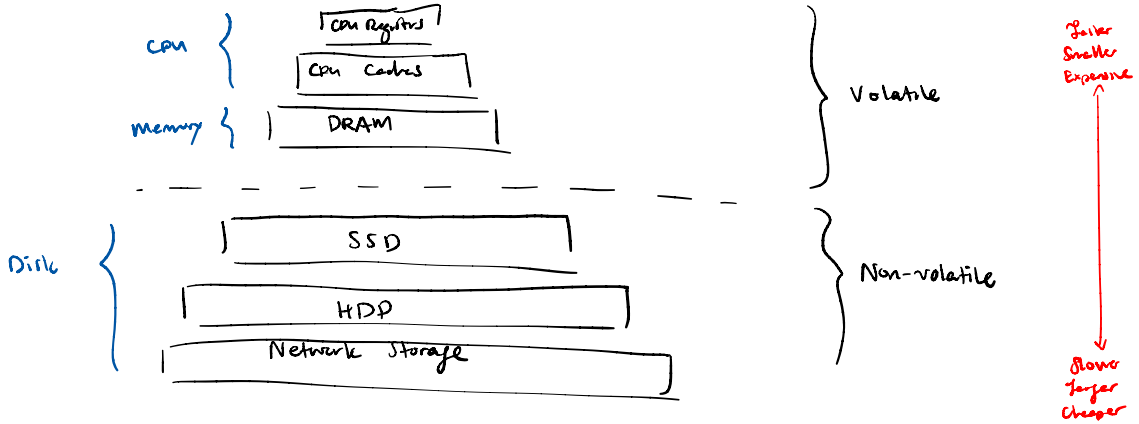
- Query Planning
- Operator Execution
- Access Methods
- Buffer Pool Manager
- Disk Manager

Lecture 1: Data Storage

Disk-Based Architecture

DBMS assumes primary storage of database is non-volatile storage
(if lose power, data is wiped out)

Storage Hierarchy



Access Times

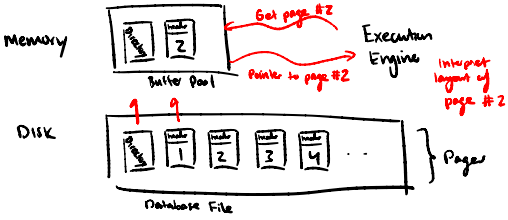
1 ns	L1 Cache Ref	← 1 sec
4 ns	L2 Cache Ref	← 4 sec
100 ns	DRAM	← 100 sec
16,000 ns	SSD	← 4.4 hrs
2,000,000 ns	HDD	← 3.3 weeks
~50,000,000 ns	Network Storage	← 1.5 years
1,000,000,000 ns	Tape Archives	← 31.7 years

Sequential vs Random Access

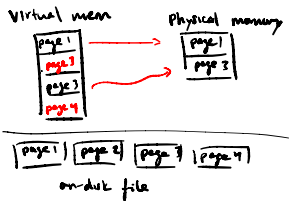
- Random access on non-volatile storage is almost always slower than sequential access
- DBMS want to maximize sequential access
  - Algorithms try to reduce # of writes to random pages so data is stored in contiguous blocks
  - Allocating multiple pages at same time is called an extent

Design Goals

- Allow DBMS manages databases that exceed the amount of memory available.
- Avoid large stalls / performance degradation
  - manage reading/writing to disk
  - maximize sequential access and random access.



Why not use the OS?



Proposed: use mmap (memory mapping) to store contents of a file into address space of a program.
- allow multiple threads to access mmap files to hide page fault stalls.

Problems:

1) Transaction Safety
   - OS can flush dirty pages at any time
2) I/O Stalls
   - DBMS doesn't know which pages are in memory.
   - OS will stall on a page fault.
3) Error Handling
   - difficult to validate pages. (invalid memory address access)
   - any access can cause SIGBUS that DBMS must handle.
4) Performance Issues
   - OS data structure issues
   - TLB shootdowns (when a processor restricts access to a page of shared memory, every processor must flush their Translation Lookaside Buffer)

=> The OS is not your friend

Database Storage

2 main problems to solve

- How DBMS represents the database in files on disk
- How the DBMS manages memory and transfers data to/from disk.

File Storage

DBMS stores a database as one or more files on disk in a proprietary format
- OS doesn't know anything about contents of these files.

