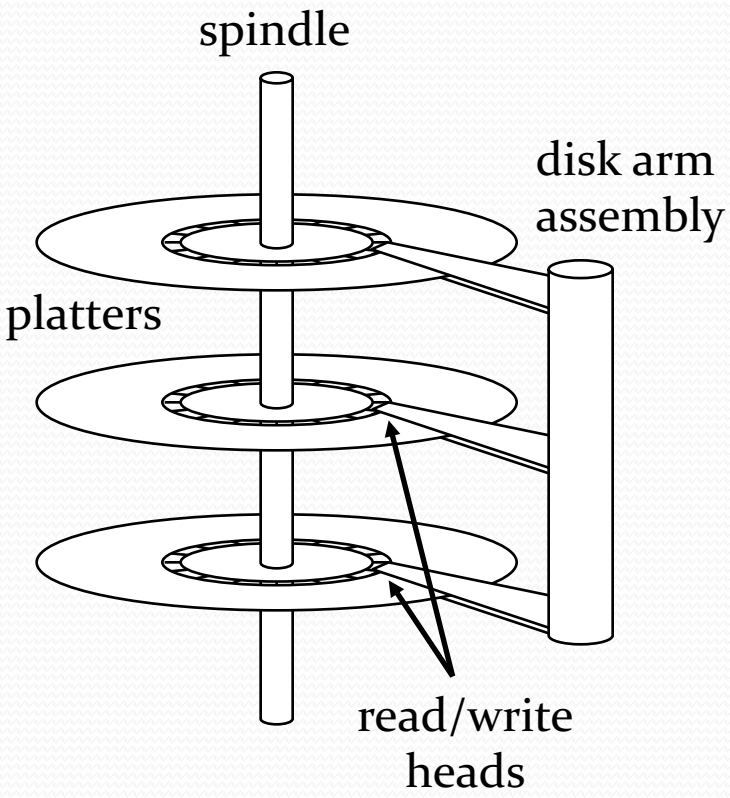
Relational Database System Implementation

CS122 – Lecture 2

Winter Term, 2018-2019

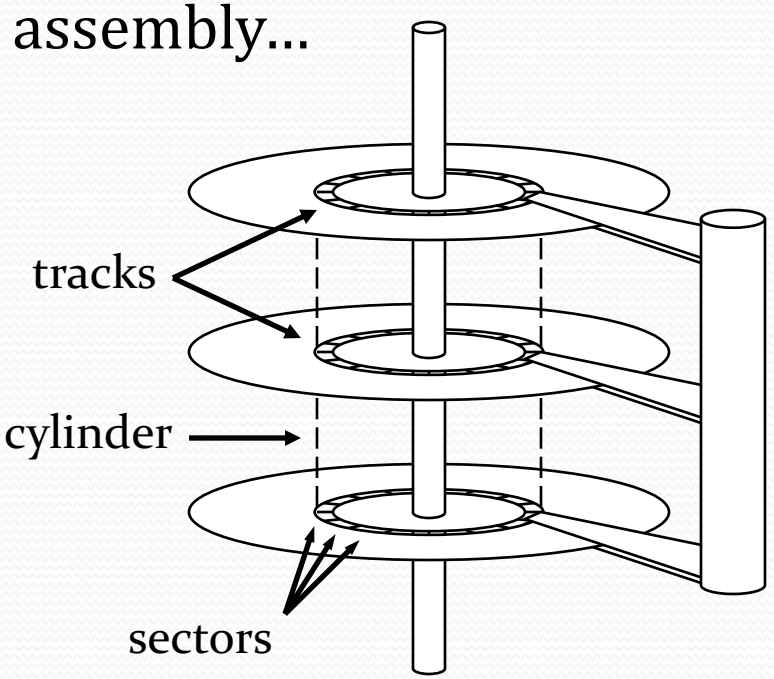
Magnetic Disks

- 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



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



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)

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

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

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!

