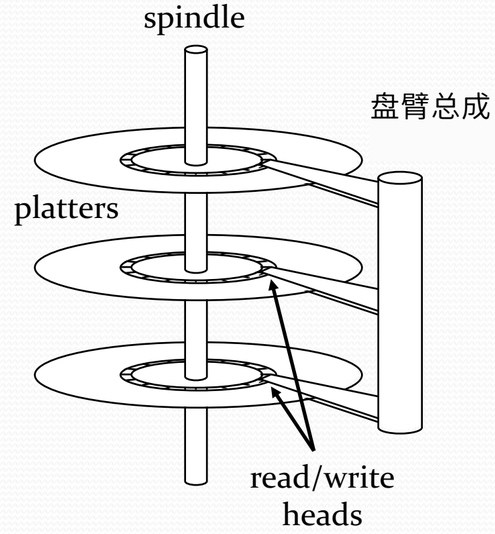
- Magnetic disks are most widely used online storage medium in computers
  - Hard disk drives (HDD)
- Drive contains some number of *platters* mounted on a *spindle*
  - Platters spin at a constant rate of speed
  - 5400 RPM, up to 15000 RPM
- Read/write heads are suspended above platters on a *disk arm*
  - All heads move together as a unit



2

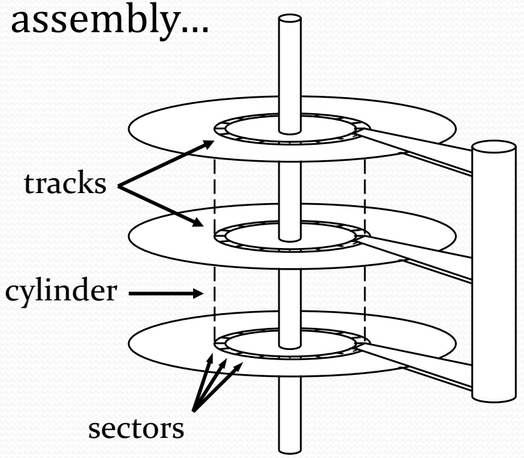
磁盘

磁盘是使用最广泛的在线存储
  计算机中的介质
    硬盘驱动器(HDD)
驱动器包含一些
  安装在主轴上的盘片
    盘片以恒定的速度旋转
      速度率
    5400RPM，最高15000RPM
读写头暂停
  磁盘臂上的盘片上方
    所有头部作为一个整体一起移动



Magnetic Disks (2)

- Platters are divided into *tracks*
- Tracks are divided into *sectors*
  - Modern drives have more sectors towards edge of disk
- All heads are positioned by one assembly...
  - A *cylinder* is made up of the tracks on all platters at the same position
- To read a sector from disk:
  - Assembly *seeks* to the appropriate cylinder
  - Sector is read when it rotates under the disk head



磁盘(2)

盘片被划分为磁道磁道被划分为扇区

现代驱动器在磁盘边缘有更多扇区

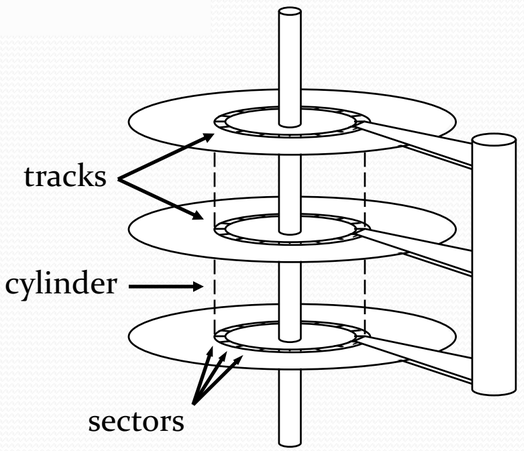
所有头都由一个组件定位……

气缸由
同一位置的所有盘片上的轨道

从磁盘读取扇区：

大会寻求
合适的气缸

扇区在旋转时被读取
磁头下



Disk Performance Measures

- *Access time* is the time between a read/write request being issued, and the data being returned
  - Read/write heads must be moved to appropriate track
  - Sectors must rotate past the read/write heads
- First operation is called a *seek*
  - Average seek time of a disk is measured from a series of random seeks (uniform distribution)
  - Generally ranges from 3-15ms
  - Typical consumer drives are in the range of 9-12ms
- Seeking nearby tracks will obviously be faster
  - Track-to-track seek times in range of 0.2-0.8ms
- (SSDs have “seek times” in the 0.08-0.16ms range)

4

磁盘性能测量

访问时间是读写请求之间的时间
发出，并返回数据

读写头必须移动到适当的轨道扇区必须旋转经过读写头

第一个操作称为查找

磁盘的平均寻道时间是从一系列
随机搜索（均匀分布）

一般范围为3–15ms典型的消费类驱动器在9–12ms
范围内

寻找附近的曲目显然会更快

磁道到磁道寻道时间在0.2–0.8ms范围内

（SSD的“寻道时间”在0.08–0.16ms范围内）

Disk Performance Measures (2)

- *Rotational latency time* is amount of time for sector to pass under read/write heads
  - Average rotational latency is ½ the time for a full rotation
  - 5,400 RPM: 5.6ms
  - 7,200 RPM: 4.2ms
  - 15,000 RPM: 2ms
- Disks can only read/write information so quickly
  - *Data transfer rate* specifies how fast data is read from/written to the disk
  - Current interfaces can support up to 600+ MB/sec
  - Actual transfer rate depends on several things:
    - The disk and its controller, motherboard chipset, etc.
    - The section of the disk being accessed

5

磁盘性能测量(2)

旋转延迟时间是扇区从
通过读写头

平均旋转延迟是完整旋转时间的 5 400RPM：5.6ms7 20
0RPM：4.2ms15 000RPM：2ms

磁盘只能如此快速地读取写入信息

数据传输率指定从写入到读取数据的速度
到磁盘

当前接口可以支持高达600+MB秒实际传输速率取
决于几个因素：

磁盘及其控制器、主板芯片组等。
正在访问的磁盘部分

Disk-Access Optimizations

- Wide range of techniques used to improve hard disk performance
  - Implemented in the HDD itself, and/or in operating system
- Buffering
  - When data is read, store it in a memory buffer
  - If same data is requested again, provide it from the buffer
- Read-ahead
  - When a sector is read, read other sectors in the same track
  - If a program is scanning through a file, subsequent accesses can be satisfied immediately from cache

