UMD DATA605 - Big Data Systems

# Storage Query Processing Transactions

**Instructor**: Dr. GP Saggese - gsaggese@umd.edu**
**TAs**: Krishna Pratardan Taduri, kptaduri@umd.edu Prahar Kaushikbhai Modi, pmodi08@umd.edu
**v1.1**

SCIENCE ACADEMY

# Outline

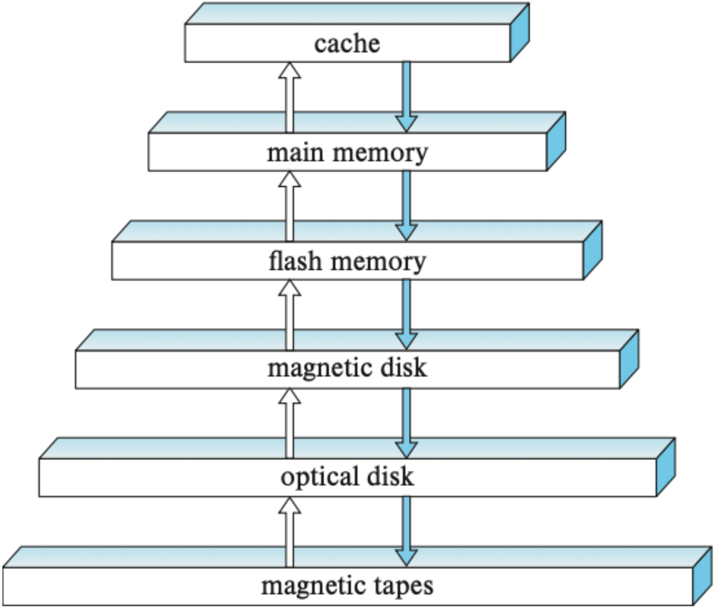
- Storage
  - Physical storage
    - **Storage Hierarchy**
    - Magnetic disks / SSD
    - RAID
  - Logical storage
- Query Processing
- Transactions Sources: Silberschatz et al. 2020, Chap 12, Physical Storage Systems

SCIENCE ACADEMY

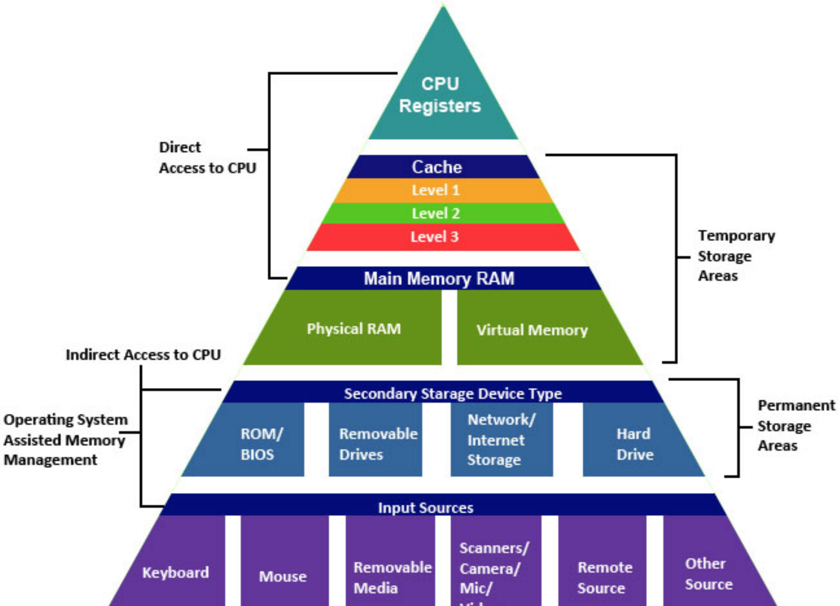
# Storage Characteristics

- **Storage media presents a trade-off between:**
  - Speed of access (e.g., 200MB / sec)
  - Cost per unit of data (e.g., 50 USD / GB)
  - Reliability
- **Volatile vs non-volatile storage**
  - **Volatile**: loses contents when power switched off
  - **Non-volatile**: can survive failures and system crashes
- **Sequential vs random access**
  - **Sequential**: read the data contiguously
- **SELECT * FROM employee**
  - **Random**: read the data from anywhere at any time
- **SELECT * FROM employee**
- **WHERE name LIKE 'Einst*'**
  - Need to know how data is stored in order to optimize access

# Storage Hierarchy



# Storage Hierarchy



# How Important is Memory Hierarchy?

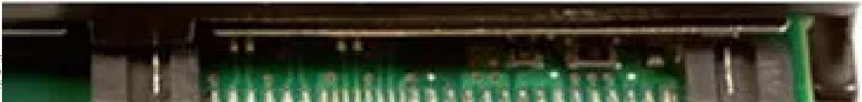
- **Trade-offs shifted drastically over last 10-15 years**
  - Several innovations
    - Large memories
    - SSDs
    - Fast network
  - However, the volume of data is growing quite rapidly
- **Observations**
  - Cache is playing more and more important role
  - In-memory DBs
    - Enough memory that data often fits in memory of a cluster of machines
  - It's faster to access another computer's memory through network than accessing your own disk
  - "Disk" considerations less important
    - HDDs vs SSDs
    - Still disks are where most of the data lives today
- **Similar changes for algorithms**
  - Design and pick algorithms based on current trade-offs
  - E.g., many algorithms designed to minimize access time (e.g., elevator algorithms for disk access)

# Outline

- Storage
  - Physical storage
    - Storage Hierarchy
    - **Magnetic disks / SSD**
    - RAID
  - Logical storage
- Query Processing
- Transactions

# Connecting Disks to a Server

- **Disks (magnetic and SSDs) can be connected to computer:**
  - Directly through high-speed interconnection, or
  - Through high-speed network
- **Through a high-speed interconnection**
  - Serial ATA (SATA)
  - Serial Attached SCSI (SAS)
  - NVMe (Non-volatile Memory Express)
- **Through high-speed networks**
  - Storage Area Network (SAN)
    - ISCSI
    - Fiber Channel
    - InfiniBand
- Network Attached Storage (NAS)
  - Provides a file-system interface (e.g., NFS)
  - Cloud storage
    - Data is stored in the cloud and accessed via an API
    - Object store
    - High latency



# Magnetic Disks



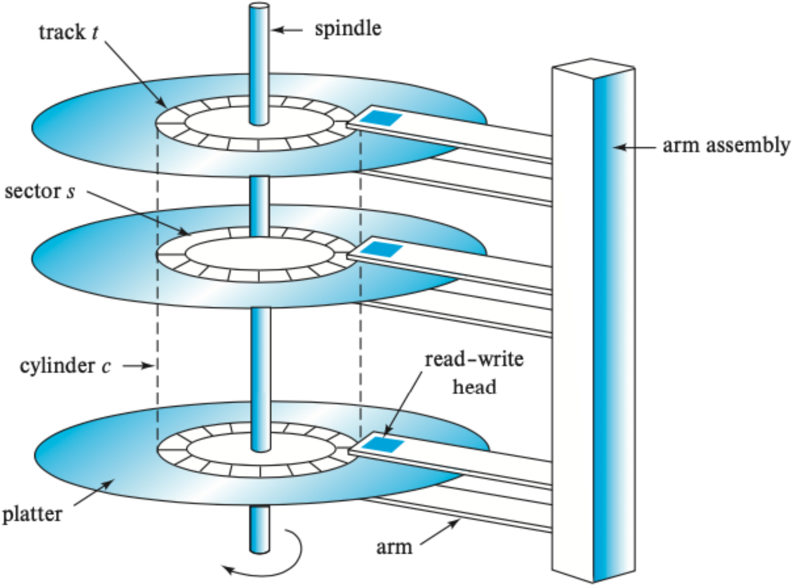
SCIENCE ACADEMY

# Magnetic Disks

- **1979**
- SEAGATE
- 5MB

From Computer Desktop Encyclopedia
Reproduced with permission.