Disk-Access Optimizations (3)

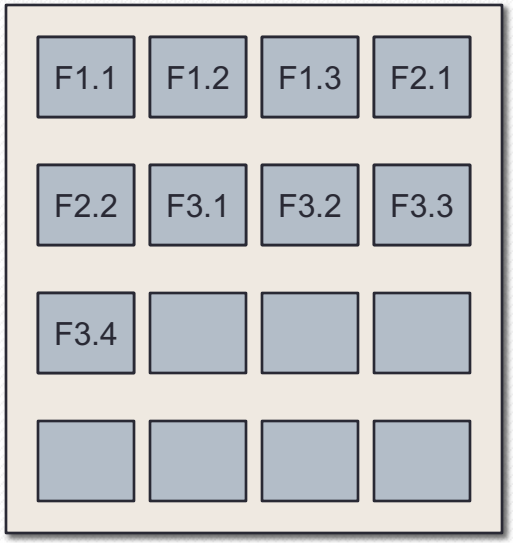
- RAID (Redundant Array of Independent Disks)
  - Employed for both performance and reliability reasons
  - One storage device can transfer up to 600MB/s
  - Processor memory bus can transfer GB/s
  - Idea: Access multiple storage devices in parallel
    - Data is either duplicated on multiple devices,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

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…

Solid State Drives (2)

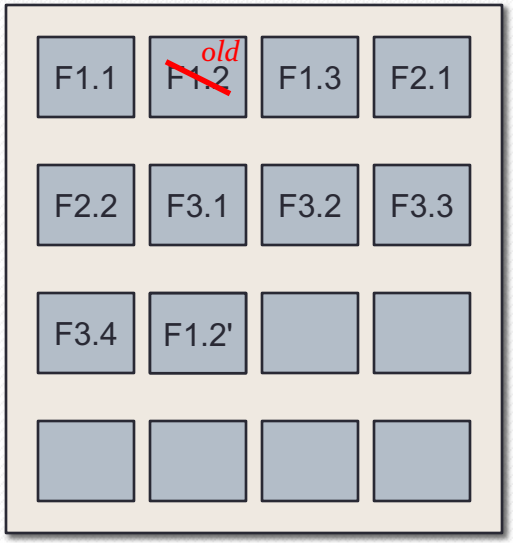
- SSDs are comprised of flash memory blocks
  - Each block can hold e.g. 4KB of data
  - As usual, break data files into blocks
- Example: three files on our SSD: F1, F2 and F3
- SSDs must follow specific rules when writing to blocks:
  - SSDs can only write data to blocks that are currently empty
 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



Solid State Drives (3)

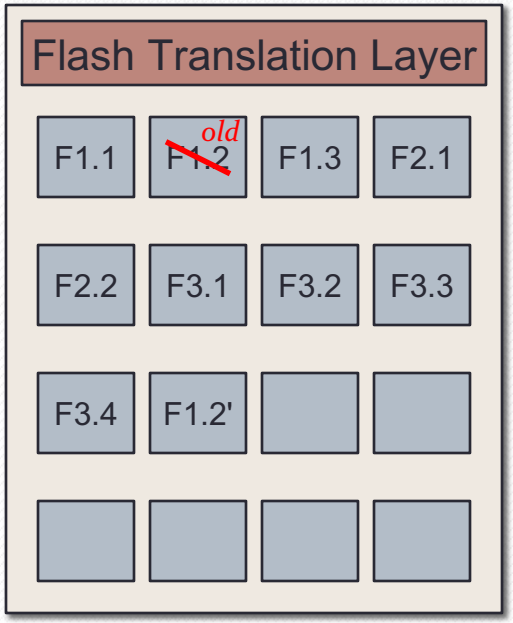
- SSDs can only write to blocks that are currently empty
- Example: we want to modify the data in block 2 of F1
  - Can't just change the data in-place!
- Instead, must write a new version of F1.2
  - SSD marks old version of F1.2 as not in use, and stores a new version F1.2'

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



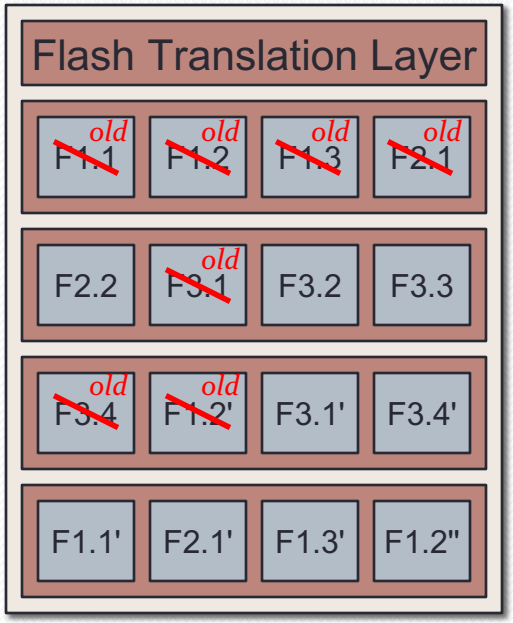
Solid State Drives (4)

