CS3223 Cheatsheet

Intro to Database Management

DB Requirements – Persistence, efficiency, high-level access(SQL), transaction management (allow concurrent access), access control (permissions), Integrity management (comply to constraints of DB), Resiliency (recover from system failures).

Disk, Memories, Buffer Management

- DBMS stores relations (actual data), indexes, metadata(relation schemas → structure of relations/constraints/triggers, view definitions, statistical info about relations for query optimization, index metadata), log files

→ Currently used data stored in primary memory (RAM), main database stored in disk

Components of Disk

Components of a Disk

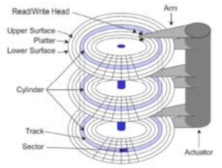
The platters spin (say, 120rps)

The arm assembly is moved in or out to position a read/write head on a desired track. Tracks under the head make a (imaginary) *cylinder*

Only one head reads/writes at any one time

Block size is a multiple of *sector size* (which is fixed)

CS3223 - Storage



→ Read/write head to read information under track. Tracks under the head (Including those in the other platters) is a cylinder.

→ 1 platter surface has many tracks, 1 track has many sectors, 1 cylinder can also have multiple tracks (11 tracks in 1 cylinder means 11 platters I think), 1 sector has many bytes. **One page/block can consist of 2/4 sectors**

Accessing a Disk Page

Seek Time + Rotational Delay + Transfer Time

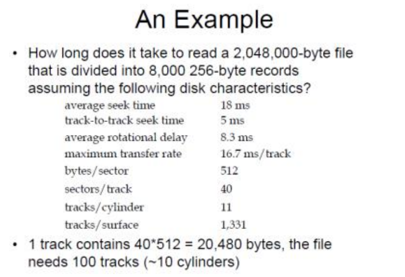
Seek Time – Moving arms to position disk head on track

Rotational Delay – Waiting for block to rotate under head

Transfer Time – Transferring data from disk

→ Seek time and rotational delay dominates (Seek time ~0.3-10ms, rot. Delay 0 to 4ms, transfer rate about 0.05ms/8kb)

→ Reduce I/O cost by reducing seek/rotational delays



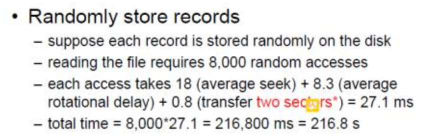
An Example

- How long does it take to read a 2,048,000-byte file that is divided into 8,000 256-byte records assuming the following disk characteristics?

average seek time	18 ms
track-to-track seek time	5 ms
average rotational delay	8.3 ms
maximum transfer rate	16.7 ms/track
bytes/sector	512
sectors/track	40
tracks/cylinder	11
tracks/surface	1,331

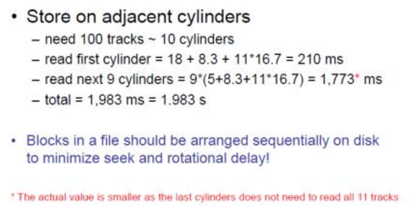
- 1 track contains 40*512 = 20,480 bytes, the file needs 100 tracks (~10 cylinders)

→ If random access, and each page is 2 sectors, then each access will incur full seek+rot delay+transfer_time(2 pages)



- Randomly store records
  - suppose each record is stored randomly on the disk
  - reading the file requires 8,000 random accesses
  - each access takes 18 (average seek) + 8.3 (average rotational delay) + 0.8 (transfer two sectors*) = 27.1 ms
  - total time = 8,000*27.1 = 216,800 ms = 216.8 s

→ If is stored on adjacent cylinders, then we first calculate the time taken to retrieve 1 entire cylinder (first cylinder got longer seek+rot delay). Take note for the next few cylinders there is still a rot delay because we need to find the starting sector/page



- Store on adjacent cylinders
  - need 100 tracks ~ 10 cylinders
  - read first cylinder = 18 + 8.3 + 11*16.7 = 210 ms
  - read next 9 cylinders = 9*(5+8.3+11*16.7) = 1,773* ms
  - total = 1,983 ms = 1.983 s
- Blocks in a file should be arranged sequentially on disk to minimize seek and rotational delay!

* The actual value is smaller as the last cylinders does not need to read all 11 tracks

→ This one is an estimation, actually the last cylinder we only reading 1 track for this qn

Disk Space Management

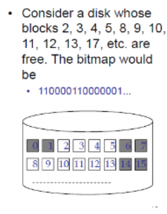
Requirements – efficient utilization of disk space, fast access

→ Use bitmaps/linked list to maintain free space list

→ Try to allocate free space based contiguously

Managing Free Space: Bitmap

Each block represented by 1 bit, if free block, corresponding bit is 0, else it is 1. Scan the map for 0s to allocate free space



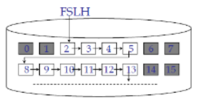
- Consider a disk whose blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, etc. are free. The bitmap would be
  - 110000110000001...

Managing Free Space: Linked List

→ Link all free disk blocks together, each free block points to next free block

→ DBMS maintain the free space list head (FSLH) to first free block.