The storage manager is responsible for maintaining db's files
- some do their own scheduling for reads and writes to improve spatial/temporal locality of pages.
- organizes files as collection of pages: tracks (data read/written to pages), available space.

Database Pages

- fixed size block of data
  - can contain tuples, meta-data, indexes, log records
  - most systems do not mix page types
  - some systems require pages to be self contained

- each page is given a unique identifier → DBMS uses an indirection layer to map page IDs to physical locations

- there are 3 notions of "pages" in a DBMS:
  1. Hardware page ~4 KB
  2. OS page ~4 KB
  3. Database page 512B - 16KB

  Largest block of data that the storage device can guarantee failsafe writes.

Page Storage Architecture

Different DBMS's manage pages in files on disk in different ways:

1) Heap file organization
2) Tree file organization
3) Sequential/sorted file organization
4) Hashing file organization

* At this level, don't need to know anything about what is inside of the pages *

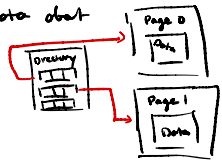
- insert a new tuple:
  1) check page directory to find a page w/ a free slot
  2) Retrieve page from disk (if not in memory)
  3) Check slot array to find empty space in page that will fit

- update existing tuple (using its record id)
  1) check page directory to find location of page
  2) Retrieve page from disk (if not in memory)
  3) Find offset in page using slot array
  4) Overwrite existing data (if new data fits)

Heap file

- unordered collection of pages w/ tuples stored in a random order.
  - Create, get, write, delete Page
  - Must support iterating over all pages.
- Ez to find pages if 1 single file.
- Need meta-data to keep track of what pages exist in multiple files and which ones have free space.

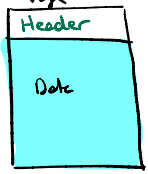
- Page directory
  - DBMS maintains special pages that track location of data pages in the database files
    - need directory pages to be in sync w/ the data pages.
  - the directory also records meta-data about available space:
    1) # of free slots per page
    2) list of free/empty pages



Page Header

Every page contains a header of meta-data about its contents

1) Page size
2) Checksum
3) DBMS version
4) Transaction visibility
5) Compression information



- some systems (eg Oracle) require pages to be self-contained

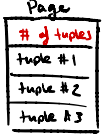
---

Page Layout

For any page storage architecture, need to decide how to organize the data inside the page.

— 2 approaches: ① tuple oriented ② log oriented

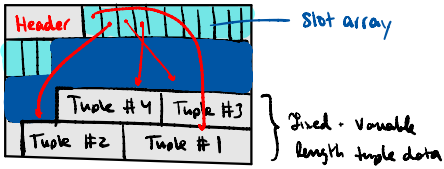
Tuple Storage



- Strawman idea: keep track of # of tuples and simply append new tuple to end.
  - Q: What if we delete a tuple?
    What if we have a variable length attribute?

- Record ID
  - DBMS needs way to keep track of individual tuples: assign a unique record identifier to each tuple.
    Solutions:
    ex: 1. (most common): page_id + offset/slot
    2. can also contain file location information

- Slotted Pages
  - most common layout scheme is slotted pages, where slot array maps "slots" to tuples' starting position offsets.
  - page header keeps track of 1) # of used slots 2) offset of starting location of last slot used.



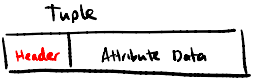
Tuple Layout

a tuple is essentially just a sequence of bytes → It's the job of the DBMS to interpret those bytes into attribute types + values.

- Tuple header

Each tuple is prefixed w/ a header that contains metadata about it, eg 1) Visibility info (concurrency control) 2) Bitmap for Null values

(W) we do NOT need to store metadata about schema!

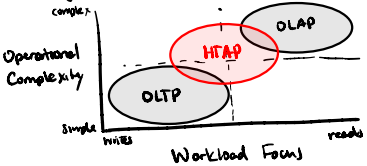


Data representation:

1) Integer / BigInt / SmallInt / TinyInt → C/C++ representation
2) Float / Real v. Numeric/Decimal → IEEE754 / fixed-point decimals
3) Varchar / Var Binary / Text / Blob → (Header w/ length) + (data bytes)
   don't worry about collating/sorting
4) Time / Date / Timestamp → 32/64-bit integer of (micro) seconds since Unix Epoch