6

Disk-Access Optimizations

用于改进硬盘的各种技术

performance

在HDD本身和/或操作系统中实现

- Buffering

读取数据时，将其存储在内存缓冲区中

如果再次请求相同的数据，则从缓冲区提供

- Read-ahead

读取一个扇区时，读取同一磁道中的其他扇区

如果程序正在扫描文件，后续访问

可以立即从缓存中满足

Disk-Access Optimizations (2)

- I/O Scheduling
  - The hard disk can queue up batches of read and write requests, then schedule them in a reasonable way
  - Goal: reduce the average seek time of accesses
  - Writes can be buffered in volatile memory to facilitate this (can cause problems if power fails before write is performed)
- Nonvolatile write buffers
  - Disk provides NV-RAM to cache disk writes
  - Data is saved in NV-RAM before being saved to disk
  - Data isn't written to disk until the disk is idle, or the NV-RAM buffer is full
  - If power fails, contents of NV-RAM are still intact!

7

Disk-Access Optimizations (2)

- I/O Scheduling

硬盘可以排队批量读写
请求，然后以合理的方式安排它们

目标：减少访问的平均寻道时间

写操作可以缓冲在易失性存储器中，以促进这一点
（如果在执行写入之前断电可能会导致问题）

非易失性写缓冲器

磁盘提供NV-RAM来缓存磁盘写入

数据在保存到磁盘之前先保存在NV-RAM中

在磁盘空闲或NV-RAM之前，数据不会写入磁盘
缓冲区已满

如果断电，NV-RAM的内容仍然完好无损！

Disk-Access Optimizations (3)

- RAID (Redundant Array of Independent Disks)
  - Employed for both performance and reliability reasons
  - One storage device can transfer up to 600MB/s
  - Processor memory bus can transfer GB/s
  - Idea: Access multiple storage devices in parallel
    - Data is either duplicated on multiple devices, or it is striped across multiple devices
    - A RAID controller translates logical accesses into corresponding accesses on the appropriate device(s)

8

Disk-Access Optimizations (3)

RAID（独立磁盘冗余阵列）

出于性能和可靠性的考虑而受聘

一个存储设备最多可传输600MBs

处理器内存总线可以传输GBs

理念：并行访问多个存储设备

数据在多个设备上重复，
或者跨多个设备条带化

RAID控制器将逻辑访问转换为
相应设备上的相应访问

Solid State Drives

- Solid State Drives are becoming increasingly common
  - Still more expensive and smaller than HDDs
  - (This trend will likely continue for a number of years)
- Use flash memory chips to provide persistent storage
  - Most common is NAND flash memory, which is read/written in 512B-4KB pages (similar to HDDs)
- Reads are very fast: on the order of a few µs
  - No seek time or rotational latency whatsoever!
  - (Still slower than main memory, of course)
- Write performance can be much more varied…

9

固态硬盘

固态硬盘正变得越来越普遍

仍然比HDD更昂贵且体积更小

（这种趋势可能会持续数年）

使用闪存芯片提供持久存储

最常见的是NAND闪存，即

读取写入512B–4KB页（类似于HDD）

读取速度非常快：大约几微秒

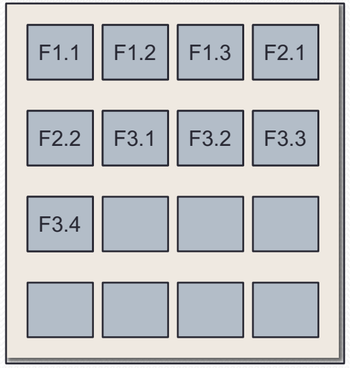
没有任何寻道时间或旋转延迟！

（当然还是比主存慢）

写入性能可以更加多样化……

Solid State Drives (2)

- SSDs are comprised of flash memory blocks
  - Each block can hold e.g. 4KB of data
  - As usual, break data files into blocks
- Example: three files on our SSD: F1, F2 and F3
- SSDs must follow specific rules when writing to blocks:
  - SSDs can only write data to blocks that are currently empty
  - Cannot modify a block that already contains data



10

固态硬盘(2)

SSD由闪存块组成

每个块可以容纳例如4KB数据

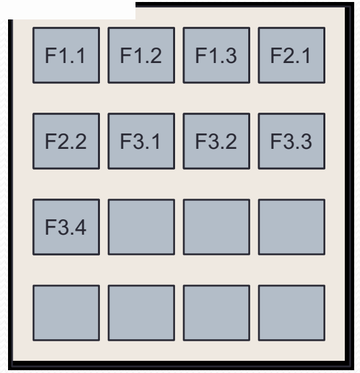
像往常一样，将数据文件分成块

示例：我们SSD上的三个文件：F1、F2和F3SSD必须遵循特定规则

写入块：

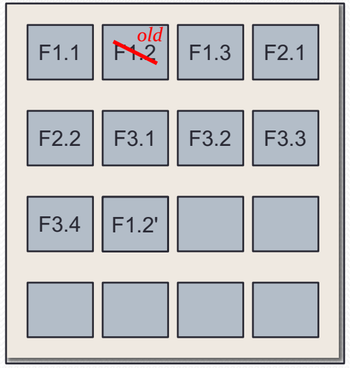
SSD只能将数据写入
目前为空

不能修改已经存在的块
包含数据



Solid State Drives (3)

- SSDs can only write to blocks that are currently empty
- Example: we want to modify the data in block 2 of F1
  - Can't just change the data in-place!
- Instead, must write a new version of F1.2
  - SSD marks old version of F1.2 as not in use, and stores a new version F1.2'



- **SSD Issue 1:**
  - SSDs aren't good at disk structures that require frequent in-place modifications

11

固态硬盘(3)

SSD只能写入当前为空的块示例：我们要修改F1的第2块中的数据

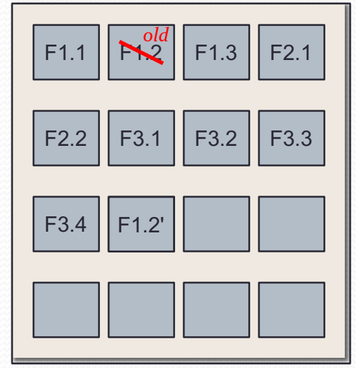
不能就地更改数据!