→ To allocate space, just look up the FSLH, follow the pointers. Once allocated, reset the FSLH to the head.



Allocation of Free Space Considerations

- **Granularity**

→ Pages vs blocks (multiple consecutive pages) vs extents (multiple consecutive blocks), if smaller granularity we get external fragmentation, large granularity leads to lower space utilization and is good as file grows in size.

→ We mainly deal with pages in this mod

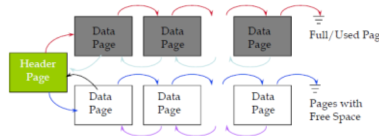
- **Allocation Methods**

→ Contiguous: All pages/blocks/extents close by. To achieve this, we might need to reclaim/rearrange space frequently

→ Linked List: Simple, but may be fragmented

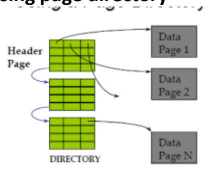
Managing Space Allocated to Files: Heap (unordered) file implemented as list

- Header page id and heap file must be stored at some place (database metadata/catalog)



→ Store one chain of free space, one chain of used pages

Heap File using page directory



→ Entry for page can include number of free bytes on the page (aside of the data page itself).

→ Directory can be implemented via linkedlist, and is much smaller than an LL of all the heap file pages

→ 1 header page size same as 1 data page size, can store metadata

Buffer Management in DBMS

- Buffer pool is partitioned into pages called frames

- DBMS maintains table of <frame#, pageid>, each frame has pin count (how many queries processing this page atm) and dirty flag

When a page is requested..

→ If requested page not in buffer pool, try to read requested page into chosen frame. If no free frames we will have to perform replacement. If the replaced frame is dirty, we must write the changes to disk. If no pages can be replaced atm, we just wait until there is

→ We increase pin count and return the address of the new page in RAM.

→ Cost to access page? To request a page in requires 1 i/o. If a dirty page is being replaced then it requires 1 more i/o to write out changes

→ We can make use of sequential scans to fetch several pages at a time, so that not every page that is needed will need 1 new i/o

Replacement Policies:

FIFO: Replace oldest buffer page (based on first reference). Good only for sequential access behaviour

LFU (Least freq used): Replace buffer page with lowest reference freq – pages with high reference activity in a short interval may never be replaced

LRU (Least Recently used): Replace buffer page that is least recently used (last reference) – worst policy when sequential flooding occurs

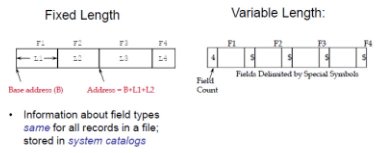
Files of Records

File: Collection of pages, each containing collection of records.

→ Must support CRUD to record, read particular record by id, scan all records

Record Format

Fixed Length / Variable Length.



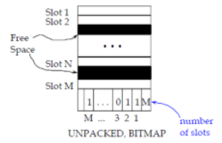
- Information about field types *same* for all records in a file; stored in *system catalogs*

→ Fixed: If we predefine data type to be string, need to allocate maximum string length, from there we can get the entire record length, might waste space when actual value is less than allocated space, but it is simpler, easily calculate where is the nth record

→ Variable Length: We can say how many fields are there in this record (field count). Delimit each field with special symbol.

Page Format: Fixed Length Records

→ To store where each record is, we keep track of a recordId = <page id, slot#>. This works because we know what size each record is, so we can just do a calculation to find get to the contents of the record.

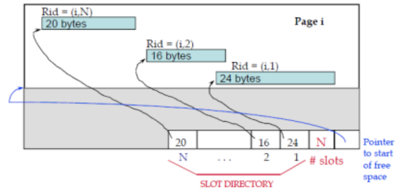


Page Formats: Variable Length Record

→ Have a slot directory at the end of the page that is constantly growing (cannot be fixed because the number of records in 1 page not fixed)

→ Slot directory has pointer to start of each record + size of each record, also tells us number of slots in this page

→ Records also have signature <page id, slot#>



Summary

- Effective buffer management is very important in DBMS, disk access are the most expensive operations. DBMS needs to manage its own buffer because it will know how to remove pages better than the OS. E.g. For SELECT *, a page replacement policy like most recently used is actually best, since once you read a page you can flush it out, but an OS would not know that.

Indexes

→ Motivation for index: queries on normal table can take very long because we have to perform sequential reads from the start of the files, also very slow to search condition based on some other attribute that is not the primary key

→ Index is a data structure that speeds up retrieval/selection based on some search key

→ Any subset of fields of a relation can be the search key for an index

→ Search key != primary key. Any set of attributes can be search key I think?

