Relational Database System Implementation

CS122 – Lecture 2

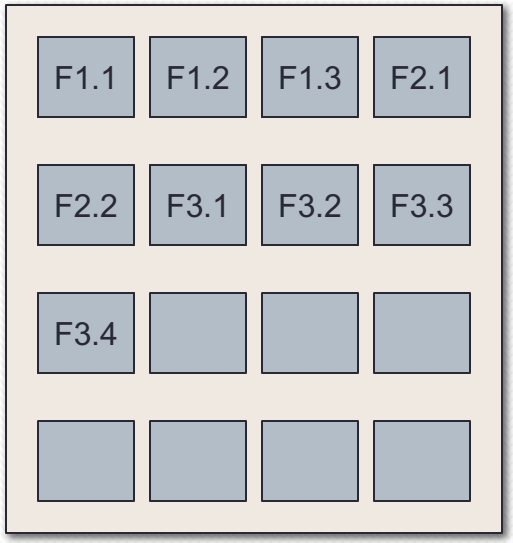
Winter Term, 2017-2018

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…

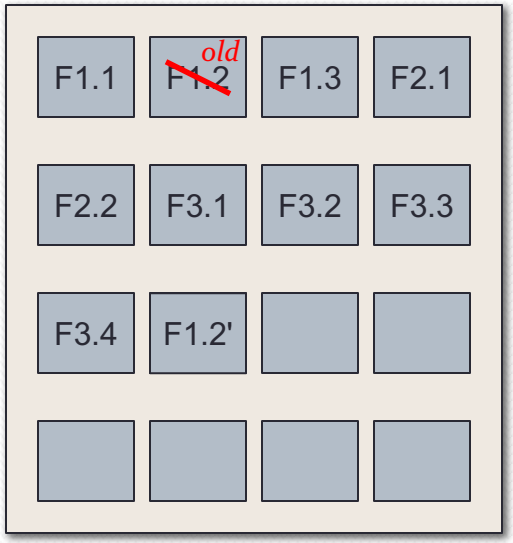
Solid State Drives (2)

- 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)

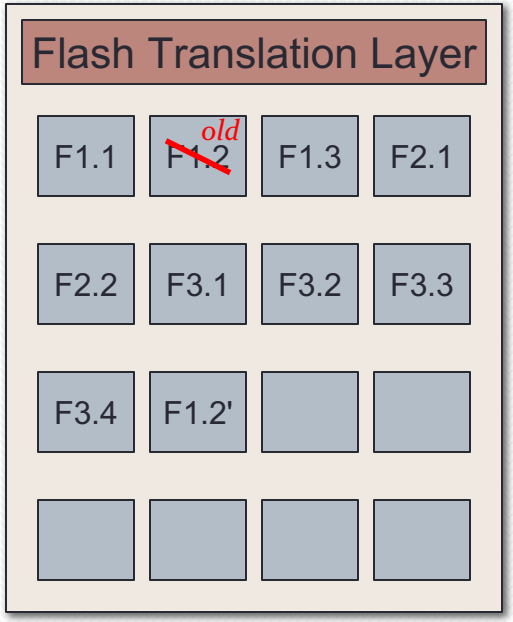
- SSDs can only write to blocks that are currently empty
- Example: we want to modify the data in block 2 of F1
  - Can't just change the data in-place!
- Instead, must write a new version of F1.2
  - SSD marks old version of F1.2 as not in use, and stores a new version F1.2'



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

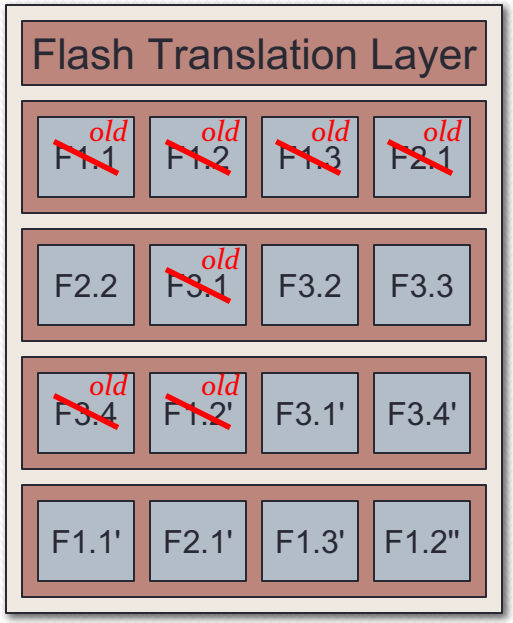
Solid State Drives (4)