# Magnetic Disks: Components



# Magnetic Disks: Current Specs

- **Capacity**
  - Terabyte and more
- **Access time**
  - = time to start reading data
  - Seek time to move the arm across cylinders (2-20ms)
  - Rotational latency time = wait for sector to be accessed (4-12ms)
- **Data-transfer rate**
  - Once the data is reached, the transfer begins
  - Transfer rate = 50-200MB / secs
  - Sector (disk block) = logical unit of storage (4-16KB)
  - *Sequential access* = when the blocks are on the same or adjacent tracks
  - *Random access* = each request requires a seek
    - IOPS (Input / Output Per Sec) = number of random single block accesses in a second (50-200 IOPS)
- **Reliability**
  - Mean time to failure (MTTF) = the average amount of time that the system runs continuously without a failure
  - Lifespan of an HDD is ~5 years

# Accessing Data

- **Random data transfer rates**
  - How long it takes to read a random sector*
  - Seek time
    - = time to move head to the track
    - Average 2 to 20ms
  - Rotational latency
    - = wait for the sector to get under the head
    - Average 4 to 12ms
  - Transfer time
    - Very low
  - About 10ms per access
    - So if randomly accessed blocks, can only do 100 block transfers
    - 100 / sec x 4 KB per block = 50KB/s
- **Serial data transfer rates**
  - Rate at which data can be transferred (without any seek)
  - 30-50MB/s to up to 200MB/s
- **Reading random data is 1000x slower than reading serial data**
  - Seeks are very bad!

Sequential access

# Solid State Disk (SSD)

- Mainstream around 2000s
- Like non-volatile RAM (NAND and NOR)
- **Capacity**
  - 250-500 GBs (vs 1-10 TB for HDD)
- **Access time**
  - Latency for random access is 1,000x smaller than HDD
    - E.g., 20-100 us (vs 10 ms HDDs)
  - Multiple random requests (e.g., 32) in parallel
  - 10,000 IOPS (vs 50-200 for HDDs)
  - Require to read an entire "page" of data (typically 4KB)
    - Equivalent to a block in magnetic disks
- **Data-transfer rate**
  - 1 GB/s (vs 200 MB/s HDD)
  - Typically limited by the interface speed
    - 500MB/s for SATA
    - 2-3 GB/s for NVMe
  - Lower power consumption than HDDs
  - Writing to SSD is slower than reading (~2-3x)
    - It requires erasing all pages in the block
- **Reliability**
  - There is a limit to how many times a page can be erased (~1m times)

SCIENCE ACADEMY

# Outline

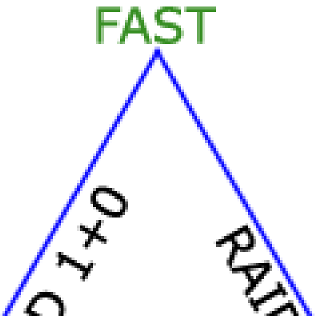
- Storage
  - Physical storage
  - Storage Hierarchy
  - Magnetic disks / SSD
  - **RAID**
  - Logical storage
- Query Processing
- Transactions

# RAID

- RAID = Redundant Array of Independent Disks
- **Problems**
  - Storage capacity has been growing exponentially
  - Data-storage requirement (e.g., web, DBs, multimedia applications) has been growing even faster
  - When you have a lot of disks, the MTTF between failure of any disk gets smaller (e.g., days)
    - If we store a single copy of the data, the frequency of data loss is unacceptable
- **Observations**
  - Disks are very cheap
  - Failures are very costly
  - Use "extra" disks to ensure reliability
    - Store data redundantly
    - If one disk goes down, the data still survives
  - Bonus: allow faster access to data
- **Goal**
  - Expose a logical view of a single large and reliable disk from many unreliable disks
  - Different reliability vs performance (RAID levels)

# Improve Reliability / Performance with RAID

**Increase reliability** - Use redundancy - Store the same data multiple times on multiple disks - E.g., mirroring - If a disk fails, the data is not lost but it can be reconstructed - Increased MTBF - Assumption is independence of disk failure - As disks age, probability of failure increases together - Power failures and natural disasters **Increase performance** - It depends on how data is replicated - Striping data across multiple disks (RAID 0) - = use multiple disks with no replication - Increase number of read requests - Same transfer rate - Mirroring (RAID 1) - = data is copied on multiple disks - Same number of read requests - Increase transfer rate



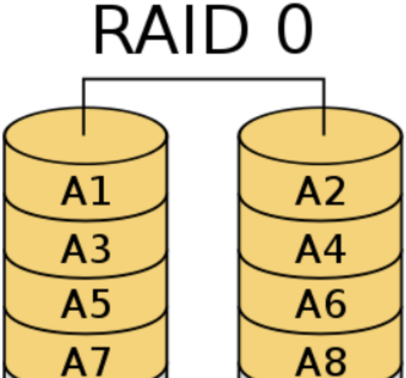
# Error Correction Codes

- = a technique used for controlling errors in data transmission over unreliable communication channels
- **Idea**:
  - the sender encodes the message in a redundant way
  - the receiver can detect errors and correct (a limited number of) errors
- 1940-1960s: Hamming, Reed-Solomon, Shannon, Viterbi
- E.g., triple redundancy
  - Send the same bit 3 times, receiver does majority voting
  - Detect and correct one bit errors
- E.g. parity bit
  - Add an extra bit representing the number of 1s
  - Detect (but not correct) one bit errors

| Triplet received | Interpreted as |
|---|---|
| 000 | 0 (error-free) |
| 001 | 0 |

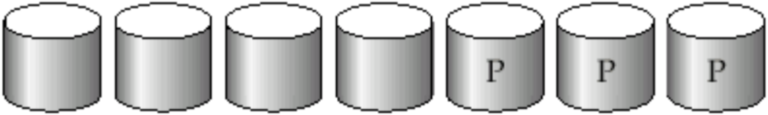
# RAID Levels

Good article on wikipedia **RAID 0: Striping / no redundancy** - = array of independent disks - Same access-time - Increase transfer rate **RAID 1: Mirroring** - = make a copy of the disks - If one disk fails, you have a copy of the data - Like double redundancy in ECC - Also you have parallel access to multiple disks - Reads - Can go to either disk - Same access time - Increase read latency with same transfer rate - Same read latency with increased transfer rate - Writes - Need to write to both disks

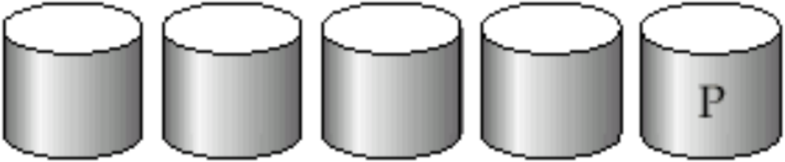


# RAID Levels

**RAID 2: Memory-style error correction** - = use extra bits so we can reconstruct data (like ECC in RAM) - Can trade-off different levels of error detection and recovery **RAID 3: Interleaved parity** - = one disk contains "parity" for the main data disks - Can handle a single disk failure - Little overhead (only 25% in the above case) **RAID 5: Block-interleaved distributed parity** - Distributed parity blocks instead of bits