相反，必须写一个新版本的F1.2

SSD将旧版本的F1.2标记为非正在使用中，并存储新版本F1.2'

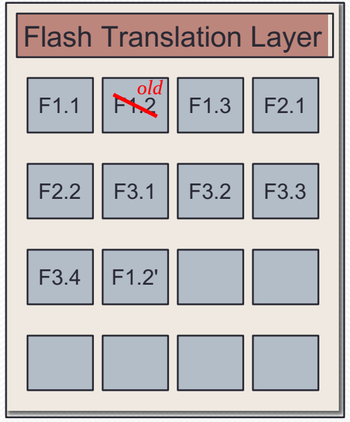
- **SSD Issue 1:**

SSD不擅长以下磁盘结构：需要频繁的就地修改



Solid State Drives (4)

- Don't want applications to have to keep track of the actual blocks that comprise their files…
  - Every time part of an existing file is written to the SSD, a new block must be used
- Solid State Drives also include a Flash Translation Layer that maps logical block addresses to physical blocks
  - This mapping is updated every time a write is performed



固态硬盘(4)

不希望应用程序必须跟踪
组成文件的实际块……

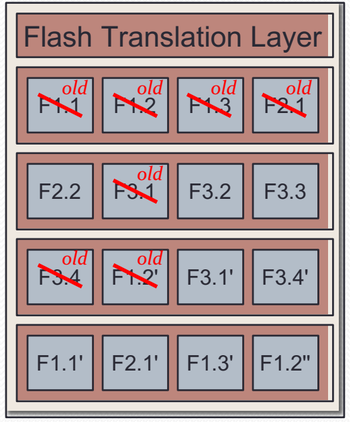
每次将现有文件的一部分写入SSD，
必须使用新块

固态驱动器还包括闪存
将逻辑块地址映射到物理块的转换层

此映射每次更新一次
执行写入

SSDs: Erase Blocks

- Over time, SSD ends up with few or no available cells
  - e.g. a series of writes to our SSD that results in all cells being used, or marked old
- Problem: SSDs can only erase cells in groups
  - Groups are called *erase blocks*
  - A read/write block might be 4-8KiB…
  - Erase blocks are often 128 or 256 of these blocks (e.g. 2MiB)!
- SSDs must periodically clear one or more erase-blocks to free up space
  - Erasing a block takes 1-2 ms to perform



13

SSD：擦除块

随着时间的推移，SSD最终会出现很少或没有可用单元

例如对我们的SSD进行一系列写入，导致所有单元正在使用或标记为旧的

问题：SSD只能分组擦除单元格

组被称为擦除块

一个读写块可能是4-8KiB…

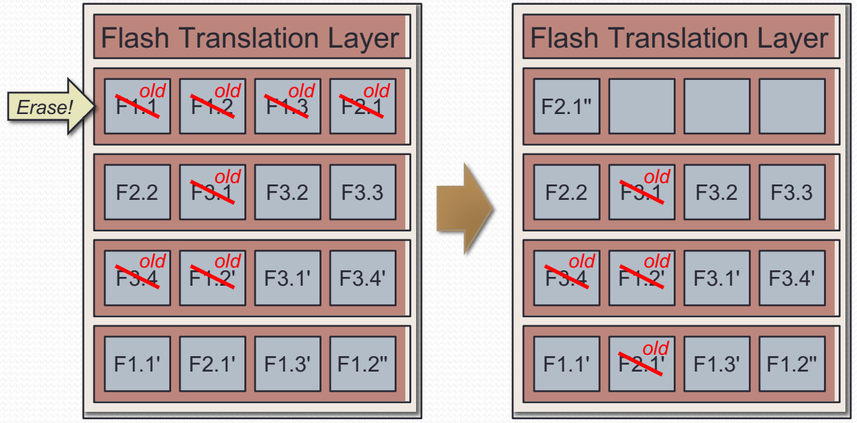
擦除块通常是128或256这些块（例如2MiB）！

SSD必须定期清除一个或更多擦除块以释放空间

擦除一个块需要1-2毫秒来执行

SSDs: Erase Blocks (2)

- Best case is when a whole erase block can be reclaimed
- Example: want to write to F2.1'
  - SSD can clear an entire erase-block and then write the new block

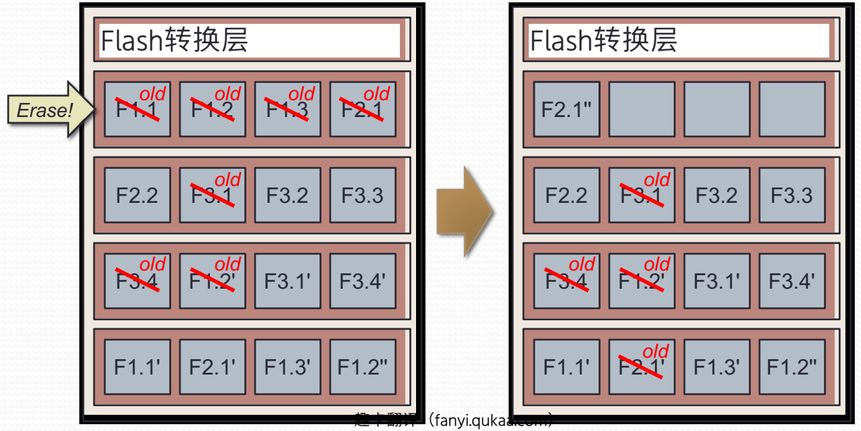


14

SSD：擦除块(2)