- 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



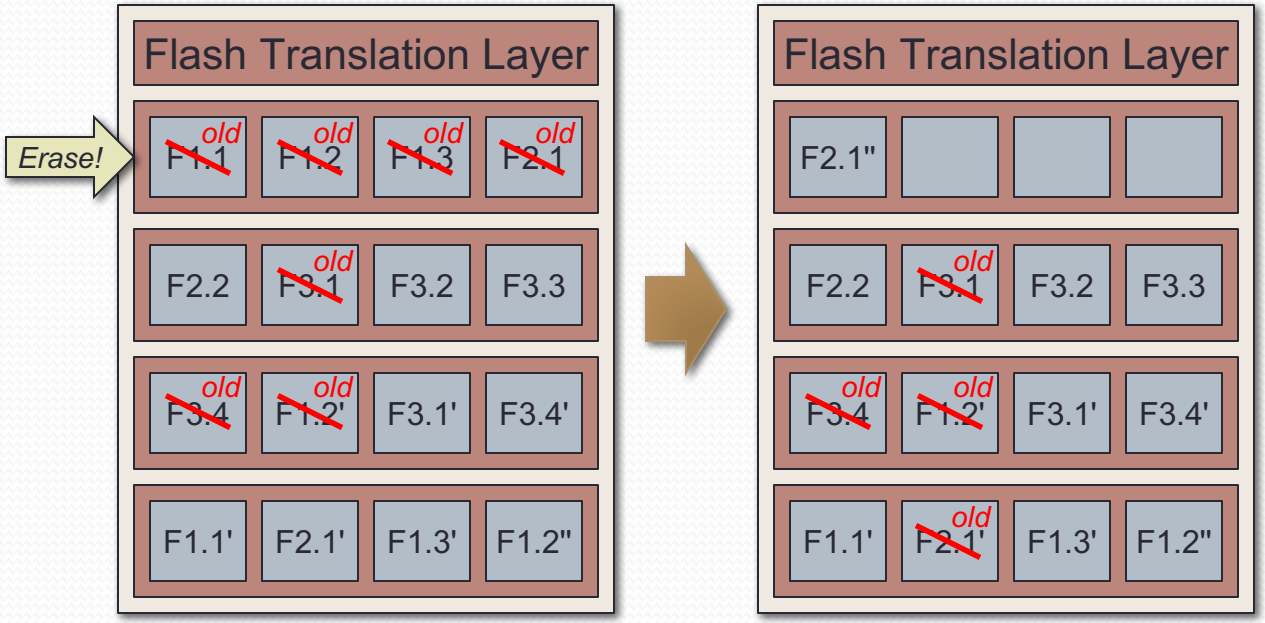
SSDs: Erase Blocks

- 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