Database Workloads

1) On-line Transaction Processing (OLTP): fast operations that only read/update a small amount of data at a time
   think: process data/are transactions
2) On-line Analytical Processing (OLAP): complex queries that read a lot of data to compute aggregates
   think: analyze aggregate data
3) Hybrid Transaction + Analytical Processing: OLTP + OLAP on the same database



OLTP

- Simple Queries that read/update a small amount of data that is related to a single entity in the database

```
Select P.*, R.*
From pages as P
inner join revisions as R
  on P.latest = R.revID
Where P.pageID = ?
```

```
UPDATE useracct
SET lastLogin = NOW(),
    hostname = ?
Where userID = ?
```

```
Insert into revisions
values(?, ?, .., ?)
```

OLAP

- Complex Queries that read large portions of the database spanning multiple entities
- execute these workloads on data collected from OLTP application(s)

```
Select COUNT(U.lastLogin),
       EXTRACT(month from U.lastLogin) AS MONTH
FROM useracct as U
where U.hostname LIKE '%.gov'
Group By EXTRACT(month from U.lastLogin)
```

Data Storage Models

- DBMS can store tuples in different ways that are better for either OLTP or OLAP workloads
  - ↳ we've been assuming n-ary storage model (or row storage) so far

↓

N-ary Storage Model (NSM)

Row Store:

- DBMS stores all attributes for a single tuple contiguously in a page.
- Ideal for ☺ insert-heavy workloads or ☹ OLTP workloads where queries tend to operate only on an individual entity.

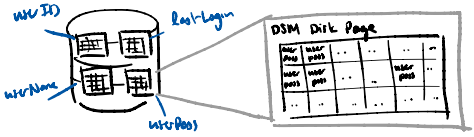
Advantages: ☺ Fast inserts, updates, & deletes
☺ Good for queries that need entire tuple (e.g. select *)

Disadvantages: ☹ Not good for scanning large portions of the table and/or subset of the attributes. (e.g. select count(single Attribute))

Decomposition Storage Model (DSM)

Column Store:

- DBMS stores values of a single attribute for all tuples contiguously in a page
- Ideal for OLAP workloads where read-only queries perform large scans over a subset of table's attributes.



Advantages: ☺ Reduces amount of wasted I/O because DBMS only reads data it needs
☺ Better query processing & data compression

Disadvantages: ☹ Slow for point queries, inserts, updates, & deletes because of tuple splitting/stitching

Observation

! I/O is main bottleneck if DBMS fetches data from disk during query execution

– DBMS can compress pages to increase utility of data moved per I/O operation

– Key trade-off is speed v. compression ratio
  1) Compressing the database reduces DRAM requirements
  2) It may decrease CPU costs during query execution.

Lecture #2

Database Storage

Problem #1 : How DBMS represents the database in files on disk

Problem #2 : How DBMS manages memory and transfers data to/from disk

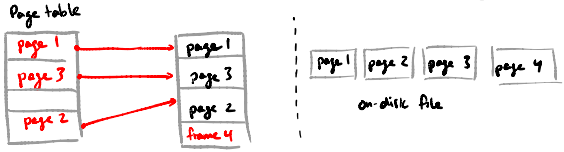
↓ today!

Spacial Control:
- Where to write pages on disk
- The goal is to keep pages that are often accessed together physically close on disk.

Temporal Control
- When to read pages into memory, and when to write them out to disk.
- The goal is to minimize the # of stalls from having to perform disk I/O

Buffer Pool Metadata

- The page table keeps track of pages that are currently in memory
- Also maintains additional meta-data per page:
  1) dirty flag
  2) pin/reference counter



Page table vs. Page Directory

mapping from page IDs to a copy of the page in buffer pool frames
- in-memory data structure that doesn't need to be stored on disk

mapping from page IDs to page locations in database files
- all changes must be recorded on disk to allow DBMS to find them on restart

Buffer Replacement Policies

- When DBMS needs to free up a frame to make room for a new page, it must decide which page to evict from the buffer pool.
- Considerations:
  1) Correctness
  2) Accuracy
  3) Speed
  4) Meta-data overhead.