→ Can have multiple indices per relation on different attributes

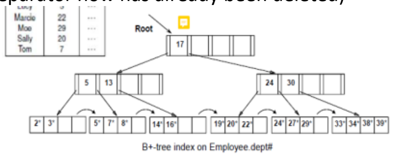
→ If its search key is a candidate key, then the index is a **unique index**, else it is non-unique

→ Index is stored as a file, records referred to as data entries

→ May not always perform better than linear scan

Index Types
→ Tree-based or hash-based
→ Considerations? Equality search vs range search performance on each type of indexes, storage overhead, update performance

B+ Tree Index
→ Supports range search and equality search
→ Leaf nodes are sorted data entries denoted by (search key value, RID)
→ Leaf nodes singly/doubly linked, mostly singly linked, advantage of doubly linked is that you can retrieve record in range queries backwards, but will complicate algorithm a lot
→ **Dense indices** have an one entry for each data record in the leaf nodes
→ Efficient for range search
→ internal nodes may not correspond to any key values (can happen because the key that is a separator now has already been deleted)



→ Each node in a tree is essentially a page, costs an i/o per node if not in memory

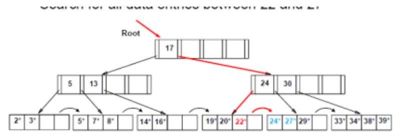
Determining Order of B+ Tree

- Assume 4 KB page, 8-byte key, 4-byte pointer, we have
  - m = 2*d
  - (2d+1)*4 + 2d*8 ≤ 4096
  - d ~ 170

→ d is order.

Properties of B+ Tree
- Height-balanced, so efficient for search/update. Insert/delete at $\log_F(N)$ (F = fanout, N = #leaf pages)
- Grow/shrink dynamically (update efficient)
- 50% occupancy except for root (storage efficient)
→ Every non leaf node contains d <= m <= 2d entries. D is order
→ Next leaf pointer in each leaf node (efficient range search)
→ Data entries in leaf are sorted

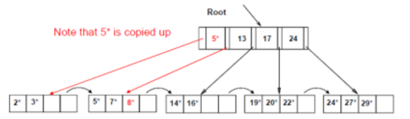
B+ Tree Search
- Start at root, at each internal node, find largest key in the internal node that is just smaller than the key I am searching. Once I find a node bigger than me, traverse the left pointer of that node.



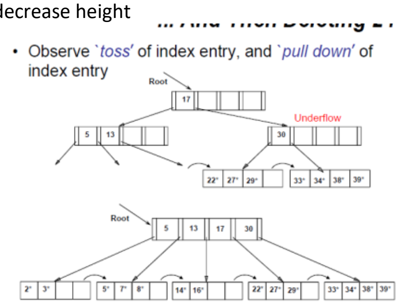
→ If we try to search for a key like 15, once we traverse down to 14, and iterate to the end of the leaf, we know for sure 15 is not in the database since our index is dense and the search key should definitely be in this page.
→ **Note:** Search keys within a leaf node don't need to be sorted. It might incur cpu cost to search through the entire leaf node, but at least it doesn't incur i/o cost (we alrd retrieved the entire page) so it is not that costly

B+ Trees In Practice
→ Typical fill-factor: 67%, (e.g. if d = 100, fanout = 133)
→ **Formats of Data Entries:** format 1 vs format 2 vs format 3
Format 1: k*: Leaf nodes store **actual data record**
Format 2 (**default in B+Trees**): (k, rid): Leaf nodes store record id which points to a data record of search key k
Format 3: (k, rid-list): list of record identifiers of data records with search key k

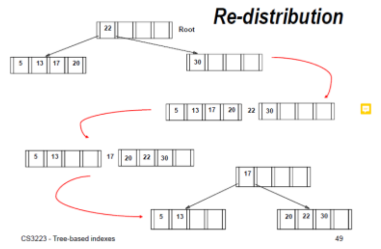
B+ Tree Insertion
Find correct leaf L, put data entry into L. If not enough space, **we need to split L into L and a new node L2**.
We split by taking the middle key (e.g. 3rd node of 5 nodes), copy up to previous level). We add a new pointer pointing from parent of L to L2.
→ We **copy up** middle key value from leaf node to parent node. But for internal nodes, the separator values are unique, so we **push up**.
→ Splitting process can happen recursively if parent nodes are also full!!
→ With splits, order grows, if root splits, height increases



B+ Tree Deletion
→ Recall B+ Tree guarantees that each internal node has at least order number of entries. So if we delete and it falls below that we need to perform redistribution.
- **Algorithm:** We start at root, find leaf L when entry belongs. We remove the entry and if the leaf L is still at least half-full, we are done. If it is less than half full (i.e. d-1 entries), we try to redistribute the records by borrowing from sibling (adjacent node with same parent as L). Take note that we might have to update the parent value of L so that only values strictly smaller than the parent value appears on the left subtree
If we cannot redistribute (because taking record from sibling will cause it to also become less than half full), then we merge L and the sibling together.
- If we merge at leaf level, we must delete entry (pointing to either L or the sibling) from the parent of L. In merging internal nodes, we pull down value from parent (parent becomes sibling)
- Merge can also propagate up to root and even decrease height


- Observe 'toss' of index entry, and 'pull down' of index entry

→ Redistribution of internal nodes is done by taking the centre element among all sibling values + parent value as the new parent value