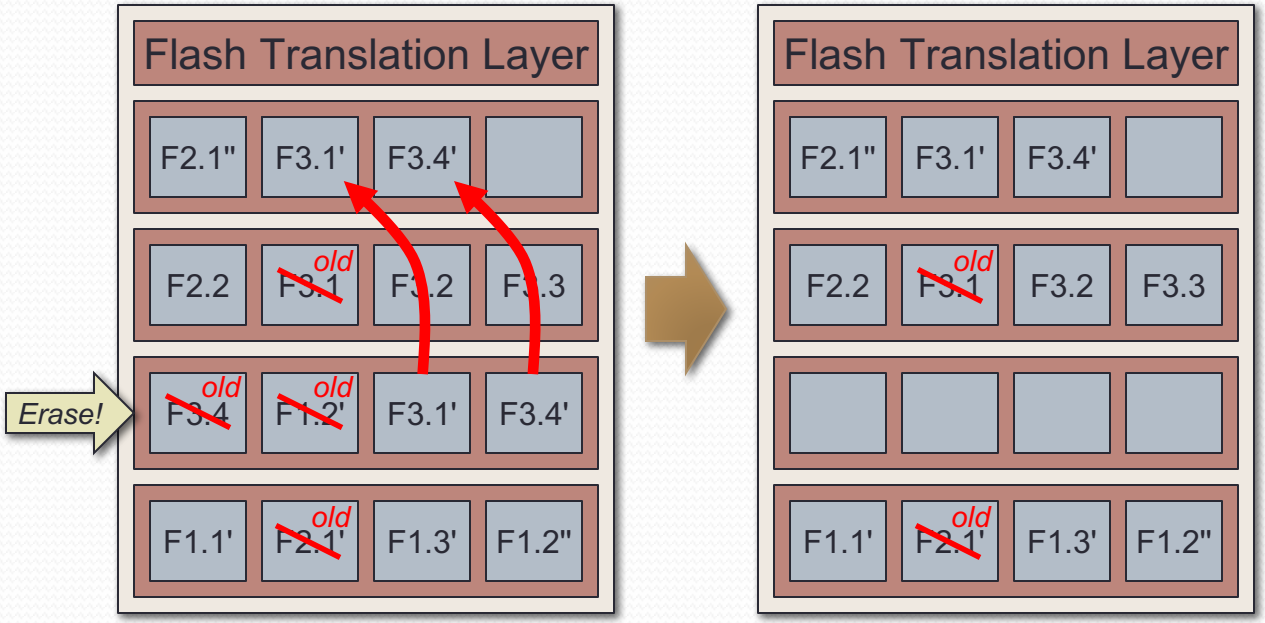
SSDs: Erase Blocks (2)

- 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



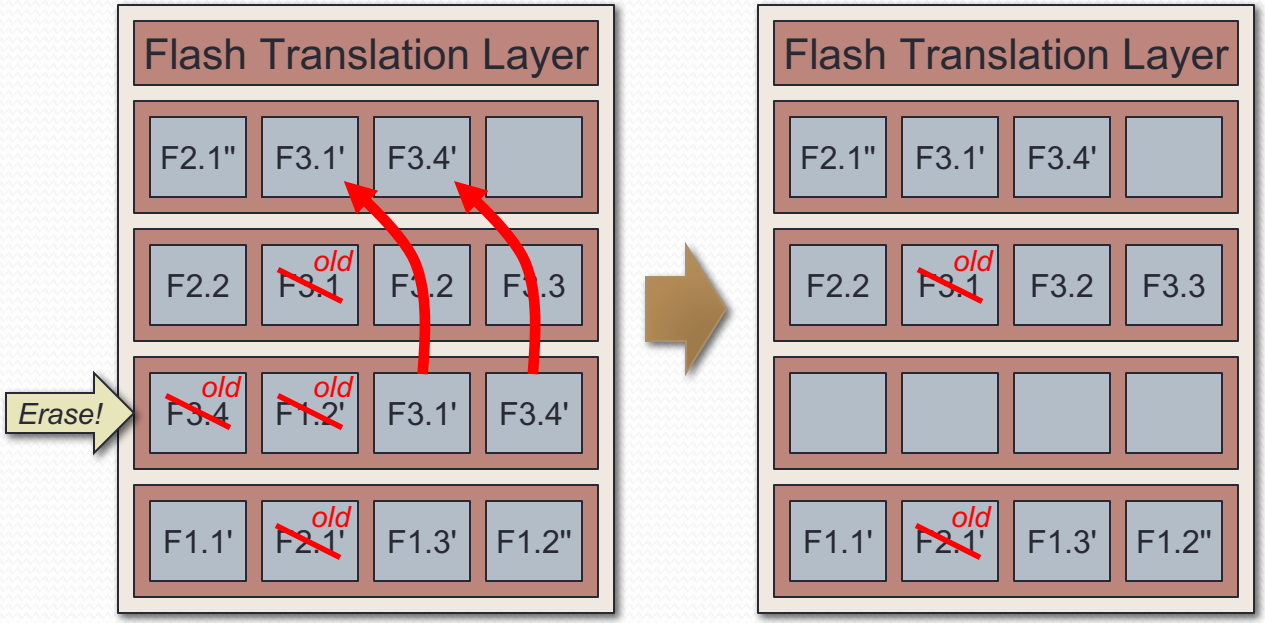
SSDs: Erase Blocks (3)