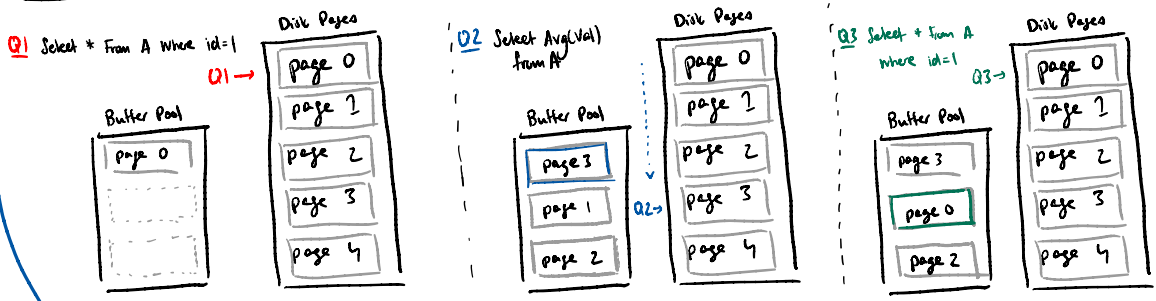
Naive Policies:

① **Least Recently Used (LRU)**
- Maintain a single timestamp for when each page was last accessed
- When the DBMS needs to evict a page, select the one w/ the oldest timestamp
  - keep the pages in sorted order to reduce search time on eviction

② **Clock**
- Approximation of LRU that does not need a separate timestamp per page.
  - each page has a reference bit
  - when a page is accessed, set to 1.
- Organize the pages in a circular buffer with a "clock hand".
  - Upon sweeping, check if a page's bit is set to 1
  - If yes, set to 0. If no, evict.

Problems
- LRU & Clock replacement policies are susceptible to sequential flooding.
  - A query performs a sequential scan that reads every page.
  - This pollutes the buffer pool w/ pages that are read once, then never again.
- In some workloads, the most-recently used (MRU) page is the most un-needed page.

Sequential Flooding



→ **Better Policies:**

(1) LRU-K

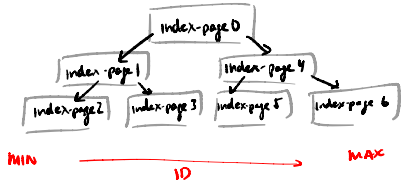
- Track the history of last K references to each page as timestamps and compute the interval between subsequent accesses.
- The DBMS then uses this history to estimate the next time that page is going to be accessed.

(2) Localization

- DBMS chooses which pages to evict on a per-query basis. This minimizes the pollution of the pollution of the buffer pool from each query
  - keep track of the pages that a query has accessed
- Ex: PostGres maintains a small ring buffer that is private to the query.

(3) Priority Hints

- DBMS knows about the context of each page during query execution.
- It can provide hints to the buffer pool about whether or not a page is important



Buffer Pool Optimizations

1) Dirty Page Eviction
2) Background Writing
3) Avoiding the OS
4) Multiple Buffer Pools
5) Pre-fetching
6) Scan-Sharing
7) Buffer Pool Bypass

- Dirty Page Eviction

how to handle dirty pages?

- **Fast Path:** If a page in the buffer pool is not dirty, the DBMS can simply 'drop' it.
- **Slow Path:** If a page in dirty, then the DBMS must write it back to disk to ensure its changes are persisted

- Need to consider trade-off between fast evictions & writing dirty pages that will not be read again

- Background Writing

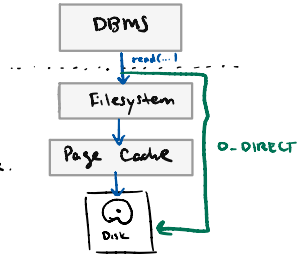
- DBMS can periodically walk through the page table and write dirty pages to disk.
- When a dirty page is safely written, the DBMS can either ① evict page or ② unset the dirty flag.
  - ↳ Need to be careful that the system doesn't write dirty pages before their log records are written.

- OS Page Cache

- Most disk operations go through the OS API. (*)
  Unless the DBMS tells it not to, the OS maintains its own filesystem cache (page/buffer cache)
- Most DBMS's use direct I/O (O_DIRECT) to bypass the OS's cache.

The OS is NOT your friend!!