Clustered vs unclustered index
- Index is clustered if the order of data entries same or close to order of data records, else unclustered

Sparse vs Dense Index
→ All data entry has a search key for dense. Every sparse index is clustered. Sparse indices cannot support exists queries
Summary
→ B+ tree – fast search for single and range queries, storage friendly, good performance

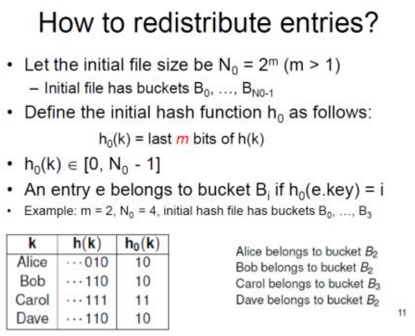
Hash-based Index
- Data stored in buckets
- h(k) returns bucket/page ID that stores record
- Best for equality search, **performance degenerate for skewed data distributions**
- Inefficient for range searches
- **Assume Format 1 for hash index**

Static Hashing
- Number of buckets do not change throughout entire lifetime of index. Data stored in M buckets, fixed at creation time
- One primary data page, chain of zero or more overflow data pages
- Allocated sequentially and never deallocated
- Long overflow chains – degrade performance! Can happen if data input into index is skewed, or that choice of hash function is poor

→ How to cope with growth? Use Linear Hash or extendible hash (dynamic hashes)

Linear Hash
- Hash file grows linearly (one bucket at a time)
→ After condition to split (e.g. current bucket full), we create a new bucket, called the split image of the current bucket Bj
→ File size doubles after one round of splitting

Redistributing entries



How to redistribute entries?

- Let the initial file size be $N_0 = 2^m$ (m > 1)
  - Initial file has buckets $B_0, \ldots, B_{N0-1}$
- Define the initial hash function h_0 as follows:

  $h_0(k)$ = last *m* bits of h(k)
- $h_0(k) \in [0, N_0 - 1]$
- An entry e belongs to bucket B_i if $h_0(e.key) = i$
- Example: m = 2, N_0 = 4, initial hash file has buckets $B_0, \ldots, B_3$

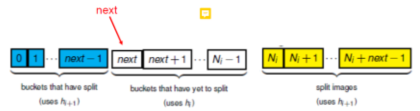
k	h(k)	h_0(k)
Alice	···010	10
Bob	···110	10
Carol	···111	11
Dave	···110	10

Alice belongs to bucket B_2
Bob belongs to bucket B_2
Carol belongs to bucket B_3
Dave belongs to bucket B_2

11

→ N0 = 4 means initially had 4 buckets. So the hash at first just needs to see last 2 bits to know which bucket it belongs to
→ (2+1) 3 bits will be needed to differentiate between split images

Modifying the hash function
→ We either use $h_i(k)$ or $h_{i+1}(k)$ to calculate the bucket it hashes to. Which do we use? We know that the buckets can be classified to 3 regions:



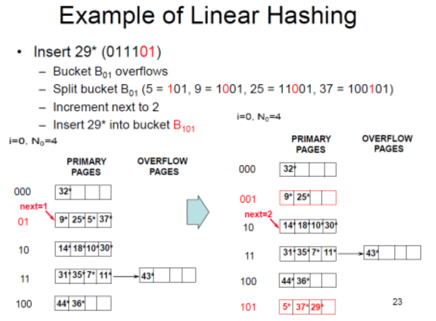
next
0
1
next − 1
next
next + 1
N_i − 1
N_i
N_i + 1
N_i + next − 1
buckets that have split (uses h_{i+1})
buckets that have yet to split (uses h_i)
split images (uses h_{i+1})

→ We first do $h_i(k)$, if we land into a bucket after next, then we know that it is for sure in this bucket, since it is a bucket that has yet to split. But if it lands into a bucket before next, then we need to do $h_{i+1}(k)$ to check if its in the current bucket or in the split image

When to split bucket?
→ Several possible criterias, e.g. when bucket utilization > 70%, or when some page overflows
→ Note: We will still have overflow pages in linear hash because the bucket that is split may not be that one that overflowed, but eventually it will be split (that's the idea of linear hash) → long overflow chains are not expected to be developed

- File grows one bucket at a time
  - Whenever a bucket is split, its records are redistributed using the last m+i+1 bits
  - Increment the *next* pointer

→ next pointer will directly point at the bucket to be split next



Example of Linear Hashing

- Insert 29* (011101)
  - Bucket B_{01} overflows
  - Split bucket B_{01} (5 = 101, 9 = 1001, 25 = 11001, 37 = 100101)
  - Increment next to 2
  - Insert 29* into bucket B_{101}

Linear Hashing: Deletion
- Locate bucket and delete entry, **if the last bucket becomes empty (only last?)**, it can be removed.
→ If next > 0, decrement next.
→ if next == 0 and i > 0 (h0 is the initial hash function), then we must update the next pointer to point to the last bucket in the previous level.
→ take note of this not to be careless. if currently next is at 0, and there are 4 buckets (0-3), means previously, next was at 1. so we move next back to 1