- 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 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 4KB 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
  - Typically see 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; page size of 4KB
- 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', 'http://…');
```

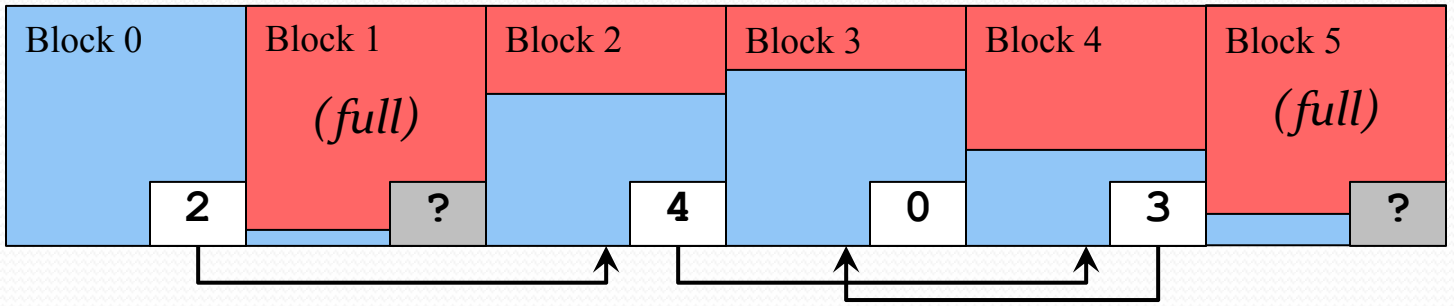
- Database must find a block with enough space to hold the new record
- NanoDB's solution:
  - Starting with first block in data file, search linearly until a block is found with enough space to hold the record
  - If we reach the end of the file, extend the file with a new block and add the record there
- 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
  - e.g. a linked list of blocks with space for more data
  - Often makes it much faster/easier to find free space…
- Can also impact database performance if the approach causes many extra disk seeks and/or block reads

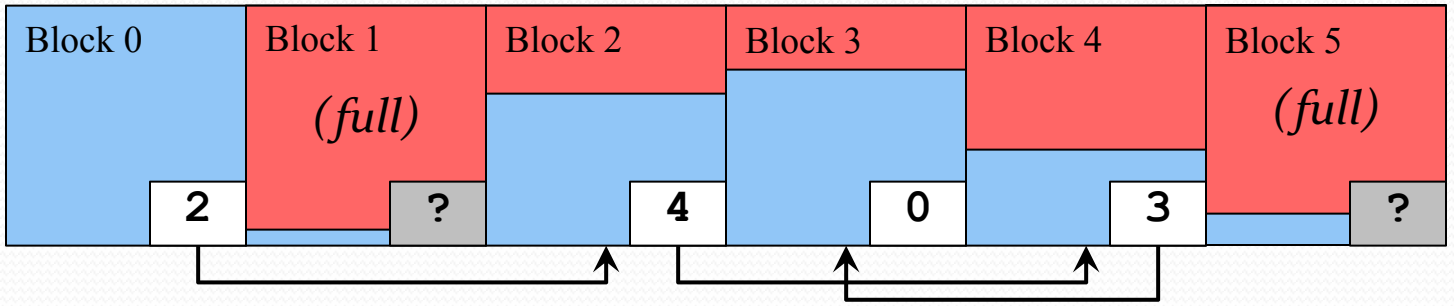
Block-Level Organization

- 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



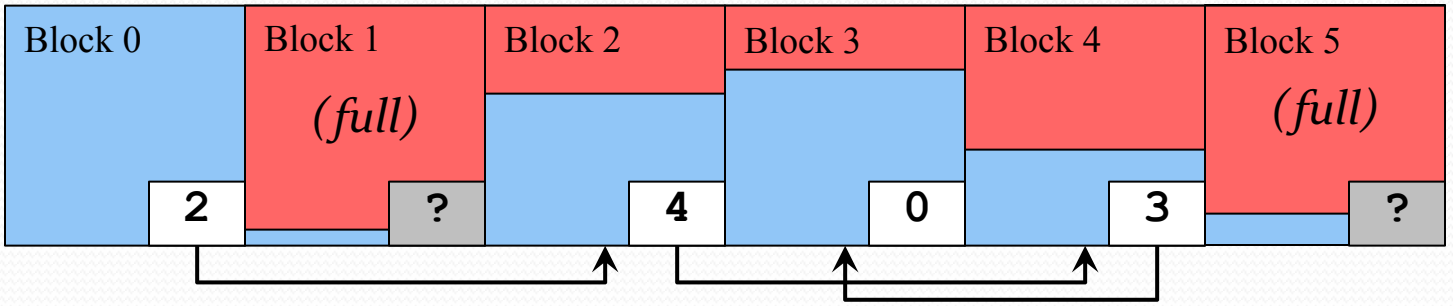
- In NanoDB, block 0 is special:
  - It holds the table-file's schema, among other things

List of Non-Full Blocks (1)



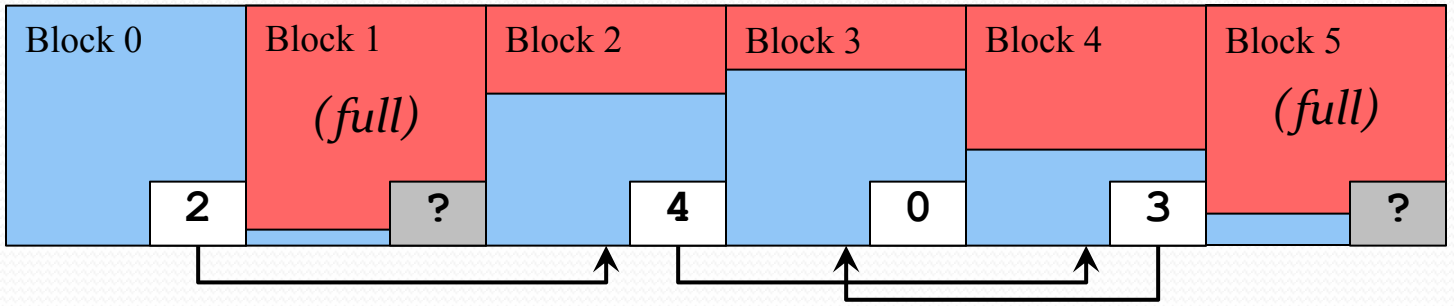
- Note that pages will almost never be *completely* full!
  - List simply specifies pages that can hold another tuple
- Can use the table's schema to compute minimum and maximum size of a tuple for that table