- Redundant copies of pages
- Different eviction policies
- Loss of control over file I/O (*)



- Multiple Buffer Pools

- DBMS doesn't always have a single buffer pool for the entire system.
  - Multiple buffer pool instances
  - Per-database buffer pool.
  - Per-page type buffer pool.

multiple threads attempt to acquire incompatible latches to the same in-memory structure

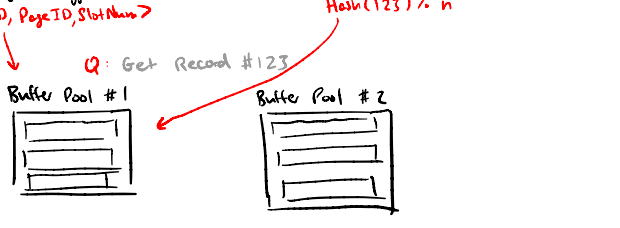
- Partitioning memory access across multiple pools helps reduce latch contention and improve locality.
- Approaches:

1) Object ID

- Embed an object identifier in record IDs and then maintain a mapping from objects to specific buffer pools.

2) Hashing

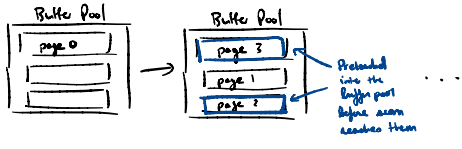
- Hash the page ID to select which buffer pool to access.

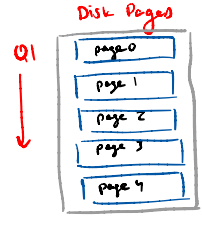


- Prefetching

- DBMS can also prefetch pages based on a query plan: 1) Sequential Scans 2) Index Scans

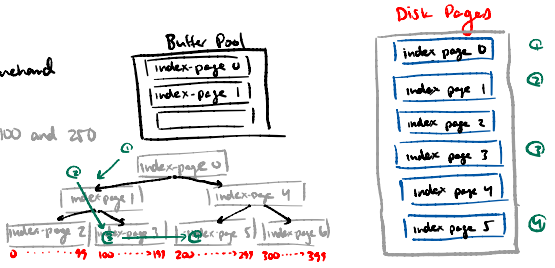
Sequential Scan





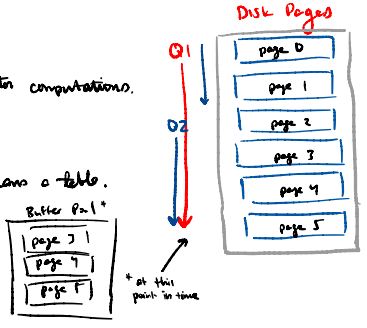
Index Scan

- need to know access patterns beforehand
- ex: Q2 : Select * from A
  Where val Between 100 and 250



- Scan Sharing

- Queries can reuse data retrieved from storage or operator computations.
  - called synchronized scans
  - different from result caching
- Allows multiple queries to attach to a single cursor that scans a table.
  - queries do not have to be the same
  - can share intermediate results
- "second query can attach to current scanning pointer instead of restarting scan"



- Buffer Pool Bypass

- aka light scans
- sequential scan operator will not store fetched pages in the buffer pool to avoid overhead.
  - memory is local to running query
  - works well if operator needs to read a large sequence of pages that are contiguous on disk.
  - can be used for temporary data (sorting, joins, etc)

Other Memory Pools

- DBMS needs memory for things other than just tables & indexes
- Other memory pools are not always backed by disk. Implementation-dependent.
- Ex:
  - Sorting + Join Buffers
  - Query Caches
  - Maintenance Buffers
  - Log Buffers
  - Dictionary Caches

TL;DR : The DBMS can almost always manage memory better than the OS

- Leverage the semantics of the query plan to make better decisions:
  1) Evictions
  2) Allocations
  3) Pre-fetching

Lecture #3

Today's Agenda: Discuss how to support DBMS's execution engine to efficiently read/write data from pages.

↳ 2 types of data structures: ① Hash tables ② Trees

- Internal meta-data
- Core data storage
- Temporary data structures
- Table indexes

Design Decisions

① Data organization: how we lay out data structure in memory/pages and what information to support efficient access

② Concurrency: how to enable multiple threads to access the data structure at the same time w/o causing problems.

Hash Tables

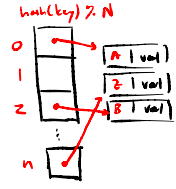
- A hash table implements an unordered associative array that maps keys to values.
- It uses a hash function to compute an offset into this array for a given key, from which the desired value can be found.