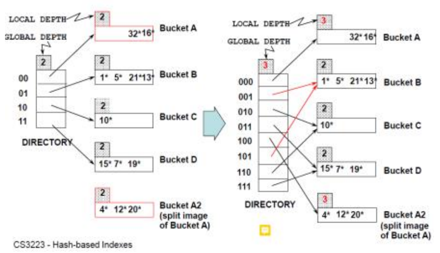
Linear Hashing: Performance
- 1 disk I/O unless bucket has overflow pages, one average 1.2 disk I/O (but go with 1 I/O unless stated).
- Worst Case: I/O is linear to number of data entries.
Motivations behind linear hash
- We don't want to simply double the file each time because doubling the file means that we need to at one shot, rehash every single element in our index. Also means poor space utilization

because there might be many buckets that do not require to be split but are already split.

Dynamic Hashing: Extensible Hashing
- Handle file growth by doubling # of buckets
- Have a directory of pointers to buckets, double # of directory entries, but only **split the bucket that overflows**
- Doubling directory much cheaper than bucket.
- No overflow pages

→ Use global depth to determine in directory how many bits to check to determine which bucket a value falls into
→ Local depth will tell us all keys in this bucket share the last x number of bits
→ look at directory first after calculating the hash value. We cannot go straight to the bucket to find the record like in Linear Hashing. So searching some value will take 2 I/O

Insert h(r)=20 (10100) – Causes Doubling



→ Directory entries double, number of pointers double, number of buckets increase linearly
→ For buckets that split, local depth++
→ When localdepth > global depth, then the number of directory entries double and global depth++

Extendible Hash: Insertion
- If bucket is full, we split
- If split bucket has local depth smaller than global depth, then we adjust the relevant directory entry to now point to the new bucket
- If the split bucket's local depth is equal to global depth, we need to double directory entries

Comments on Extendible Hashing
- Equality Search: 2I/O unless directory fits in memory
- Directory grows in spurts. If distribution of hash values is skewed the directory can grow very large!

Deletion: If removal of a data entry makes a bucket empty, we can merge it with split image.

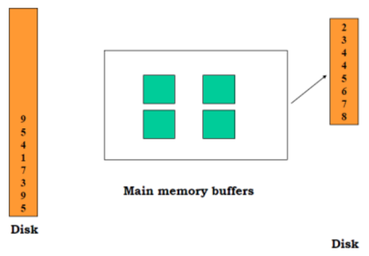
Once each directory element points to the same bucket as the split image, we can half the directory. (much harder than doubling because we need to check every local depth)

Summary of Hash Index
- Good for equality search, not so good for range search.
- Static hashing can lead to long overflow chains
- Dynamic hashing methods such as linear hashing and extendible hashing avoid overflow pages (but linear hash can still have)

External Sorting
→ Data requested in sorted order but we don't have enough buffer to perform in memory sort
→ Something like merging k-sorted list on LC
→ Minimum number of pages: 3 buffer pages, 2 to read in unsorted pages, 1 output buffer
→ Multiway Merge Sort → Phase 1: Generate sorted runs, Phase 2: Merge sorted runs

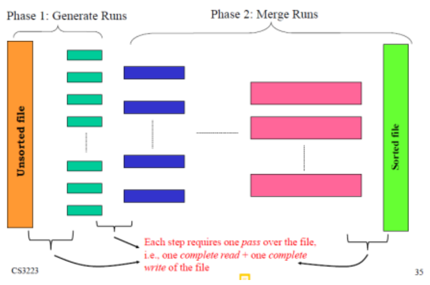
Generating Sorted Runs
→ Read in as many records into memory as possible, perform in-memory sort, write out sorted records as a sorted run.
→ Every page will be read in and out once per pass. Assuming B buffer pages, each time we read in B pages, sort the records and output one sorted run of B pages.
→ Number of sorted runs: ceil(N/B)
→ Total I/O cost: 2 * |num_pages|

How to generate sorted runs?



→ Write page into buffer, quicksort in memory, write out.

Merging Sorted Runs
→ Use B-1 buffer pages for input and 1 buffer for output
→ So we load in the smallest element from each of the B-1 sorted runs, then we iteratively pick the minimum, then when we have an empty input buffer (because its values have all been written out), we load in the page that is empty.
→ Each iteration takes 1 pass of read + write.