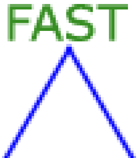



(c) RAID 2: memory-style error-correcting codes



# Choosing a RAID Level

- Main choice between RAID 0, RAID 1, and RAID 5
- **RAID 0 (striping)**
  - Better performance but no additional reliability
- **RAID 1 (mirroring)**
  - Better performance and reliability
  - High cost
  - E.g., to write a single block
    - RAID 1: only requires 2 block writes
    - RAID 5: 2 block reads and 2 block writes
  - Preferred for applications with high update rate and small data (e.g., log disks)
- **RAID 5 (interleaved parity)**
  - Lower storage cost
  - Preferred for applications with low update rate and large amounts of data (e.g., analytics)



# BACKUP

# Outline

- Storage
  - Physical storage
  - **Logical storage**
    - File Organization
    - Buffer Manager
    - Indexes
- Query Processing
- Transactions Sources:
- Silberschatz et al. 2020, Chap 13: Data Storage Structures

# (Centralized) DB Internals

- User processes
  - Issue commands to the DB
- Server processes
  - Receive commands and call into the DB code
- Process monitor process
  - Monitor DB processes
  - Recover processes from failures
- Lock manager process
  - Lock grant / release
  - Deadlock detection
- Database writer process
  - Output modified buffer blocks to disk on a continuous basis
- Log writer process
  - Output log records to stable storage
- Checkpoint process
  - Perform periodic checkpoints
- Shared memory
  - Contain all shared data
    - Buffer pool, Lock table, Log buffer (log records waiting to be saved on stable storage), Caches (e.g., query plans)
  - Data needs to be projected by mutual exclusion locks

# DB Internals

- **Storage hierarchy**
  - How are tables mapped to files?
  - How are tuples mapped to disk blocks?
- **Buffer Manager**
  - Bring pages from disk to memory
  - Manage the limited memory
- **Query Processing Engine**
  - Given a user query, decide how to “execute” it
  - Specify sequence of pages to be brought in memory
  - Operate upon the tuples to produce results

# BACKUP

# Outline

- Storage
  - Physical storage
  - Logical storage
    - **File Organization**
    - Buffer Manager
    - Indexes
- Query Processing
- Transactions

# File Organization

- We need to map onto disk files and blocks (aka *pages*):
  - **Tables** (aka *relations*)
  - **Records** (aka *tuples*, *rows*)
  - **Attributes** (aka *fields*, *columns*)
- **How are tables mapped to disk blocks?**
  - File is a collection of blocks (typically 4-16 KBs)
  - Mapping of tables to file
    - One table to one file
    - Advantages in storing multiple tables clustered together (e.g., joins)
  - Use a standard OS file system
  - High-end DB systems have their own OS
    - OS interferes more than helps in many cases
    - Often DB needs to be aware of underlying blocks
    - Read / write directly disk blocks

| *ID* | *name* | *dept_name* | *salary* |
|---|---|---|---|
| 10101 | Srinivasan | Comp. Sci. | 65000 |
| 12121 | Wu | Finance | 90000 |
| 15151 | Mozart | Music | 40000 |

# File Organization

- **How are records mapped to disk blocks?**
  - Fetch a particular record, specified by record id
  - Find all records that match a condition (say SSN = 123)
  - Insert / delete of records in the table
- Simplest case
  - Each table is mapped to a file
  - Each record is contained in a single block (i.e., records are not split between blocks)
- **Problem**
  - Attributes are of different sizes
- **Solutions**
  - Fixed-length records
  - Variable-length records

| *ID* | *name* | *dept_name* | *salary* |
|---|---|---|---|
| 10101 | Srinivasan | Comp. Sci. | 65000 |
| 12121 | Wu | Finance | 90000 |
| 15151 | Mozart | Music | 40000 |

# Records -> Blocks

- **Which block of a file should a record go to?**
  - Heap organization
    - Records are allocated contiguously in no particular order
    - A new record is appended at the end of free space and the header is updated
    - A new record is deleted, the space is freed, the entry is deleted
    - Need to scan the records to find for "SSN = 123"
  - Sorted by key
    - Sequential organization
    - Keeping it sorted is complex
    - Can search with binary search O(log(n))
  - Based on a hash key
    - Hashing organization
    - E.g., Store the record with SSN = x in the block number x%1000

# Record Organization

- Heap organization
  - Records are allocated contiguously in no particular order
  - A new record is appended at the end of free space and the header is updated
  - A new record is deleted, the space is freed, the entry is deleted
- Sequential organization
- Keep sorted by some search key
- Insertion
  - Find the block in which the tuple should be
  - If there is free space, insert it
  - Otherwise, must create overflow pages
- Deletions
  - Delete and keep the free space
  - Databases tend to be insert heavy, so free space gets used fast
- Can become *fragmented*
  - Must reorganize once in a while
- What if I want to find a particular record by value ?
  - *Account info for SSN = 123*
- Binary search
  - Takes *log(n)* number of disk accesses
    - Random accesses

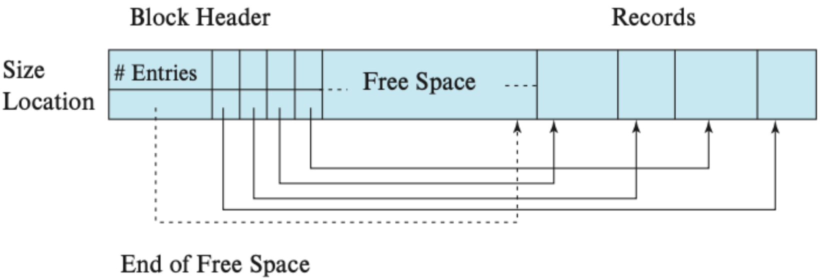
# Fixed-length Records

- **Physical layout of records on blocks**
  - n = number of bytes per record
  - E.g., a record in *instructor* requires 53 bytes
    - char -> 1 byte
    - numeric(8, 2) -> 8 bytes
    - 5 + 20 + 20 + 8 = 53
- **Fetch i-th record**
  - Store record *i* at position
    - offset = (i – 1) * n
  - The block can contain a non-integer number of records
    - Leave bytes unused
- **Insert a new record**
  - Depends on the policy used for deletion
  - Simply append at the end of the record
- **Delete a record**
  - Deletes are much less frequent than insertions
  - Rearrange (move all the records to reclaim space)
  - Keep a *free list* and use for next insert
    - Like a linked list of free blocks (e.g., after deleting record 1, 4, 6)
  - Conceptually store pointers, in practice file header

SCIENCE

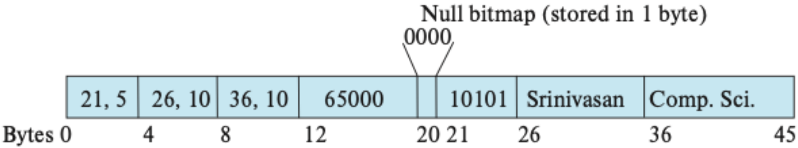
**type** *instructor* = **record**

# Variable-length Records




- **Variable-length records occur** due to: - The presence of variable-length fields (e.g., strings, arrays, sets) - Multiple record types in the same file / blocks - **How to extract a record of variable-length from a block?** - Slotted-page structure - Number of record entries - Array with location and size of each record (indirection) - No pointers directly to records - Pointers to the entry in the header that contains the location of the record - The records may move inside the block, but the outside world is oblivious to it - End of free space in the block