List of Non-Full Blocks (2)



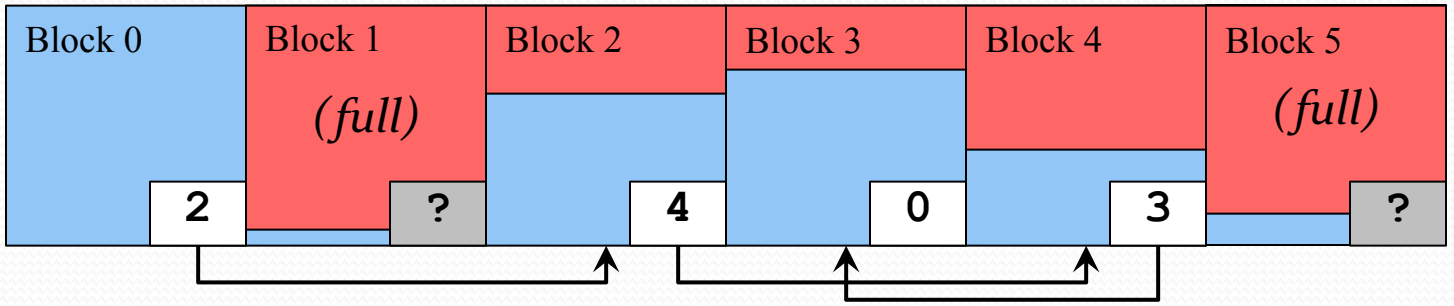
- When a new row is inserted:
  - Starting with first block, search through list of blocks with free space, for space to store the new tuple
  - When space is found, store the tuple
  - If the block is now full, remove it from the list
- Now we sometimes modify *two* pages instead of one

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)

Disk Records and Fields

- Tuples are ordered sets of attribute-value pairs
  - Every attribute has an associated type (a.k.a. “domain”)
  - A value may also be **`NULL`** to represent unknown data
  - The data dictionary specifies the schema for every table
- Issues:
  - Can’t expect a table to have all tuples be the same size
  - Also can’t expect a table to have all non-**`NULL`** values
- Need a way to represent tuples within a disk page, where tuples can vary in size, and some attribute-values are unspecified

Disk Records and Fields (2)

- Fixed-size data types are easy to store into a tuple
  - e.g. **`INTEGER`**, **`CHAR(25)`**, **`DATE`** fields
  - Table's schema records each column's type
    - For columns with size/precision details, these are also stored
  - Just use schema to guide reading/writing the column
- Variable-size values also require a size to be stored
  - e.g. **`VARCHAR(`*`n`*`)`** fields
  - If $n < 256$: store 1-byte size, then string data
  - If $n < 65536$: store 2-byte size, then string data