- 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



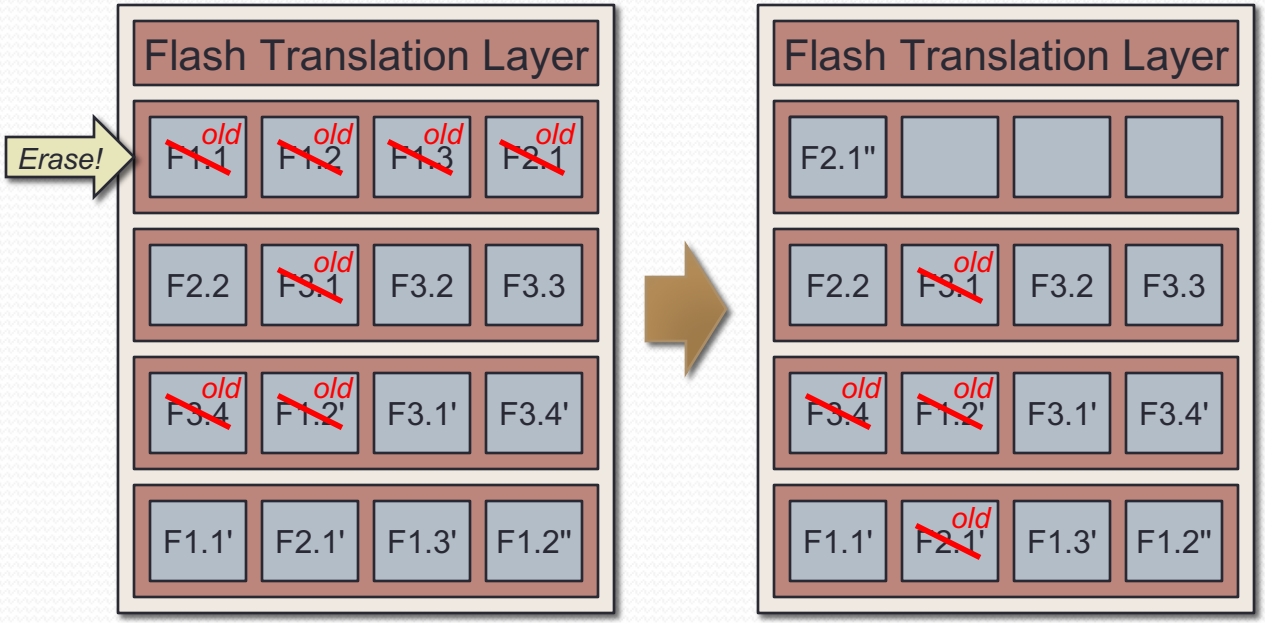
SSDs: Erase Blocks

- 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



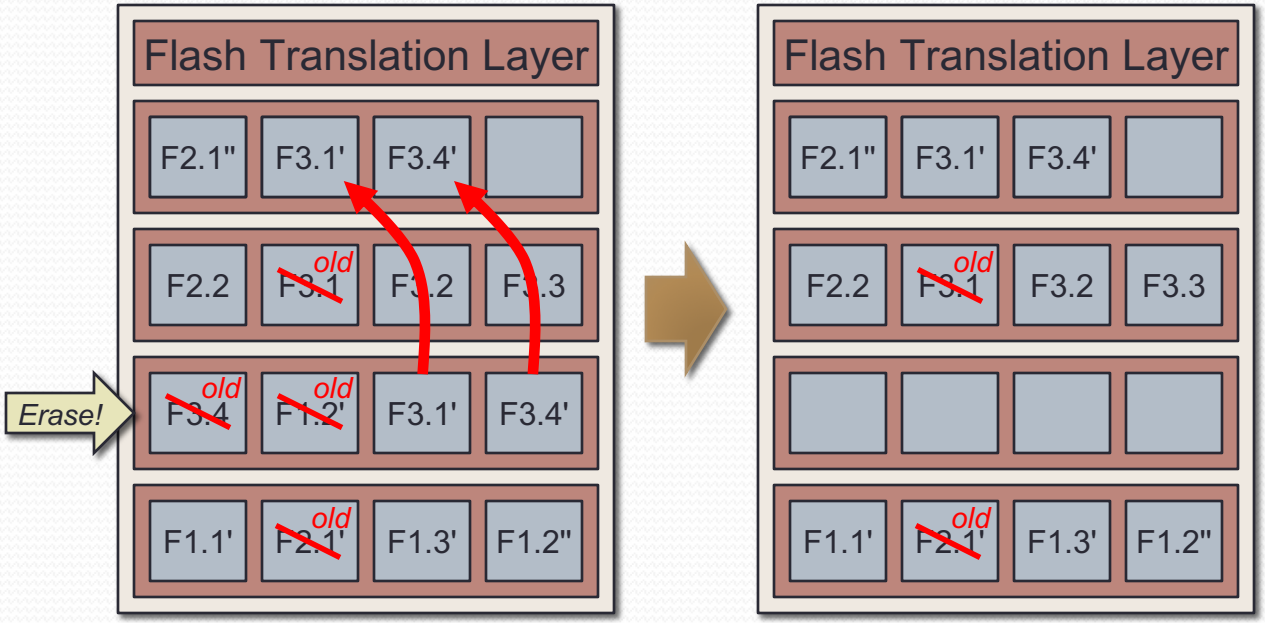
SSDs: Erase Blocks (2)