# Variable-length Attributes




- **How to extract an attribute of variable-length of a record?** - The fixed-length attributes are allocated statically at the beginning - Variable-length attributes (e.g., VARCHAR type) are represented by an offset and length - How to represent the value NULL? - Null bitmap - Store which attributes of the record have a null value

# Outline

- Storage
  - Physical storage
  - Logical storage
    - File Organization
    - **Buffer Manager**
    - Indexes
- Query Processing
- Transactions

# Buffer Manager

- Many DBs are much larger than available memory on servers
  - 1 PB data can't fit in a 100 GBs memory
  - DB data resides on disk
  - The block must be in memory to operate upon
- When the Query Processor Engine wants a block, it asks the Buffer manager
- **Buffer pool**
  - Similar to OS virtual memory
  - Copy of disk blocks in memory
- **Buffer manager**
  - Similar to OS virtual memory manager
  - If block already in memory, return a pointer to it
  - If not:
    - Allocate space, evicting a current page
    - Either write it back to its original location, or
    - Just throw it away (if it was read from disk, and not modified)
    - Read data
    - Make a request to the storage subsystem to fetch it
    - Pass the address back

# Buffer Manager

- MAIN MEMORY
- DISK
- disk page
- free pages
- Page Requests from Higher Levels
- Buffer pool
- Choice of page dictated by **replacement policy**

# Buffer Manager

- Similar to virtual memory manager
- Buffer replacement policies
  - What page to evict?
  - LRU: Least Recently Used
    - Throw out the page that was not used in a long time
  - MRU: Most Recently Used
    - The opposite
    - E.g., during a join there is a nested loop, the block just used won't be used until the next iteration

**for each** tuple $i$ of *instructor* **do**
  **for each** tuple $d$ of *department* **do**
    **if** $i[dept\_name] = d[dept\_name]$
    **then begin**
      let $x$ be a tuple defined as follows:
      $x[ID] := i[ID]$
      $x[dept\_name] := i[dept\_name]$
      $x[name] := i[name]$
      $x[salary] := i[salary]$
      $x[building] := d[building]$

# Buffer Manager

- **Pinning a block**
  - There can be multiple concurrent processes
  - One block was just read and another query tries to evict it
  - One query pins / unpins the block to lock it while it's being used
- **Writing-ahead a block**
  - The DB can write the update block to disk before it's evicted
  - When the space of that block is needed (eviction), no need to wait the write to go through
- **Forced output of blocks**
  - If there is a crash, the buffer pool and the memory content are lost
  - It is necessary to write a block to disk to make sure the transaction was "committed"

# Disk-block Access

- DB systems (i.e., the query processing sub-system) generate requests for disk I/O
  - Data can be stored on OS files
  - Can specify directly logical block
- In case of direct block access techniques to reduce number of random accesses (especially for magnetic disks, less for SSD)
- Buffering
  - Data is read in memory waiting for future requests
- Read-ahead
  - Consecutive blocks are read from the same track to minimize seek time
- Scheduling
  - Read blocks as they pass under the head to minimize disk-arm movement
  - E.g., elevator algorithm
- File organization
  - If data is accessed sequentially keep all data in blocks on adjacent cylinders
- Defragmentation
  - Due to appends to sequential files, the data becomes scattered over the disk
  - Make a copy of the sequential data and delete the old one
- Non-volatile write buffers
  - Save data in non-volatile RAM to speed-up writes and survive system

# Outline

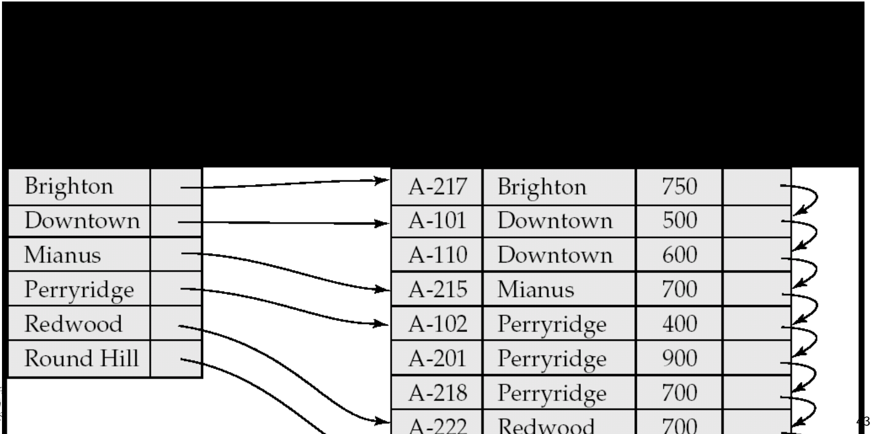
- Storage
  - Physical storage
  - Logical storage
    - File Organization
    - Buffer Manager
    - **Indexes**
- Query Processing
- Transactions

# Index

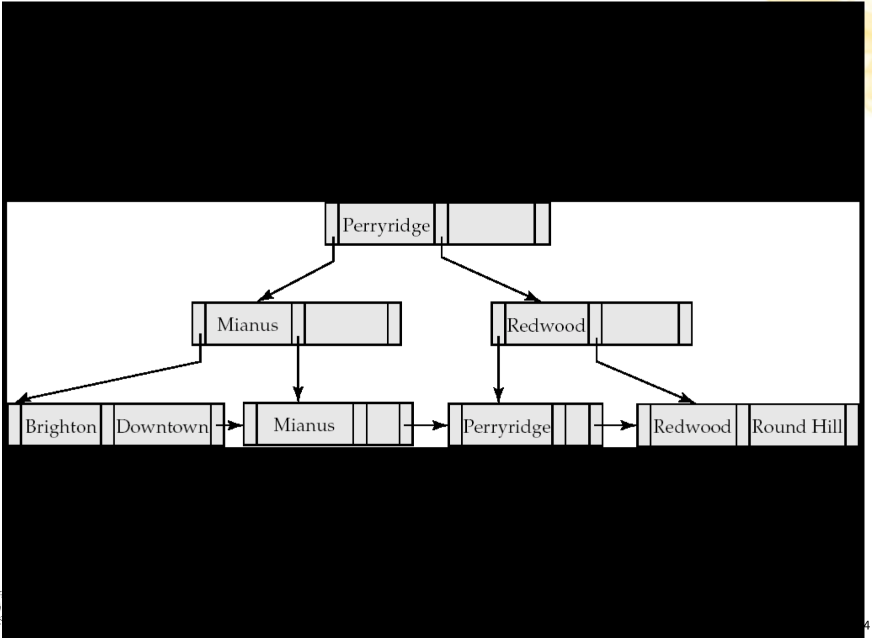
- **Index** = a data structure for efficient search through large databases
- **Search key**
  - Attribute or set of attributes used to look up records
  - E.g. SSN for a *person* table
- Key ideas:
  - Records are mapped to the disk blocks in specific ways
    - Heap, sorted, or hash-based
  - Auxiliary data structures maintained that allow quick search and access
  - Think book index, library catalogue
- **Ordered indexes**
  - Another data structure with search key sorted and pointers to the corresponding records
- **Hash-based indexes**
  - Locate the index
  - Order is not preserved
  - Collisions

# Ordered Indexes

- Map from key to corresponding record / row
- Primary index
  - Key is sorted to allow binary search
  - Can have only one primary index on a relation
- Secondary index
  - Key is not sorted