  - (Can also terminate the field with a special character)

Disk Records and **NULL** Values

- In each tuple, include a bit for each attribute indicating whether its value is **NULL**
  - If bit is 1 then corresponding attribute has a **NULL** value
    - (Don't need to store data for **NULL** attributes in the record…)
  - Store bits in packed format: each byte holds 8 null-bits
  - Called a *null bitmap*
- Example record format:

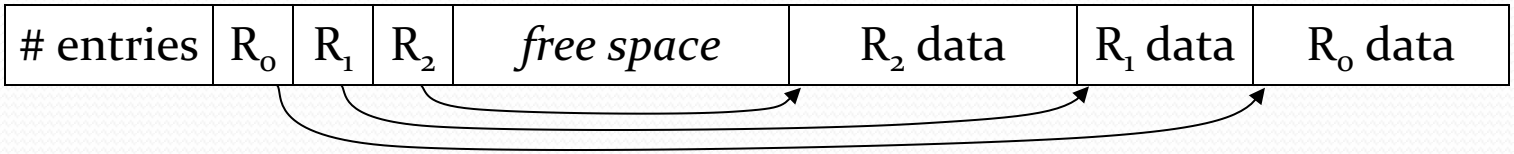
null bitmap	*user_id* (big-endian)	*username*	*name*	*website_url*
`0x04`	`0xF0,0x95,0x01,0x00`	`0x06,'donnie'`	`NULL`	`0x22,'http://www.cs…'`

  - (no data is actually stored for the *name* field)

Variable-Size Record Storage

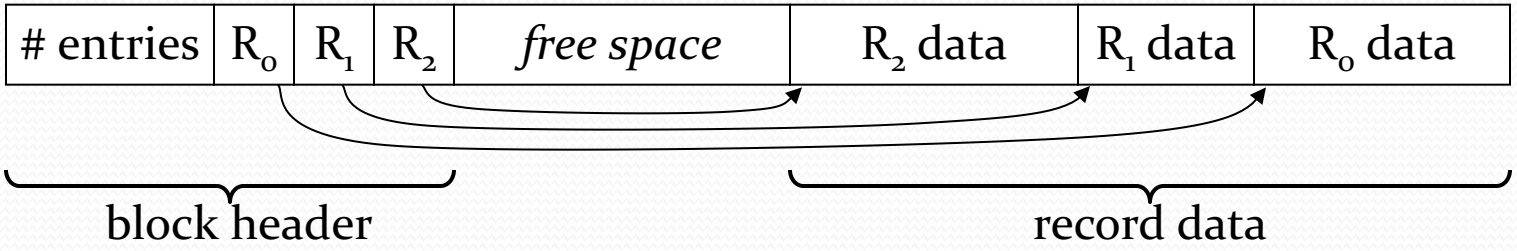
- Some row-values can vary in size
  - **`VARCHAR`**, **`BLOB`**, **`CLOB`**, **`TEXT`**, **`NUMERIC`**, etc. types
  - Also, don't store any value for **`NULL`** fields
- Records will also vary in size

- Variable-size records can be stored into fixed-size blocks using a *slotted-page structure*



Slotted Page Structure (1)

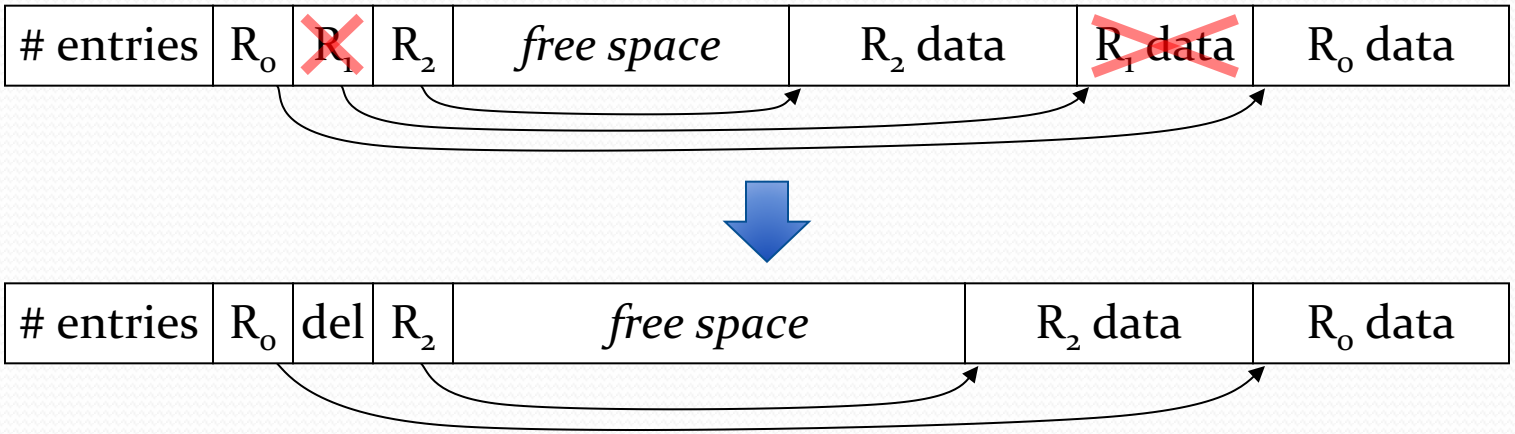
- The slotted-page structure:



- Records in a block are stored contiguously, starting from the *end* of the block
  - Records are stored in reverse order
- Start of block has a header specifying where each record in the block starts
  - First value specifies total number of records *N* in the block