  Space Complexity: O(n)

  Time Complexity: O(1) average, O(n) worst

  } Note: DBMS needs to care about constants! (Asymptotic complexity isn't sufficient metric)

- (Naive) Example: Static hash tables.
  - Allocate a giant array that has one slot for every element you need to store.
  - To find an entry, mod the key by # of elements to find offset into the array.
  - Necessary assumptions:
    1) # elements is known ahead of time & fixed
    2) Each key is unique
    3) Perfect hash function: key1 ≠ key2 ⇒ hash(key1) ≠ hash(key2)



- Design Decisions

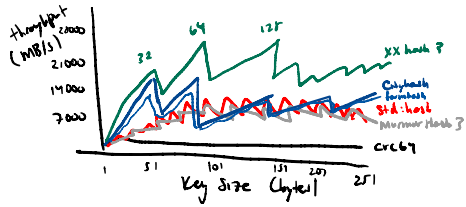
  1) Hash Function:
     - how to map a large key space into a smaller domain
     - trade-off between ① speed & ② collision rate

  2) Hashing Scheme:
     - how to handle key collisions after hashing
     - trade-off between ① allocating large hash table & ② additional instructions to get/put keys.

1)

Hash Functions

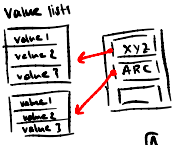
- For any input key, return an integer representation of that key
- Want: something fast + low collision rate
  - ↳ do NOT want cryptographic hash (eg SHA-2) for DBMS hash tables
- EX:
  - CRC-64 (1975) - error detection in networking
  - MurmurHash (2008) - fast general purpose
  - Google CityHash (2011) - fast for short keys (<64 bytes)
  - Facebook XXHash (2012) - from creator of zstd compression — State of the art
  - Google FarmHash (2014) - newer CityHash w/ better collision rates



2) Static Hashing Schemes

1) Linear Probe Hashing

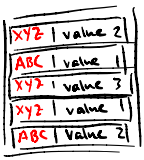
- single giant table of slots
- resolve collisions by linearly searching for the next free slot in the table
  - to determine whether an element is present: hash to a location and scan for it.
  - must store the key in the index to know when to stop scanning
  - insertions + deletions are generalizations of lookups



- Q: How to handle non-unique keys:

  Choice #1: Separate linked list
  - store values in separate storage area for each key

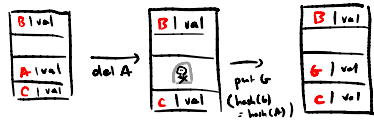
  Choice #2: Redundant keys
  - store duplicate key entries together in the hash table.
  - easier to implement - do what most systems do.



Deletes:

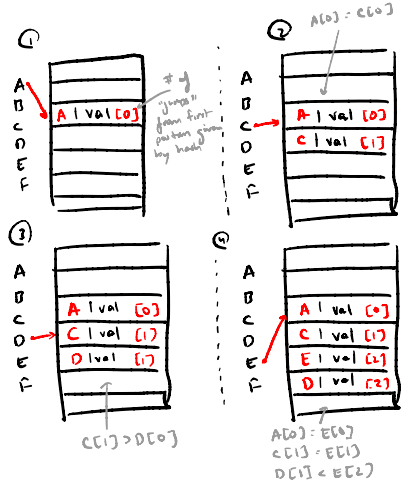
Approach 1: Re-hash
- re-hash and update the slots for all keys.
- nobody actually does this

Approach 2: Tombstone
- set marker indicating that entry in the slot is logically deleted
- can reuse the slot for inserting new keys
- may require periodic garbage collection



2) Robinhood Hashing

- variant of linear probe hashing that steals slots from "rich" keys and gives them to "poor" keys
- WDYM?
  1) Each key tracks the # of positions they are from optimal position in the table
  2) On insert, a key takes the slot of another key if the first key is farther away from optimal position than second key.

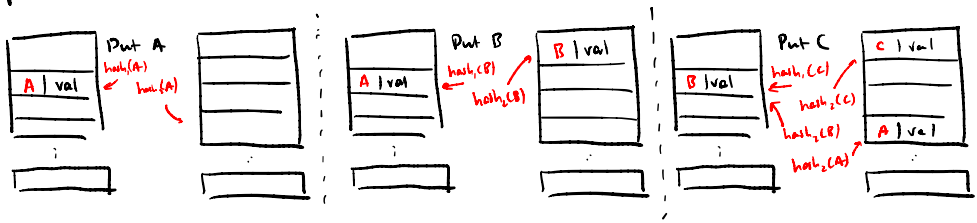


3) Cuckoo Hashing

- use multiple hash tables w/ different hash function seeds
- on insert, check every table and pick anyone that has a free slot.