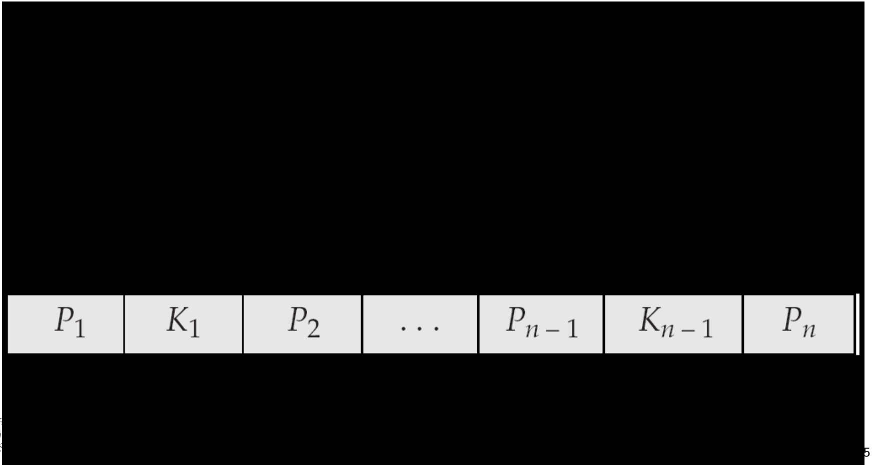


# Example B+- Tree Index



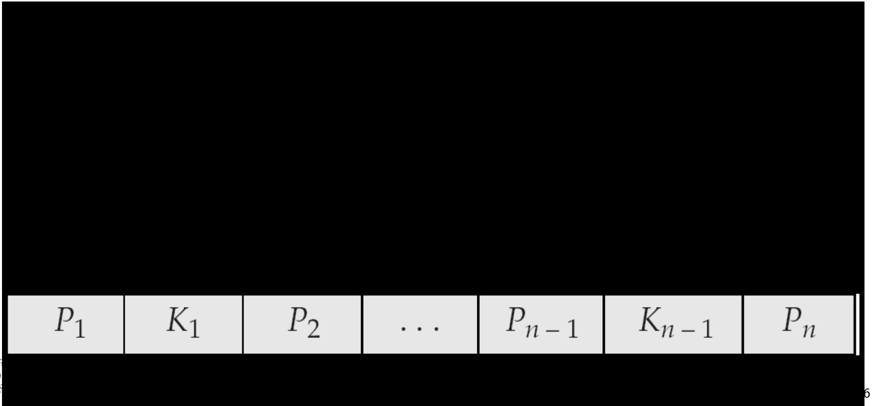
# B+-Tree Node Structure

- Typical node
  - Ki are the search-key values
  - Pi are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered K1 < K2 < K3 < . . . < Kn–1

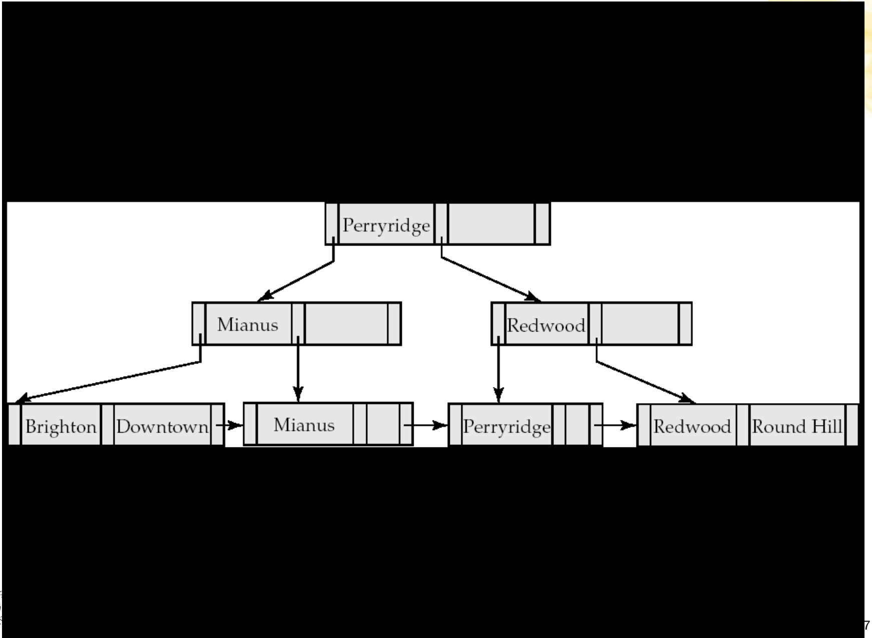


# Properties of B+-Trees

- It is balanced
  - Every path from the root to a leaf is same length
- Leaf nodes (at the bottom)
  - $P1$ contains the pointers to tuple(s) with key $K1$
  - ...
  - $Pn$ is a pointer to the *next* leaf node
  - Must contain at least n/2 entries

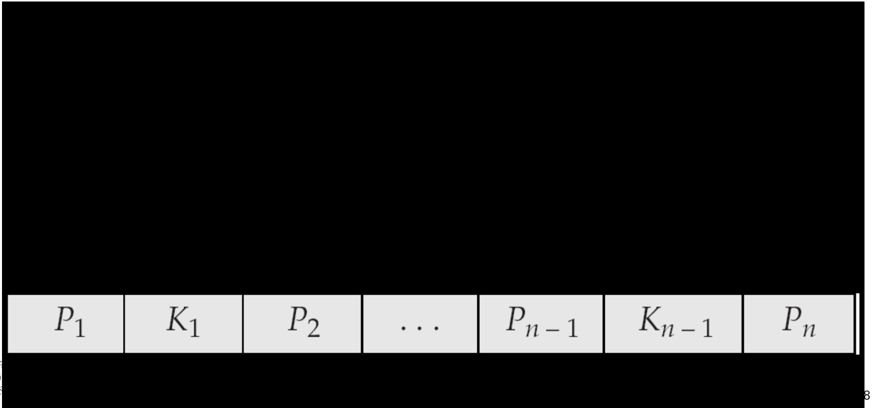


# Example B+-Tree Index

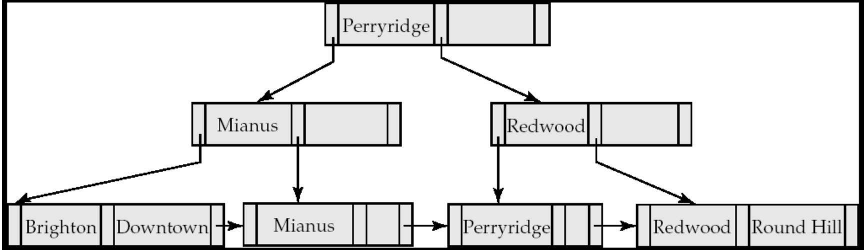


# Properties

- Interior nodes
  - All tuples in the subtree pointed to by *P*1, have search key $<$ *K*1
  - To find a tuple with key *K*1' $<$ K1, follow *P*1
  - ...
  - Finally, search keys in the tuples contained in the subtree pointed to by *P*n, are all larger than *K*n-1
  - Must contain at least n/2 entries (unless root)



# Example B+-Tree Index



# Outline

- Storage
- Query Processing
    - Selection operation
    - Join operators
    - Sorting
    - Other operators
    - Putting it all together...
- Transactions

# Overview

# "Cost"

- Complicated to compute
- We will focus on disk:
  - Number of I/Os?
    - Not sufficient
    - Number of seeks matters a lot... why ?
  - *t**T* – time to transfer one block
  - *t**S* – time for one seek
  - Cost for *b* block transfers plus *S* seeks *b*  t**T* + S * t**S*
  - Measured in *seconds*