  - Next *N* values specify the starting offset of each row's data

Slotted Page Structure (2)

- When a record is deleted:
  - Record's entry in the index is marked as "deleted"
    - (e.g. its index is set to an invalid value, such as 0)
  - The record's space is reclaimed within the block by moving other records toward end of block
- Example: Delete record 1 from this block:

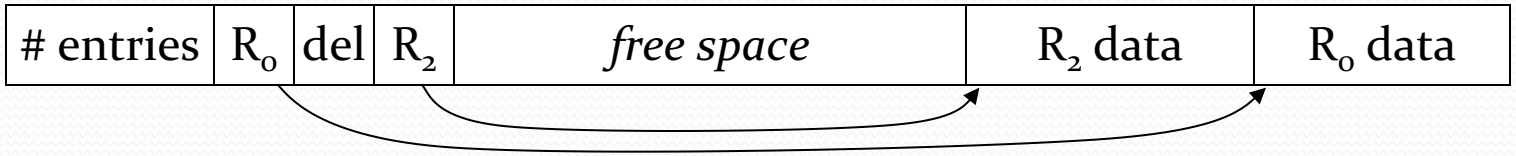


Indexes and Tables

- Table records may be referenced from other files
- Example:
  - Indexes allow specific rows to be found and retrieved, based on the values of some set of attributes
  - The index needs some way to reference a particular record
- Every record has a specific location in a data file:
  - The block the record is stored within
  - The offset of the record within the block
- Example: NanoDB record pointers:
  - Block number (unsigned short: 0 to 65535)
  - Offset within block (unsigned short: 0 to 65535)

Slotted Page Structure (3)

- With the slotted-page structure, records can be referenced by their index in the *block header*
  - Level of indirection allows record data to be moved within the block, without affecting data that references the record



- We can only shrink the slotted-page header when deleted records are at the *end* of the header area
  - e.g. cannot move entry R_2 to index 1 and shrink the header
  - When R_2 is deleted, then we can eliminate both entries
  - Or, if a new row is added to this block, it could occupy R_1