Phase 1: Generate Runs
Phase 2: Merge Runs
Unsorted file
Sorted file
Each step requires one pass over the file, i.e., one complete read + one complete write of the file
CS3223
35

Cost of External Sorting
→ Combining generation of sorted runs + merging sorted runs, total cost given by:
- Number of passes: $1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil$
- Cost = 2N * (# of passes)

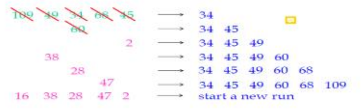
→ N is the number of pages
- E.g., with 5 buffer pages, to sort 108 page file:
  - Phase 1: $\lceil 108 / 5 \rceil$ = 22 sorted runs of 5 pages each (last run is only 3 pages)
  - Phase 2:
    - Pass 1: $\lceil 22 / 4 \rceil$ = 6 sorted runs of 20 pages each (last run is only 8 pages)
    - Pass 2: 2 sorted runs, 80 pages and 28 pages
    - Pass 3: Sorted file of 108 pages
  - Cost = (2*108) * 4 = 864

Replacement selection
→ While quicksort is a fast way to sort in memory, it generates quite a lot of runs especially if the buffer size is small compared to the total table size. What if we want to reduce the number of sorted runs (create a run that is larger than size of memory?)
→ Advantage? Less sorted runs means that we can merge with less passes → less I/O
→ replacement selection

Replacement Selection
→ Another way to create sorted runs
→ Idea: We only write out values to the current run that are bigger than the last number that we wrote out. E.g. If the last value we wrote out is 50, then anything less than 50 cannot be written out. If our entire buffer has values less than 50, then we create a new sorted run

Replacement Selection (Example)

- 109, 49, 34, 68, 45, 60, 2, 38, 28, 47, 16, 19, 35, 59, 98, 78, 76, 40, 35, 86, 10, 27, 61, 92



→ We hold n-1 numbers, last page to save previously inserted value. (Feels like you can do the sorting on this page then write out one entire page tho).

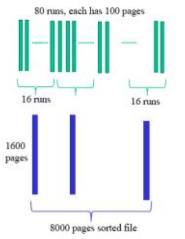
→ Everytime, we choose the minimum value and is larger than the prev inserted and write into disk. This value is our new prev.
→ Any smaller values than prev are frozen. When all values in the buffer are frozen, we start a new run.
→ So I guess the number of runs is quite unpredictable, would depend on the order of the elements in its unsorted state.
→ **Average Length of a run is 2B**. (Means half the number of runs, since num(runs) = num_pages / num_buffer)
→ Worst-case: The attributes are reverse sorted. Then every run will at be the size of the buffer (same as original sorted run strategy)
→ Best-case: If we have an **almost-sorted** run, we can end up with an entirely sorted run
→ Quicksort is faster, but longer runs means fewer passes → less I/O

Sequential vs Random I/O
→ Aside of minimizing passes, consider the type of I/O we are incurring.
Consider the following:
- Suppose we have 80 sorted runs, each 100 pages long and we have 81 pages of buffer space
- We can merge all 80 runs in a single (i.e., one) pass
  - Minimal number of passes!
- We can also merge the 80 runs in two passes
  - Pass 1: merge 16 runs first, resulting in 5 longer runs
  - Pass 2: merge the 5 runs
- Which is better??

→ If we read 5 pages into the buffer at each time, that is 5 sequential i/os which only incurs 1 seek per time plus 5 consecutive reads.
Compared to reading in 1 page into the buffer at each time, that is 1 random i/o and also incurs 1 seek (the most exp operation)
→ Let's compare the number of seeks in both scenarios:
- Suppose we have 80 sorted runs, each 100 pages long and we have 81 pages of buffer space
- We can merge all 80 runs in a single (i.e., one) pass
  - each page requires a seek to access (Why?)
  - there are 100 pages per run, so 100 seeks per run
  - total cost = 80 runs X 100 seeks = 8000 seeks

→ Usually what we would want to do is have 1 buffer page loaded into memory for each run we have. So in this case, each run will incur 100 random I/Os, so total 80runs * 100 = 8000 seeks

- We can merge all 80 runs in two steps
  - 5 sets of 16 runs each
    - read 80/16=5 pages of one run
    - 16 runs result in sorted run of 1600 pages
    - each merge requires 100/5X16 = 320 seeks
    - for 5 sets, we have 5X320 = 1600 seeks
  - merge 5 runs of 1600 pages
    - read 80/5=16 pages of one run
      => 1600/16=100 seeks in total
    - 5 runs => 5X100 = 500 seeks
  - total: 1600+500=2100 seeks!!!
- Number of passes increases, but number of seeks decreases!



→ Now we require 2 merges. First merge will merge 80 runs of 100 pages into 5 runs of 16 * 100 = 1600 pages. Then next merge will merge into 1 sorted run of 8000 pages.
→ For the first merge, each read is 5 sequential I/Os (1 seek + 5 reads), and for each of the 80 runs, we do that 20 times = 20 * 80 = 1600 seeks.
→ For the second merge, each read is 16 sequential I/Os (we got B=80, 5 sorted runs, so each run can load in 16 pages at once), for each run we do it 1600 / 16 = 100 times, 5 * 100 = 500 seeks.
→ Total: 2100 seeks

Even faster way to retrieve sorted records? Clustered B+ Tree
- Cost incurred is simply I/Os to traverse tree + retrieving leaf pages + retrieve data pages. If the B+ tree index is clustered, I think its confirm better than external sorting? (I would imagine that even if we sort and merge in 1 pass each, that would mean 4 * |num_pages| I/O. But for clustered B+ tree index, the most incurred I/O is at retrieving data pages, which is basically 1 * |num_pages| and everything else is less than that)