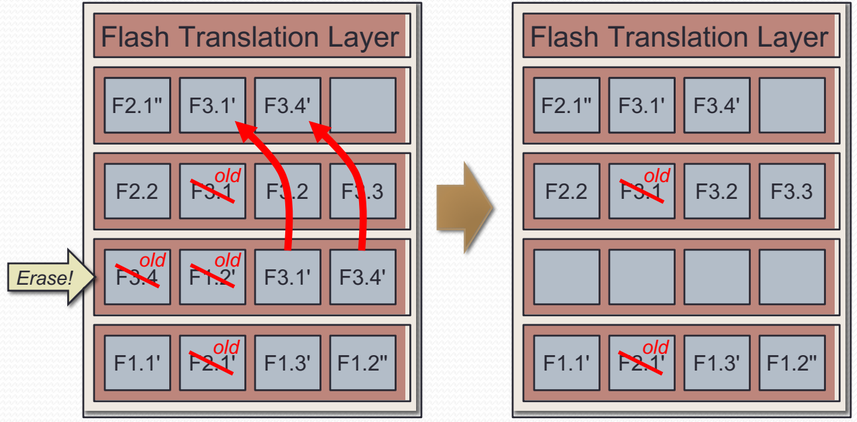
最好的情况是可以回收整个擦除块示例：想要写入F2.1'

SSD可以清除整个擦除块，然后写入新区块



SSDs: Erase Blocks (3)

- More complicated when an erase block still holds data
  - e.g. SSD decides it must reclaim the third erase-block
- SSD must relocate the current contents before erasing
- Example: SSD wants to clear third erase-block



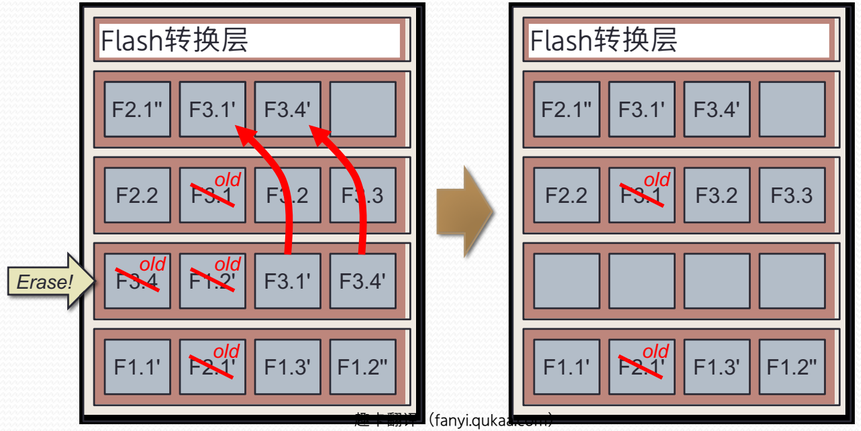
15

SSD：擦除块(3)

当擦除块仍然保存数据时更复杂

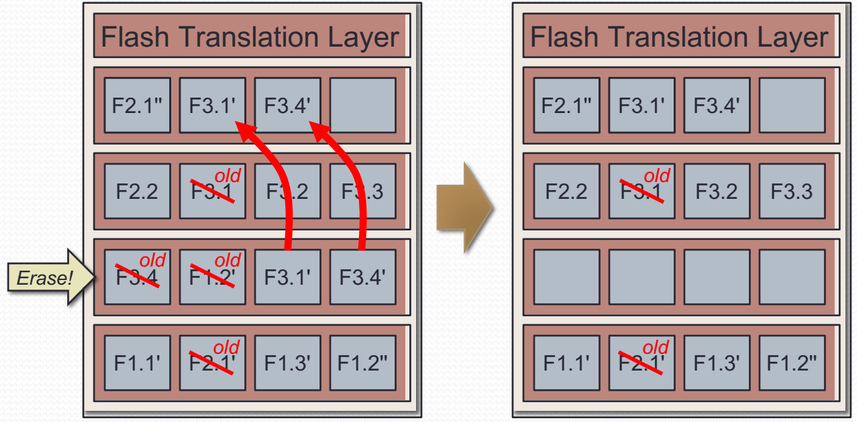
例如SSD决定它必须回收第三个擦除块

SSD必须在擦除之前重新定位当前内容示例：SSD想要清除第三个擦除块



SSDs: Erase Blocks (4)

- **SSD Issue 2**:
  - Sometimes a write *to* the SSD incurs additional writes *within* the SSD
  - Phenomenon is called *write amplification*



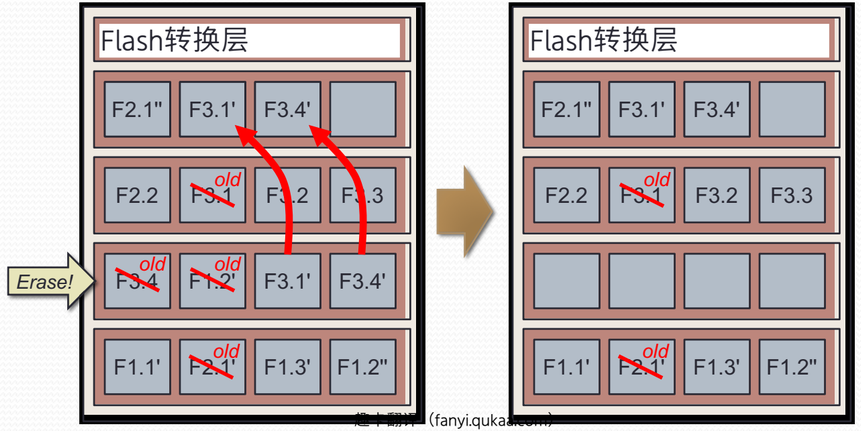
SSD：擦除块(4)

- **SSD Issue 2**:

有时对SSD的写入会导致额外的写入

SSD内

现象称为写放大



SSDs: Erasure and Wear

- A block can only be erased a fixed number of times…
- SSDs ensure that different blocks wear evenly
  - Called *wear leveling*
  - Data that hasn't changed much (*cold data*) is moved into blocks with higher erase-counts
  - Data that has changed often (*hot data*) is moved into blocks with lower erase-counts
- Theoretically, SSDs should last longer than hard disks

17

SSD：擦除和磨损

一个块只能被擦除固定的次数……

SSD确保不同块均匀磨损

称为磨损均衡

变化不大的数据（冷数据）被移入
具有较高擦除计数的块

经常变化的数据（热点数据）被移入
具有较低擦除计数的块

从理论上讲，SSD的使用寿命应该比硬盘长

SSDs and HDDs: Failure Modes

- SSDs fail in different ways than hard disks generally do
- Hard disks tend to degrade more slowly over time
  - Sensitive to mechanical shock and vibration
  - Surface defects can slowly become apparent over time
  - Result: usually, data is slowly lost over time (although disk controllers can also burn out, etc.)
- Solid state drives are far less sensitive to mechanical shock and other environmental factors
  - But, SSD controller electronics can fail, particularly due to power surges / outages
  - Result: all the data disappears at once, without warning

18

SSD和HDD：故障模式

SSD的故障方式与一般硬盘不同随着时间的推移，硬盘的降级趋于缓慢

- 对机械冲击和振动敏感
- 随着时间的推移，表面缺陷会慢慢变得明显
- 结果：通常，数据会随着时间慢慢丢失（尽管磁盘控制器也可能烧坏等）

固态硬盘对机械设备的敏感性要低得多冲击和其他环境因素

- 但是，SSD控制器电子设备可能会出现故障，尤其是由于电涌停电
- 结果：所有数据立即消失，没有警告

Database External Storage

- Virtually all of our discussion going forward will assume spinning magnetic disks, not solid state drives
  - Data volumes continue to grow, and HDDs are both larger and cheaper than SSDs
  - HDDs will continue to be relevant for the time being
- Observation 1: Solid-state drives obviate some of the issues we will take into account!
  - e.g. designing algorithms and file-storage formats to minimize disk seek overhead
  - There is no seek overhead with SSDs

19

数据库外部存储

我们接下来的几乎所有讨论都将
假设旋转磁盘，而不是固态驱动器

数据量持续增长，HDD
比SSD更大更便宜

硬盘驱动器暂时将继续具有相关性

观察1：固态驱动器消除了一些
我们会考虑的问题！

例如设计算法和文件存储格式
最小化磁盘寻道开销

SSD没有寻道开销

Database External Storage (2)

- Most of our discussions assume that there is no overhead for in-place modification of data
- Observation 2: Solid-state drives really aren't capable of modifying data in-place
  - They can present the abstraction, but under the hood, the SSD is doing something completely different
  - SSDs are more efficient with file formats that minimize in-place modification of data
- This is an active area of research

20

数据库外部存储(2)

我们的大多数讨论都假设没有
就地修改数据的开销

观察2：固态硬盘真的不行
就地修改数据

他们可以呈现抽象，但在幕后，
SSD正在做一些完全不同的事情

SSD的效率更高，文件格式最小化
就地修改数据

这是一个活跃的研究领域

Database Files

- Databases normally store data in files…
  - The filesystem is provided by the operating system
- Operating system provides several essential facilities:
  - Open a file at a particular filesystem path
  - Seek to a particular location in a file
  - Read/write a block of data in a file
  - (other facilities as well, e.g. memory-mapping a file into a process' address-space)

21

数据库文件

数据库通常将数据存储在文件中……

文件系统由操作系统提供

操作系统提供了几个重要的功能：

在特定文件系统路径打开文件

查找文件中的特定位置

读写文件中的数据块

（还有其他功能，例如将文件内存映射到process' address-space)

Database Files (2)

- Operating systems also provide the ability to *synchronize* a file to disk
  - Ensures that all modified data caches are flushed to disk
  - Includes flushing of OS buffers, hard-disk cache, etc.
  - Expectation is that if the operation completes, the data is now persistent (e.g. on the disk platter, or in NV-RAM)
- If the system crashes before a modified file is sync'd to disk, data will very likely be corrupted and/or lost
- Once the file is sync'd, the OS effectively guarantees that the disk state reflects the latest version of the file

22

数据库文件(2)

操作系统还提供了以下能力：
- 将文件同步到磁盘
  - 确保所有修改的数据缓存都刷新到磁盘
  - 包括刷新操作系统缓冲区、硬盘缓存等。
  - 期望是如果操作完成，数据是现在持久化（例如在磁盘盘片或NV-RAM中）

如果在修改文件同步到之前系统崩溃
- 磁盘，数据很可能会损坏或丢失

文件同步后，操作系统有效保证
- 磁盘状态反映文件的最新版本

Disk Files and Blocks

- Databases normally read and write disk files in blocks
  - Block-size is usually a power of 2, between 2^9 and 2^{16}
- Main reason is performance:
  - Disk access latency is large, but throughput is also large
  - Accessing 4KiB is just as expensive as accessing one byte
- Also makes it easier for Storage Manager to manage buffering, transactions, etc.
  - Disk pages are a convenient unit of data to work with
- The OS presents files as a contiguous array of bytes…
  - Typically want the database block size to be some multiple of the storage device block size

23

磁盘文件和块

数据库通常以块为单位读写磁盘文件
块大小通常是2的幂，介于29和216之间

主要原因是性能：
磁盘访问延迟大，但吞吐量也大
访问4KiB与访问一个字节一样昂贵

还使StorageManager更易于管理
缓冲、交易等
磁盘页是一种方便使用的数据单元

操作系统将文件显示为连续的字节数组……
通常希望数据库块大小是
存储设备块大小

Disk Files and Blocks (2)

- Blocks in a file are numbered starting at 0
- To read or write a block in a data file:
  - Seek to the location *block_num* × *page_size*
  - Read or write *page_size* bytes
- To create a new block:
  - Most platforms will automatically extend a file's size when a write occurs past the end of the file
  - Seek to location of new block, then write new block's data
- To remove blocks from the end of the file:
  - Set the file's size to the desired size
  - File will be truncated (or extended) to the specified size

24

磁盘文件和块(2)

文件中的块从0开始编号读取或写入数据文件中的块：

寻找位置block_num×page_size

读取或写入page_size字节

创建一个新块：

大多数平台会自动扩展文件的大小
写入发生在文件末尾之后

寻找新区块的位置，然后写入新区块的数据

从文件末尾删除块：

将文件大小设置为所需大小

文件将被截断（或扩展）到指定的大小

Files and Blocks… and Tuples?