SCIENCE ACADEMY

# Outline

- Storage
- Query Processing
  - Selection operation
  - Join operators
  - Sorting
  - Other operators
  - Putting it all together...
- Transactions

# Selection Operation

**SELECT** * **FROM** person **WHERE** SSN = "123" - **Option 1: Sequential Scan** - Read the relation start to end and look for "123" - Can always be used (not true for the other options) - Cost - *Let br* ***= Number of relation blocks - Then: - 1 seek and br block transfers - So: - t**S* + *b**r* * *t**T *sec - Improvements: - If SSN is a key, then can stop when found* - *So on average, b**r/2 *blocks accessed*

SCIENCE ACADEMY

# Selection Operation

**SELECT** * **FROM** person **WHERE** SSN = "123" - **Option 2 : Binary Search** - Pre-condition: - The relation is sorted on SSN - Selection condition is an equality - E.g. can't apply to "*Name like '%424%'*" - Do binary search - Cost of finding the *first* tuple that matches - [log2(*b**r*)]* (*t**T* + *t**S*) - All I/Os are random, so need a seek for all - The last few are closeby, but we ignore such small effects - Not quite: what if 10,000 tuples match the condition? - Incurs additional cost

# Selection Operation

**SELECT** * **FROM** person **WHERE** SSN = “123” - **Option 3 : Use Index** - Pre-condition: - An appropriate index must exist - Use the index - Find the first leaf page that contains the search key - Retrieve all the tuples that match by following the pointers - If primary index, the relation is sorted by the search key - Go to the relation and read blocks sequentially - If secondary index, must follow all pointers using the index

# Query Processing

- Overview
- Selection operation
- Join operators
- Sorting
- Other operators
- Putting it all together...

# Join

- **SELECT** * **FROM** R, S **WHERE** R.a = S.a
  - Called an "*equi-join*"
- **SELECT** * **FROM** R, S **WHERE** $|R.a - S.a| < 0.5$
  - *Not an "equi-join"*
- Option 1: Nested-loops *for each tuple r in R for each tuple s in S check if* r.a = s.a (or whether $|r.a - s.a| < 0.5$)
- Can be used for any join condition
  - As opposed to some algorithms we will see later
- R called *outer relation*
- S called *inner relation*

SCIENCE ACADEMY

# Nested-loops Join

- Cost? Depends on the actual values of parameters, especially memory
- *b*r, b*s* -> Number of blocks of R and S
- *n*r, n*s* -> Number of tuples of R and S
- Case 1: Minimum memory required = 3 blocks
  - One to hold the current *R* block, one for current S block, one for the result being produced
  - Blocks transferred:
    - Must scan *R* tuples once: *b*r
    - For each *R* tuple, must scan *S: n*r * b*s
  - Seeks?
    - *n*r + b*r

SCIENCE ACADEMY

# Nested-loops Join

- Case 1: Minimum memory required = 3 blocks
  - Blocks transferred: $n_r * b_s + b_r$
  - Seeks: $n_r + b_r$
- Example:
  - Number of records – $R$: $n_r = 10{,}000$, S: $n_s = 5000$
  - Number of blocks – $R$: $b_r = 400$ , S: $b_s = 100$
- Then:
  - blocks transferred: 10000 * 100 + 400 = 1,000,400
  - seeks: 10400
- What if we were to switch R and S ?
  - 2,000,100 block transfers, 5100 seeks
- Matters

# Nested-loops Join

- Case 2: *S* fits in memory
  - Blocks transferred: $b$s + b*r
  - Seeks: *2*
- Example:
  - Number of records – *R: n*r = 10,000, S: n*s = 5000
  - Number of blocks – *R: b*r = 400 , S: b*s = 100
- Then:
  - blocks transferred: 400 + 100 = 500
  - seeks: 2
- This is orders of magnitude difference

# Hash Join

- Case 1: Smaller relation *(S)* fits in memory
- Nested-loops join: *for each tuple r in R for each tuple s in S check if* r.a = s.a
- Cost: *b*r + b*s transfers, 2 seeks
- The inner loop is not exactly cheap (high CPU cost)
- Hash join: *read S in memory and build a hash index on it for each tuple r in R* use the hash index on S to find tuples such that S.a = r.a

# Hash Join

- Case 1: Smaller relation *(S)* fits in memory
- Hash join: *read $S$ in memory and build a hash index on it for each tuple $r$ in $R$ use the hash index on $S$ to find tuples such that $S.a = r.a$*
- Cost: $br$ + b*s transfers, 2 seeks (unchanged)
- Why good ?
  - CPU cost is much better (even though we don't care about it too much)
  - Performs much better than nested-loops join when $S$ doesn't fit in memory (next)

# Hash Join

- Case 2: Smaller relation *(S)* doesn't fit in memory
- Two "phases"
- Phase 1:
  - Read the relation *R* block by block and partition it using a hash function, *h1(a)*
    - Create one partition for each possible value of *h1(a)*
  - Write the partitions to disk
    - R gets partitioned into R1, R2, . . . , Rk
  - Similarly, read and partition S, and write partitions S1, S2, . . . , Sk to disk
  - Only requirement:
    - Each S partition fits in memory

# Hash Join

- Case 2: Smaller relation *(S)* doesn't fit in memory
- Two "phases"
- Phase 2:
  - Read S1 into memory, and build a hash index on it (S1 fits in memory)
    - Using a different hash function, *h2(a)*
  - Read R1 block by block, and use the hash index to find matches.
  - Repeat for S2, R2, and so on.

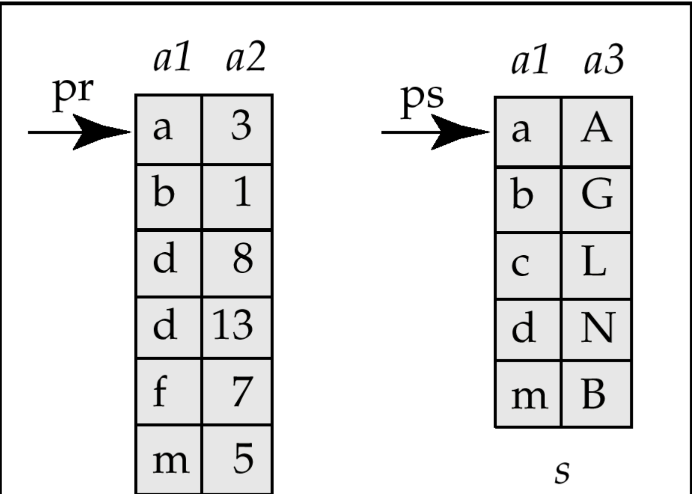
# Hash Join

- Case 2: Smaller relation *(S)* doesn't fit in memory
- Two "phases":
- Phase 1:
  - Partition the relations using one hash function, $h1(a)$
- Phase 2:
  - Read Si into memory, and build a hash index on it (Si fits in memory)
  - Read Ri block by block, and use the hash index to find matches.
- Cost?
  - $3(br + bs) + 4 * nh$ block transfers $+ 2(\lceil br / bb \rceil + \lceil bs / bb \rceil)$ seeks
    - Where $bb$ is the size of each output buffer
  - Much better than Nested-loops join under the same conditions*

# Hash Join

# Merge-Join (Sort-merge join)

- Pre-condition:
  - The relations must be sorted by the join attribute
  - If not sorted, can sort first, and then use this algorithms
- Called "sort-merge join" sometimes



# Merge-Join (Sort-merge join)

- Cost:
  - If the relations sorted, then just
    - br + bs block transfers, some seeks depending on memory size
  - What if not sorted?
    - Then sort the relations first
    - In many cases, still very good performance
    - Typically comparable to hash join
- Observation:
  - The final join result will also be sorted on *a1*
  - This might make further operations easier to do
    - E.g., duplicate elimination

SCIENCE ACADEMY

# Joins: Summary

- Block Nested-loops join
  - Can always be applied irrespective of the join condition
- Index Nested-loops join
  - Only applies if an appropriate index exists
- Hash joins – only for equi-joins
  - Join algorithm of choice when the relations are large
- Hybrid hash join
  - An optimization on hash join that is always implemented
- Sort-merge join
  - Very commonly used – especially since relations are typically sorted
  - Sorted results commonly desired at the output
    - To answer group by queries, for duplicate elimination, because of ASC/DSC

# Query Processing

- Overview
- Selection operation
- Join operators
- Sorting
- Other operators
- Putting it all together...