Unclustered B+ Tree for sorting
→ One I/O per data record. If we are retrieving a small range, it might be sensible to use it. But for large ranges its too costly

Number of runs at last pass of a multi-way merge
→ Calculate number of sorted runs at the start, keep diving by B-1 until the value is less than B-1. The ceiling is the number of runs it merges before we get one full run
→ To know how many pages are there in each run, recall that in merging, what happens is that we take B-1 runs and merge into 1 run. So if maybe each run has 30 pages, and B = 6, we are merging B – 1 = 5 runs, so that new run has 5 * 30 pages = 150 pages. Use the invariant that in total we always have the same number of pages to do calculation

Calculating time cost of performing external sort
→ Always calculate number of sorted runs first, given by num_pages/ num_buffers
→ If there are different number of input and output buffers allocated, then we need to calculate the I/O costs separately. E.g. 16 input buffers of 16 pages each, 1 output buffer of 64 pages.
→ Calculate the number of passes needed. To do that, we take the number of $\log_{input\_buffers}$(sorted runs). E.g. In this case, $\log_{16}(31250) = 4$
→ If one input buffer of 16 pages, each I/O operation will read in 16 pages at once. So the number of I/O operations is given by num_pages / 16. E.g. num_pages = 10,000,000 / 16 = 625,000 sequential I/Os
→ The number of I/O operations to write out will be given by 10,000,000 / 64 = 156,250, because we perform one I/O for every 64 pages.

Relational Operators

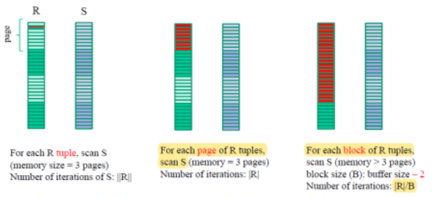
→ Motivation: Improve performance by processing queries optimally
→ We ignore output costs

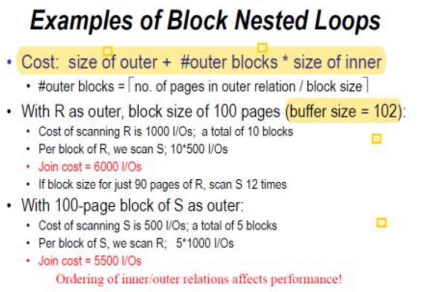
Join Algorithms
- Block-nested loop (iteration based), index nested loop (index based), sort-merge join (sort-based), hash join (partition based)

Naïve:
To perform join, for each tuples in S, we compare all the tuples in R and check if the match condition joins. Number of I/Os: numPages(R) + (numTuples(R) * numPages(S)).
We scan in 1 R page, for each R tuple, we scan all pages in S

Improvement: **Page-based Nested Loop Join**
- For each page R, we scan in all pages of S, then do a join.
I/O Cost: numPages(R) + (numPages(R) * numPages(S))

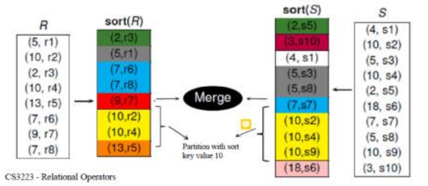
Improvement: **Block Nested Loop Join**
→ We have B buffers. We should exploit it to load in as many pages of R as possible, before doing nested loop on one page
→ For each block of R tuples, scan entire S
→ I/O Cost: (numPages(R)) + [(numPages(R) / B-2) * numPages(S)]
→ Cheaper to load smaller table as left table if buffer size divides perfectly (?)
→ Scan in R in blocks, so in total we scan each page of R once. Scan in S numBlocks time (which depends on size of B)
→ B-2 because among all the buffers we have, one is used to write in S, one is used to output


R
S
page
For each R tuple, scan S
(memory size = 3 pages)
Number of iterations of S: ||R||
For each page of R tuples, scan S (memory = 3 pages)
Number of iterations: |R|
For each block of R tuples, scan S (memory > 3 pages)
block size (B): buffer size – 2
Number of iterations: |R|/B


Examples of Block Nested Loops
- Cost: size of outer + #outer blocks * size of inner
  - #outer blocks = ⌈no. of pages in outer relation / block size⌉
- With R as outer, block size of 100 pages (buffer size = 102):
  - Cost of scanning R is 1000 I/Os; a total of 10 blocks
  - Per block of R, we scan S; 10*500 I/Os
  - Join cost = 6000 I/Os
  - If block size for just 90 pages of R, scan S 12 times
- With 100-page block of S as outer:
  - Cost of scanning S is 500 I/Os; a total of 5 blocks
  - Per block of S, we scan R; 5*1000 I/Os
  - Join cost = 5500 I/Os

Ordering of inner/outer relations affects performance!