- Issue:
  - Physical data file will be accessed in units of blocks
  - Query engine accesses data as sequences of records, often specifying predicates that the records must satisfy

- How do we organize blocks within data files?
- How do we organize records within blocks?
- Do we want to apply any file-level organization of records as well?

25

文件和块……还有元组?

- Issue:

  物理数据文件将以块为单位访问

  查询引擎以记录序列的形式访问数据，
  经常指定记录必须满足的谓词

我们如何在数据文件中组织块?

我们如何在块内组织记录？我们是否要应用任何文件级组织
也有记录?

Caveats

- Two important caveats to state up front:
- Caveat 1 (as before):
  - Most of our discussion going forward will assume spinning magnetic disks, not solid state drives
  - Data is frequently changed in-place
- Caveat 2:
  - We are discussing general implementation approaches, not theory, so there are many “right” ways to do things
  - Most implementations use these approaches, and/or minor variations on them

26

Caveats

预先声明的两个重要警告：

- Caveat 1 (as before):

  我们接下来的大部分讨论将假设
  旋转磁盘，而不是固态驱动器

  数据经常就地更改

- Caveat 2:

  我们正在讨论一般的实施方法，
  不是理论，所以有很多“正确”的做事方式

  大多数实现都使用这些方法，并且或
  它们的细微变化

Data File Organization

- Simplification 1:
  - We will store each table's data in a separate file.

- Some databases allow records from related tables to be stored together in a single file
  - e.g. records that would equijoin together are stored adjacent to each other in the file
  - Called a *multitable clustering file organization*
  - Facilitates *very* fast joins between these tables

27

数据文件组织

- Simplification 1:
  我们会将每个表的数据存储在单独的文件中。

一些数据库允许将相关表中的记录
一起存储在一个文件中

例如将等值连接在一起的记录被存储
在文件中彼此相邻

称为多表集群文件组织

促进这些表之间的快速连接

Data File Organization (2)

- Simplification 2:
  - We will require that every tuple fits entirely within a single disk block.

- Disk blocks can usually hold multiple records, but it is easy for a tuple to exceed the size of a single block
  - e.g. table with **`VARCHAR(20000)`** field; pg. size of 4KiB
- Most DBs support records larger than a disk block
  - DB can support records that span multiple blocks, or it can use separate overflow storage for large records, etc.

数据文件组织(2)

- Simplification 2:

  我们将要求每个元组完全适合
  单个磁盘块。

磁盘块通常可以容纳多条记录，但它是
元组容易超过单个块的大小

例如带有VARCHAR(20000)字段的表；pg。4KiB大小

大多数数据库支持大于磁盘块的记录

DB可以支持跨越多个块的记录，或者它
可以对大的记录等使用单独的溢出存储。

Considerations

- Operations performed on table data:
  - Inserting new records
    - *(reuse available space before increasing file size?)*
  - Deleting records
    - *(coalesce freed space if possible?)*
  - Selecting/scanning records (possibly applying updates)
  - Operations may involve only a few records, or they may involve many records

- Want to optimally handle the expected usage
  - Evaluate storage format against all above operations!
  - Don't impose too much space overhead
  - Don't unnecessarily hinder speed of operation

29

Considerations

对表数据执行的操作：

插入新记录

（在增加文件大小之前重用可用空间？）

删除记录

（如果可能，合并释放的空间？）

选择扫描记录（可能应用更新）操作可能只涉及几条记录，或者它们可能
涉及许多记录

希望以最佳方式处理预期使用情况

针对上述所有操作评估存储格式！不要在开销上施加
太多空间不要不必要地影响操作速度

Example: Inserting Records

- User executes this SQL:

```
INSERT INTO users VALUES
  (103921, 'joebob', 'Joe Bob', 'https://…');
```

- Database must find a block with enough space to hold the new record
- NanoDB's solution:
  - Starting with first data block in table file, search linearly until a block is found with enough space to hold the record
  - If we reach the end of the file, extend the file with a new block and add the record there
- What is this approach good at? What is it bad at?

30

示例：插入记录

用户执行以下SQL：

```
插入用户价值
  (103921, 'joebob', 'Joe Bob', 'https://…');
```

数据库必须找到一个有足够空间的块来容纳新纪录

- NanoDB's solution:

从表文件中的第一个数据块开始，线性搜索直到找到一个有足够空间来保存记录的块

如果我们到达文件的末尾，用一个新块扩展文件并在那里添加记录

这种方法擅长什么？它不擅长什么？

Example: Inserting Records (2)

- NanoDB approach is very slow for inserting records!
  - One benefit: reuses free space as much as possible
- Could remember the last block in the file with free space, and start there when adding new rows
- Can also use block-level structures to manage the file
  - Often focused on making it much faster/easier to find available space in the file
- Can also impact database performance if the approach causes many extra disk seeks and/or block reads

31

示例：插入记录(2)