- 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



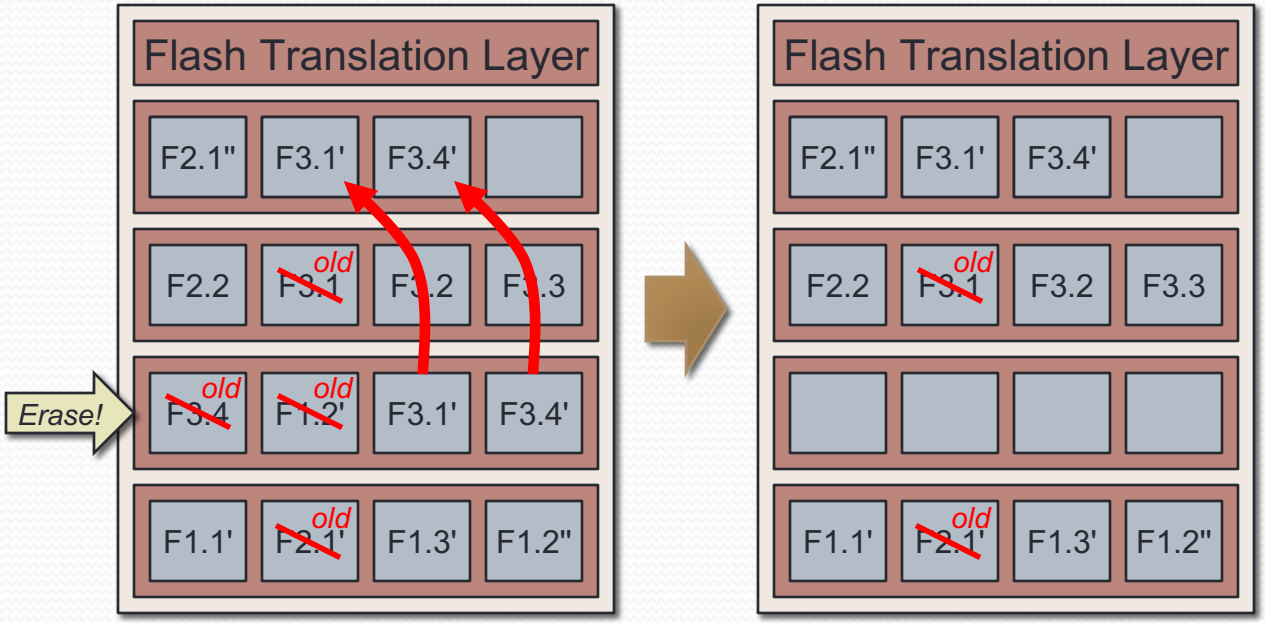
SSDs: Erase Blocks (3)

- 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



SSDs: Erase Blocks (4)

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



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

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

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

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

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)

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

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

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

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?

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

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

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.