# Sorting

- Commonly required for many operations
  - Duplicate elimination, group by's, sort-merge join
  - Queries may have ASC or DSC in the query
- One option:
  - Read the lowest level of the index
    - May be enough in many cases
  - But if relation not sorted, this leads to too many random accesses
- If relation small enough...
  - Read in memory, use quick sort (qsort() in C)
- What if relation too large to fit in memory ?
  - External sort-merge

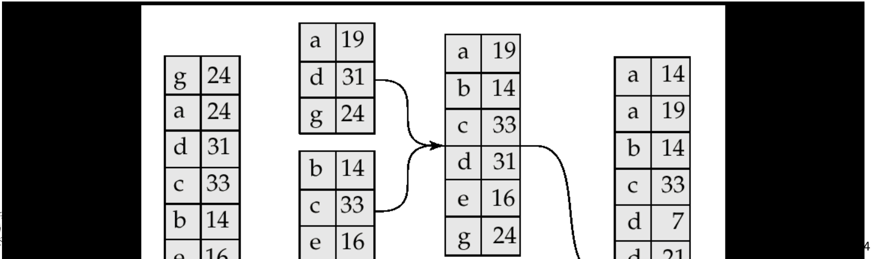
# External sort-merge

- Divide and Conquer !!
- Let $M$ denote the memory size (in blocks)
- Phase 1:
  - Read first M blocks of relation, sort, and write it to disk
  - Read the next M blocks, sort, and write to disk . . .
  - Say we have to do this "N" times
  - Result: $N$ sorted runs of size $M$ blocks each
- Phase 2:
  - Merge the $N$ runs (*$N$-way merge*)
  - Can do it in one shot if $N < M$

# External sort-merge

- Phase 1:
  - Create *sorted runs of size M* each
  - Result: $N$ sorted runs of size $M$ blocks each
- Phase 2:
  - Merge the $N$ runs (*N-way merge*)
  - Can do it in one shot if $N < M$
- What if $N > M$ ?
  - Do it recursively
  - Not expected to happen
  - If $M = 1000$ blocks $= 4$MB (assuming blocks of 4KB each)
    - Can sort: 4000MB = 4GB of data

*Example: External Sorting Using Sort-Merge



# External Merge Sort (Cont.)

- Cost analysis:
  - Total number of merge passes required:
    [log*M*-1(*b**r*/M)]
  - Disk accesses for initial run creation as well as in each pass is "' 2*b**r* "'
    - for final pass, we don't count write cost
    - we ignore final write cost for all operations since the output of an operation may be sent to the parent operation without being written to disk
  - Thus total number of disk accesses for external sorting: "' text *b**r* ( 2 [log$M$–1(*b**r* / M)] + 1) "'

# Query Processing

- Overview
- Selection operation
- Join operators
- Sorting
- Other operators
- Putting it all together...

# Group By and Aggregation

*select a, count(b) from R group by a;* - Hash-based algorithm - Steps: - Create a hash table on *a*, and keep the *count(b) so far* - Read *R* tuples one by one - For a new *R tuple, "r"* - Check if *r.a* exists in the hash table - If yes, increment the count - If not, insert a new value

# Group By and Aggregation

*select a, count(b) from R group by a;* - Sort-based algorithm - Steps: - Sort *R* on *a* - Now all tuples in a single group are contigous - Read tuples of *R* *(sorted)* one by one and compute the aggregates

# Group By and Aggregation

*select a, AGGR(b) from R group by a;* - sum(), count(), min(), max(): only need to maintain one value per group - Called "distributive" - average() : need to maintain the "sum" and "count" per group - Called "algebraic" - stddev(): algebraic, but need to maintain some more state - median(): can do efficiently with sort, but need two passes (called "holistic") - First to find the number of tuples in each group, and then to find the median tuple in each group - count(distinct b): must do duplicate elimination before the count

# Duplicate Elimination

# *select distinct a*

*from R ;* - Best done using sorting – Can also be done using hashing - Steps: - Sort the relation $R$ - Read tuples of $R$ in sorted order - prev = null; - for each tuple $r$ in $R$ *(sorted)* - if $r$ != $prev$ then - Output $r$ - $prev = r$ - else - Skip $r$

SCIENCE ACADEMY

# Set operations

*(select* from R) union (select * from S) ; (select * from R) intersect (select * from S) ; (select * from R) union all (select * from S) ; (select * from R) intersect all (select * from S) ; - *Remember the rules about duplicates - "union all": just append the tuples of* R and S - "*union": append the tuples of* R and S, and do duplicate elimination - "intersection"*: similar to joins - Find tuples of R and S that are identical on all attributes - Can use* hash-based or sort-based algorithm*

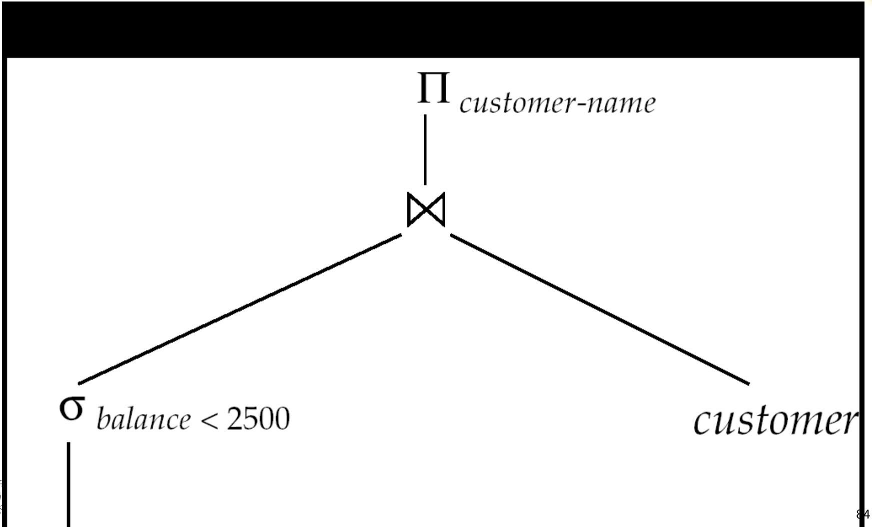
SCIENCE ACADEMY

# Query Processing

- Overview
- Selection operation
- Join operators
- Sorting
- Other operators
- Putting it all together...