NanoDB方法插入记录非常慢！

一个好处：尽可能多地重复使用可用空间

可以用free记住文件中的最后一个块空间，并在添加新行时从那里开始

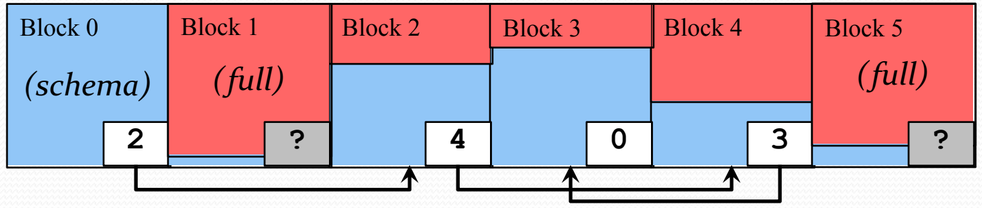
也可以使用块级结构来管理文件

通常专注于使其更快更容易找到文件中的可用空间

如果采用这种方法，也会影响数据库性能导致许多额外的磁盘寻道和/或阻塞读取

Block-Level Organization

- Introduce block-level structure to manage the file
- Example: list of blocks that can hold another tuple
  - First block in the data file specifies start of list
  - "Pointers" in the linked list are simply block numbers
    - e.g. could use a block number of 0 to terminate the list



- In NanoDB, block 0 is special:
  - It holds the table-file's schema, among other things

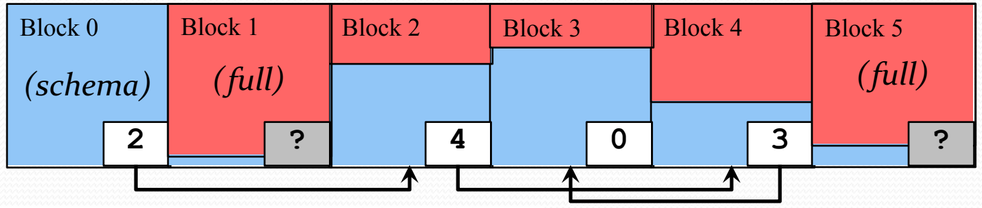
32

Block-Level Organization

引入块级结构来管理文件示例：可以容纳另一个元组的块列表

数据文件中的第一个块指定链表的开始链表中的"指针"只是块号

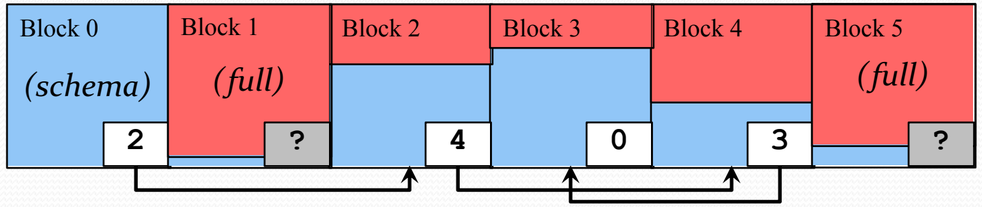
例如可以使用块号0来终止列表



在NanoDB中，块0是特殊的：

它包含表文件的模式，除其他外

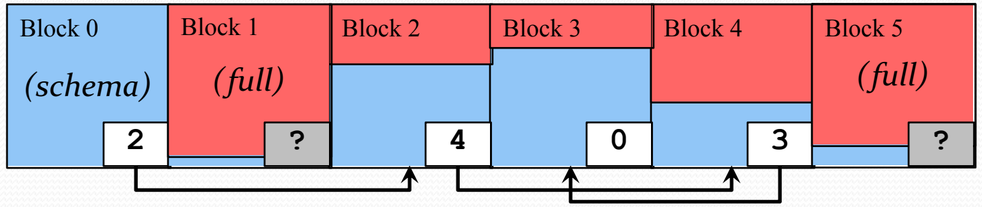
List of Non-Full Blocks (1)



- Note that pages will almost never be *completely* full
  - List simply specifies pages that can hold another tuple
- Can use the table's schema to compute minimum and maximum size of a tuple for that table

33

非完整区块列表(1)

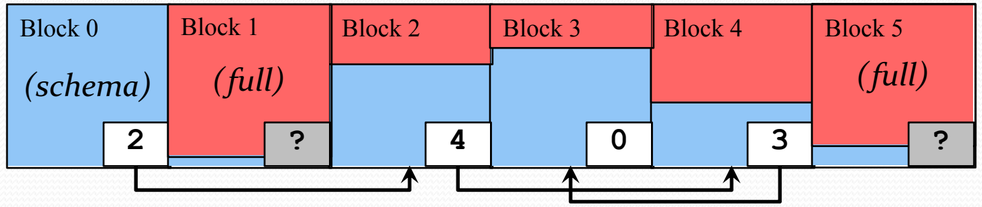


请注意，页面几乎永远不会完全填满

List只是指定可以容纳另一个元组的页面

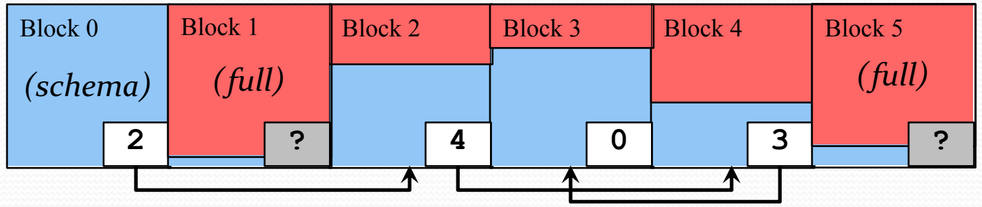
可以使用表的模式来计算最小值和
该表的元组的最大大小

List of Non-Full Blocks (2)



- When a new row is inserted:
  - Starting with first block, search through list of blocks with free space, for space to store the new tuple
  - When space is found, store the tuple
  - If the block is now full, remove it from the list
- Now we sometimes modify *two* pages instead of one

非完整区块列表(2)