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

Example: Inserting Records

- User executes this SQL:

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

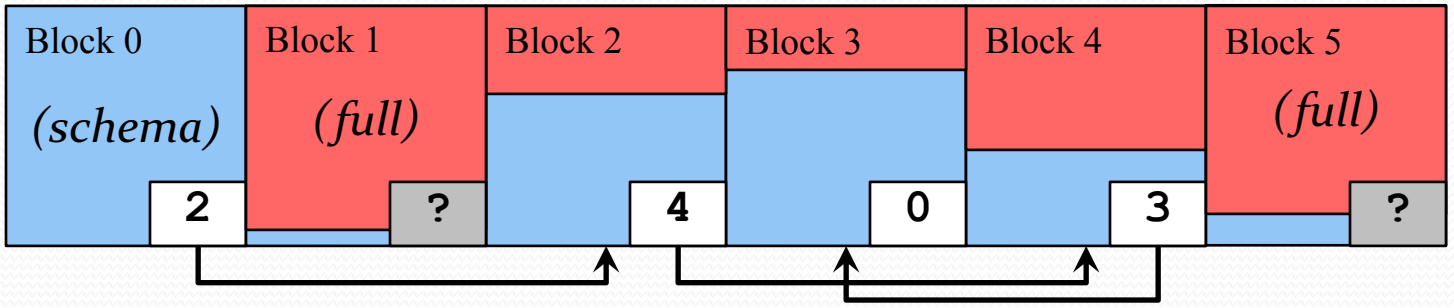
- Database must find a block with enough space to hold the new record
- NanoDB's solution:
  - Starting with first data block in table file, search linearly until a block is found with enough space to hold the record
  - If we reach the end of the file, extend the file with a new block and add the record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

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

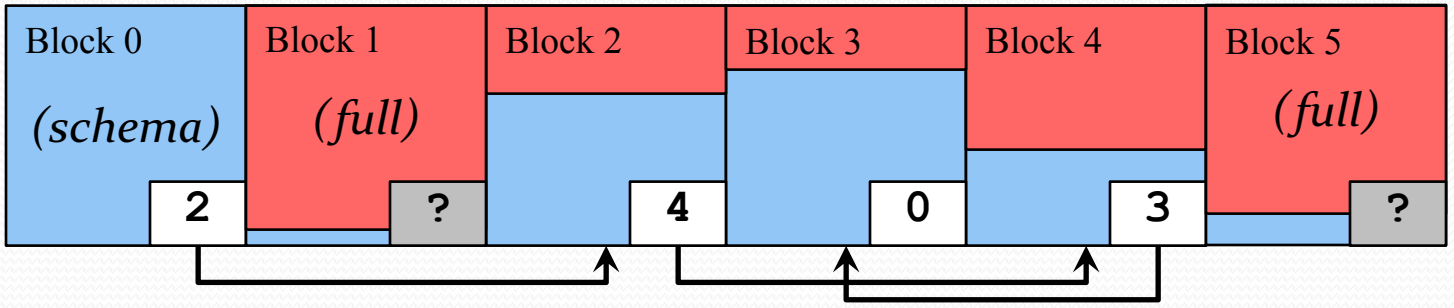
Block-Level Organization

- 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



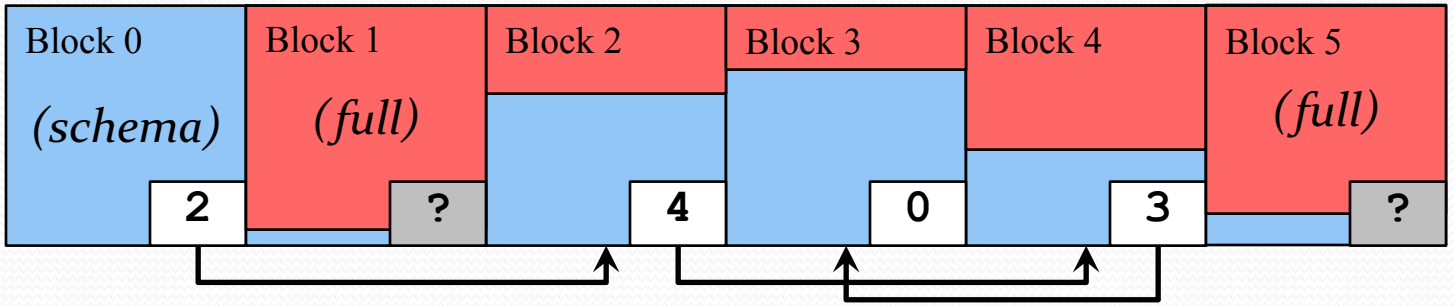
- In NanoDB, block 0 is special:
  - It holds the table-file's schema, among other things

List of Non-Full Blocks (1)