# Evaluation of Expressions

- Two options:
  - Materialization
  - Pipelining



# Evaluation of Expressions

- Materialization
  - Evaluate each expression separately
  - Store its result on disk in *temporary relations*
  - Read it for next operation
- Pipelining
  - Evaluate multiple operators simultaneously
  - Skip the step of going to disk
  - Usually faster, but requires more memory
  - Also not always possible..
    - E.g., Sort-Merge Join
  - Harder to reason about

# Materialization

- Materialized evaluation is always applicable
- Cost of writing results to disk and reading them back can be quite high
    - Our cost formulas for operations ignore cost of writing results to disk, so
        - Overall cost = Sum of costs of individual operations + cost of writing intermediate results to disk
- Double buffering: use two output buffers for each operation, when one is full write it to disk, while the other is getting filled
    - Allows overlap of disk writes with computation and reduces execution time

# Pipelining

- Evaluate several operations simultaneously, passing the results of one operation on to the next.
- E.g., in previous expression tree, don't store result of
  - instead, pass tuples directly to the join.. Similarly, don't store result of join, pass tuples directly to projection.
- Much cheaper: no need to store a temporary relation to disk.
- Requires higher amount of memory
  - All operations are executing at the same time (say as processes)
- Somewhat limited applicability
- A "blocking" operation: An operation that has to consume entire input before it starts producing output tuples

$$\sigma_{balance<2500}(account)$$

SCIENCE ACADEMY

# Pipelining

- Need operators that generate output tuples while receiving tuples from their inputs
  - Selection: Usually yes.
  - Sort: NO. The sort operation is blocking
  - Sort-merge join: The final (merge) phase can be pipelined
  - Hash join: The partitioning phase is blocking; the second phase can be pipelined
  - Aggregates: Typically no. Need to wait for the entire input before producing output
    - However, there are tricks you can play here
  - Duplicate elimination: Since it requires sort, the final merge phase could be pipelined
  - Set operations: see duplicate elimination

# Transactions

- Storage
- Query Processing
- Transactions
    - Overview
    - Concurrency Control
    - Recovery

# Overview

- *Transaction*: A sequence of database actions enclosed within special tags
- Properties:
  - **Atomicity**: Entire transaction or nothing
  - **Consistency**: Transaction, executed completely, takes database from one consistent state to another
  - **Isolation**: Concurrent transactions *appear* to run in isolation
  - **Durability**: Effects of committed transactions are not lost
- Consistency: transaction programmer needs to guarantee that
  - DBMS can do a few things, e.g., enforce constraints on the data
- Rest: DBMS guarantees

# Bank application example

- TODO

# How does..

- .. this relate to *queries* that we discussed ?
  - Actual queries don't update data, so *durability* and *consistency* not relevant
  - Would want *concurrency*
    - Consider a query computing total balance at the end of the day
  - Would want *isolation*
    - What if somebody makes a *transfer* while we are computing the balance
    - Typically not guaranteed for such long-running queries

# Performance benchmark

- Transaction per minute
- TPC-C:
  - OLTP application
  - a wholesale distributor with a small number of warehouses full of inventory servicing a larger number of retail locations
  - Heady on disk I/O
- TPC-E:
  - modern OLTP application
  - a stock brokerage, simulated world driven by fluctuating stock prices and outside world of customers placing market orders, limit orders and stop-limit orders
  - Heavy on RAM
- TPC-H
  - OLAP application
  - Query analytics in a ‘data warehouse’ context

# Assumptions and Goals

- Assumptions:
  - The system can crash at any time
  - Similarly, the power can go out at any point
    - Contents of the main memory won't survive a crash, or power outage
  - BUT... **disks are durable. They might stop, but data is not lost.**
  - Disks only guarantee *atomic sector writes*, nothing more
  - Transactions are by themselves consistent
- Goals:
  - Guaranteed durability, atomicity
  - As much concurrency as possible, while not compromising isolation and/or consistency
    - Two transactions updating the same account balance... NO
    - Two transactions updating different account balances... YES

# Transactions

- Overview
- Concurrency Control
- Recovery

# Lock-based Protocols

- A transaction *must* get a *lock* before operating on the data
- Two types of locks:
  - *Shared* (S) locks (also called *read locks*)
    - Obtained if we want to only read an item
  - *Exclusive* (X) locks (also called *write locks*)
    - Obtained for updating a data item

# Lock instructions

- New instructions
  - lock-S: shared lock request
  - lock-X: exclusive lock request
  - unlock: release previously held lock Example schedule:

# Lock instructions

- New instructions
  - lock-S: shared lock request
  - lock-X: exclusive lock request
  - unlock: release previously held lock Example schedule:

# Lock-based Protocols

- Lock requests are made to the *concurrency control manager*
  - It decides whether to *grant* a lock request
- T1 asks for a lock on data item A, and T2 currently has a lock on it?
  - Depends
- If *compatible*, grant the lock, otherwise T1 waits in a *queue*.

| T2 lock type | T1 lock type | Should allow ? |
|---|---|---|
| Shared | Shared | YES |
| Shared | Exclusive | NO |
| Exclusive | - | NO |

# Snapshot Isolation

- Very popular scheme, used as the primary scheme by many systems including Oracle, PostgreSQL etc...
  - Several others support this in addition to locking-based protocol
- A type of "optimistic concurrency control"
- Key idea:
  - For each object, maintain past "versions" of the data along with timestamps
    - Every update to an object causes a new version to be generated

# Snapshot Isolation

- Read queries:
  - Let "t" be the "time-stamp" of the query, i.e., the time at which it entered the system
  - When the query asks for a data item, provide a version of the data item that was latest as of "t"
    - Even if the data changed in between, provide an old version
  - No locks needed, no waiting for any other transactions or queries
  - The query executes on a consistent snapshot of the database
- Update queries (transactions):
  - Reads processed as above on a snapshot
  - Writes are done in private storage
  - At commit time, for each object that was written, check if some other transaction updated the data item since this transaction started
    - If yes, then abort and restart
    - If no, make all the writes public simultaneously (by making new versions)

# Snapshot Isolation

- Advantages:
  - Read query don't block at all, and run very fast
  - As long as conflicts are rare, update transactions don't abort either
  - Overall better performance than locking-based protocols
- Major disadvantage:
  - Not serializable
  - Inconsistencies may be introduced
  - See the wikipedia article for more details and an example

# Transactions

- Overview
- Concurrency Control
- Recovery

# Context

- ACID properties:
  - We have talked about Isolation and Consistency
  - How do we guarantee Atomicity and Durability?
    - Atomicity: Two problems
    - Part of the transaction is done, but we want to cancel it: ABORT/ROLLBACK
    - System crashes during the transaction. Some changes made it to the disk, some didn't.
    - Durability:
    - Transaction finished. User notified. But changes not sent to disk yet (for performance reasons). System crashed.
- Essentially similar solutions

# Reasons for crashes

- Transaction failures
  - **Logical errors**: transaction cannot complete due to some internal error condition
  - **System errors**: the database system must terminate an active transaction due to an error condition (e.g., deadlock)
- System crash
  - Power failures, operating system bugs, etc.
  - **Fail-stop assumption**: non-volatile storage contents are assumed to not be corrupted by system crash
    - Database systems have numerous integrity checks to prevent corruption of disk data
- Disk failure
  - Head crashes; **we will assume**
    - **STABLE STORAGE: Data never lost. Can approximate by using RAID and maintaining geographically distant copies of the data**

# Log-based Recovery

- Most commonly used recovery method
- Intuitively, a log is a record of everything the database system does
- For every operation done by the database, a *log record* is generated and stored *typically on a different (log) disk*
- *<T1, START>*
- *<T2, COMMIT>*
- *<T2, ABORT>*
- *<T1, A, 100, 200>*
  - T1 modified A; old value = 100, new value = 200