插入新行时：

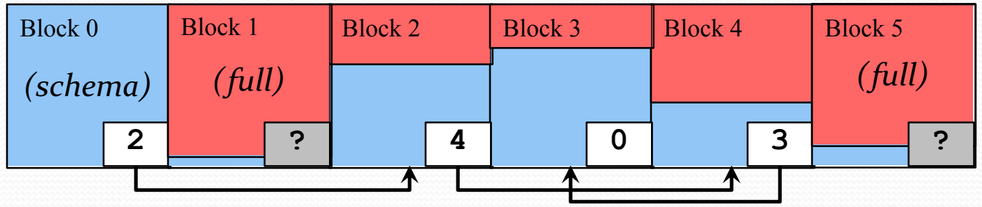
从第一个块开始，搜索块列表
有空闲空间，用于存储新元组的空间

找到空间后，存储元组

如果块现在已满，请将其从列表中删除

现在我们有时会修改两页而不是一页

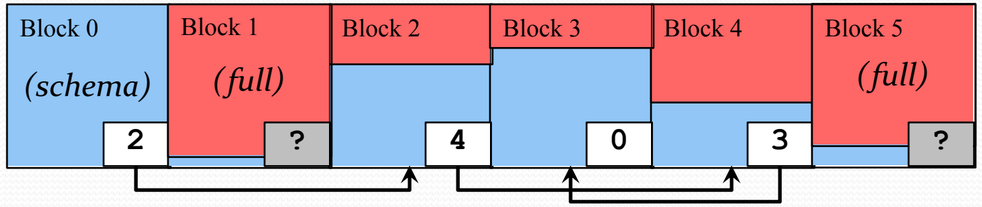
List of Non-Full Blocks (3)



- When a new row is inserted:
  - Starting with first block, search through list of non-full blocks for space to store the new tuple
- Other performance issues?
  - Scanning through the list of non-full blocks will likely incur many disk seeks
  - Could mitigate this by keeping free list in sorted order, but this would be more expensive to maintain

35

非完整区块列表(3)



插入新行时：

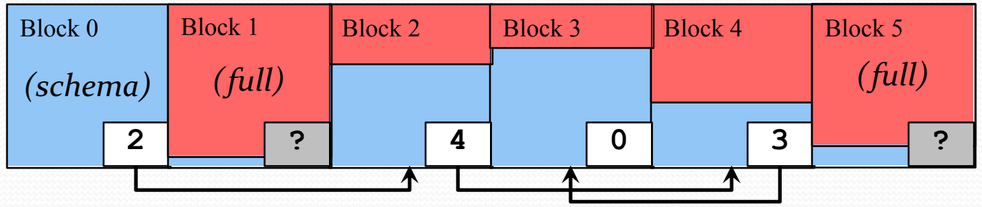
从第一个块开始，搜索非完整块列表
用于存储新元组的空间

其他性能问题?

扫描非完整块列表可能会导致
许多磁盘寻道

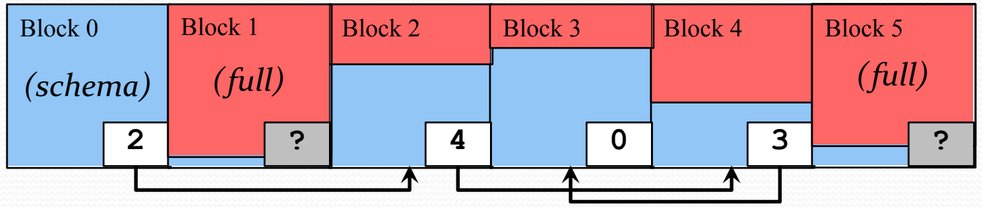
可以通过保持空闲列表排序来缓解这种情况，但是
这将更加昂贵

List of Non-Full Blocks (4)



- When a row is deleted:
  - If block was previously full, need to add it to the non-full list
    - e.g. if tuple was deleted from block 5
  - A simple solution: always add the block to start of the list
    - (Issue: Non-full list will become out of order)
  - Again, two blocks are written in some situations
  - (It's likely that block 0 will already be in cache, though)

非完整区块列表(4)



当一行被删除时：

如果块之前已满，则需要将其添加到未满列表中

例如如果元组从块5中删除

一个简单的解决方案：总是将块添加到列表的开头

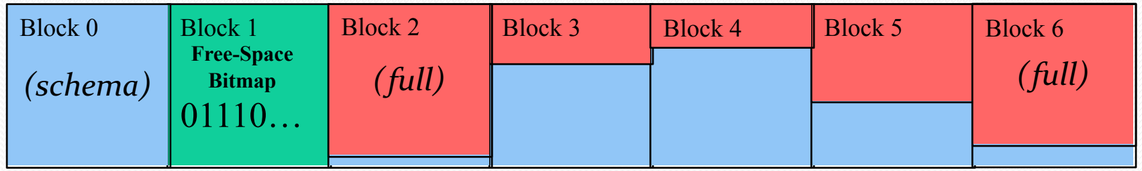
（问题：非完整列表将变得乱序）

同样，在某些情况下会写入两个块

（不过，块0很可能已经在缓存中）

Free-Space Bitmap

- Can also use a *free-space bitmap* to record blocks with available space
  - 0 = “block is full”
  - 1 = “block may have room for another tuple”

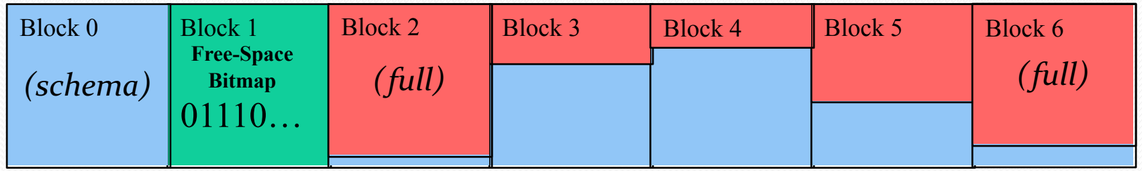


- Achieves same benefits as a list of non-full blocks, but with far fewer seeks, less space consumed, etc.
  - Requires same operations as non-full list, but they all operate on the free-space bitmap
  - Can become a performance bottleneck w/concurrent writes

37

Free-Space Bitmap

也可以使用自由空间位图来记录块
可用空间
0=“块已满”1=“块可能有空间容纳另一个元组”



实现与非完整块列表相同的好处，但具有
更少的搜索，更少的空间消耗等。
需要与非完整列表相同的操作，但它们都可以操作
在自由空间位图上
并发写入可能成为性能瓶颈