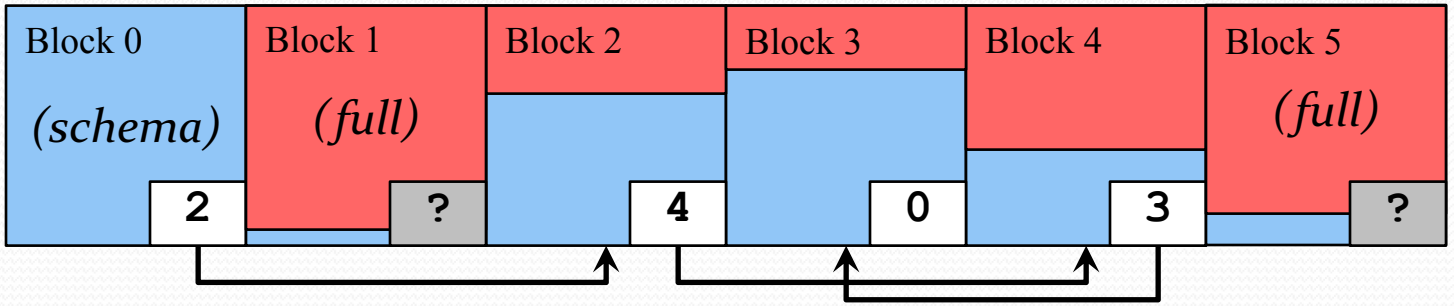
- Note that pages will almost never be *completely* full
  - List simply specifies pages that can hold another tuple
- Can use the table's schema to compute minimum and maximum size of a tuple for that table

List of Non-Full Blocks (2)



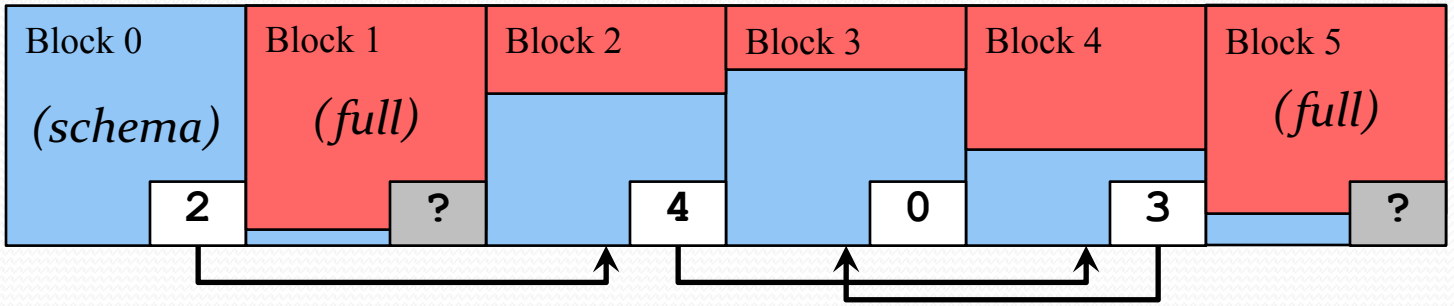
- When a new row is inserted:
  - Starting with first block, search through list of blocks with free space, for space to store the new tuple
  - When space is found, store the tuple
  - If the block is now full, remove it from the list
- Now we sometimes modify *two* pages instead of one

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

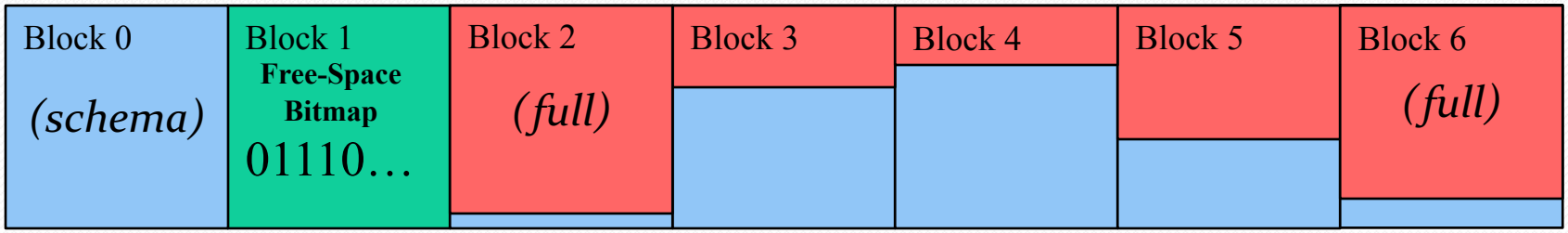
List of Non-Full Blocks (4)



- 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)

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