  ↳ If no table has a free slot, evict the element from one of them and re-hash it to find a new location

- Lookups / deletions are always O(1) b/c only one location per hashtable is checked.



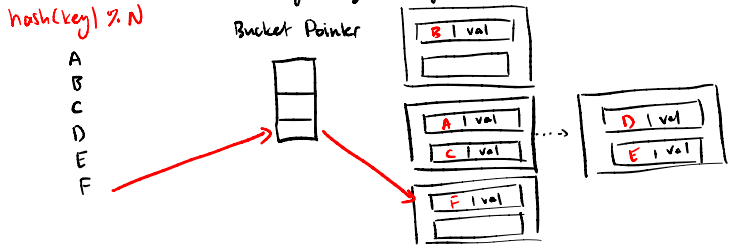
Observation about static hash tables:

- Previous hash tables require DBMS to know # of elements it wants to store.
  - ↳ Otherwise, it must rebuild the table if it needs to grow/shrink in size.
- Dynamic hash tables resize themselves on demand
  1) Chained Hashing
  2) Extendible Hashing
  3) Linear Hashing

3 Dynamic Hashing Schemes

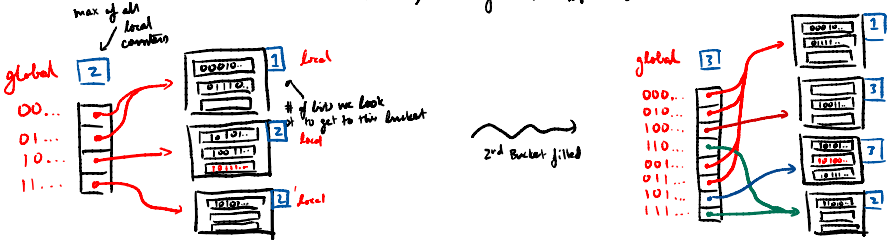
1) Chained Hashing

- Maintain a linked list of buckets for each slot in the hash table.
- Resolve collisions by placing all elements w/ the same hash key in the same bucket.
  - to determine whether an element is present, hash to its bucket & scan for it.
  - insertions / deletions are generalizations of lookups.



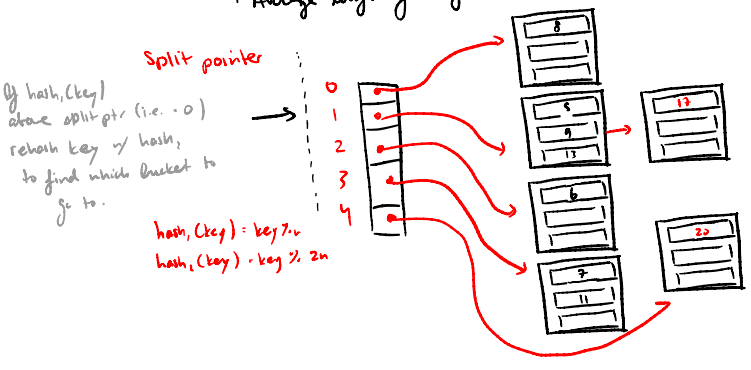
2) Extendible Hashing

- Chained-hashing approach where we split buckets instead of letting the linked list grow forever.
- Multiple slot locations can point the same bucket chain
- Reshuffle bucket entries on split and increase the # of bits to examine
  - Data movement is localized to just the split chain



3) Linear Hashing

- Hash table maintains a pointer that tracks the next bucket to split.
  - When any bucket overflows, split the bucket at the pointer location
- Use multiple hashes to find the right bucket for a given key.
- Can use different overflow criterion:
  1) Space utilization
  2) Average length of overflow chains.



Find 6
$hash_1(6) = 6 \% 4 = 2$

Insert 17
$hash_1(17) = 17 \% 4 = 1$

Find 20
$hash_1(20) = 20 \% 4 = 0$ ← above split pointer
$hash_2(20) = 20 \% 8 = 4$

Find 9
$hash_1(9) = 9 \% 4 = 1$

- When the pointer reaches the last slot, delete the first hash function and move back to the beginning

---

Conclusion:

- Hash tables are fast data structures that support O(1) lookups and are used throughout DBMS internals
  - tradeoff between speed + flexibility
- They are usually not what you want to use for a table index..
- Next: B^+ Trees (aka "The Greatest Data Structure of All Time")