Sort Merge Join
→ Sort R and S on join column
→ Scan and do a merge on the join column
→ Imagine we have a pointer on both R and S, we load the pages in that have the same sort key, then we can perform the join in memory. At best, everything just needs to be loaded into memory once, so |R| + |S|. But imagine both tables are very huge and all have the same join key, then when we load in as many pages as we can for R, for that block we loaded in, we need to scan entire S, which is basically a block nested loop join



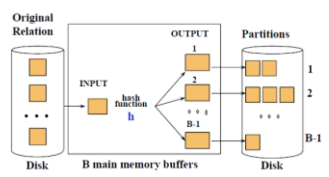
CS3223 - Relational Operators

Cost of Sort-Merge Join
- I/O Cost: Cost of sorting R and Cost of sorting S + Merge cost
- Recall cost to sort a table is 2 * #passes * numPages(S or R), i.e.
  - Cost to sort R = $2|R| \left(1 + \lceil \log_{B-1} \lceil |R|/B \rceil \rceil\right)$ for external merge sort
    - B = number of buffers
- Merging Cost Best Case: As numPages(R) + numPages(S)
- Worst Case: When each tuple of R requires scanning entire S
numPages(R) + [numPages(R) * numPages(S)]
(Can be reduced using block nested loops)
- If |R| < |S|, min cost: |R| + 1

Grace Hash Join
- Idea: Hash all the search keys, they should end up in separate buckets with minimal collisions.
- Then in each bucket, we can perform joining operation, since all equal keys are already in the same bucket (this assumes that the number of pages in each bucket can fit the buffer size)
→ 2 phases: Partition Phase, where we do the separating of the search keys into buckets. Join Phase, where we read in a partition, then try to do the matching in memory

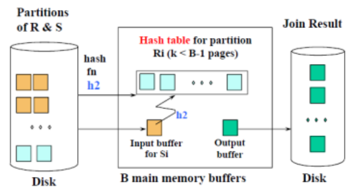
Partitioning Phase

Partitioning Phase



→ One input buffer, B-1 buckets. Hash function can be something like h(v) = v mod (B-1)!!
→ So now, we got R that is partitioned to R1, R2…R(B-1) and likewise for S1, S2…, S(B-1)

Joining Phase



→ Build hash table using all values in Ri with join key as the hash table key.
→ Now we load in pages from Si page by page. Then we can match with the hashtable, and perform the join. Each joined tuple will be added to the output buffer. Once output buffer is full we can write one page to disk

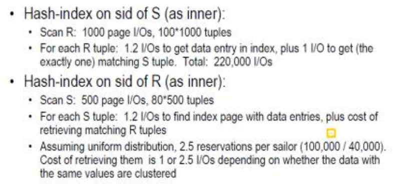
Cost of Hash Join
→ To partition, we need to read in, hash, then write out each page of each relation => 2(numPages(R) + numPages(S))
→ To join, we read in both relations once => numPages(R) + numPages(S)
→ Total: 2(numPages(R) + numPages(S)).
→ **Important:** This makes one important assumption tho, it assumes that at the join phase, we are able to fit the entire R partition into memory at one go → Each partition must take at most B-2 pages.
→ Also recall that number of partitions we have is B-1
→ Condition for this to work: M / (B-1) <= B-2, so M <= approximately B^2, or B >= root(M)
→ If cannot fit into memory, then we need to partition again. (Calculate the number of times needed to partition by doing logB-1(numPages) * 2 (1 read in 1 write out) * numPages for both relations

Index Nested Loop Join
- Idea: For each tuple in R, we search the index on S. For each matching tuple for S, we do a join. Take note that each time we search for a matching S there can be more than 1 tuple!

→ Condition: There must be an index on the join column of one of the relation, so that we can perform the index based search. This table will be the inner relation
Cost: numPages(R) + (numTuples(R) * cost to find matching Tuple in S)
→ It really depends for the cost to find matching tuple in S. If is B+ Tree, we need the cost of traversing the tree + retrieve tuple.

If is clustered index, 1I/O to retrieve all tuples of that value, if is unclustered, 1I/O per matching tuple, assuming that one page can fit all the tuples with the same search key
-→ For hash index, then cost to hash is usually 1 i/o to retrieve record for Format 1

E.g. For each tuple of R we retrieve on avg 2.5 tuples of S


- Hash-index on sid of S (as inner):
  - Scan R: 1000 page I/Os, 100*1000 tuples
  - For each R tuple: 1.2 I/Os to get data entry in index, plus 1 I/O to get (the exactly one) matching S tuple. Total: 220,000 I/Os
- Hash-index on sid of R (as inner):
  - Scan S: 500 page I/Os, 80*500 tuples
  - For each S tuple: 1.2 I/Os to find index page with data entries, plus cost of retrieving matching R tuples
  - Assuming uniform distribution, 2.5 reservations per sailor (100,000 / 40,000). Cost of retrieving them is 1 or 2.5 I/Os depending on whether the data with the same values are clustered

Other General Join Conditions
→ What if we join over several attributes => We treat the rest as selections, join on the predicate that gives us the lowest number of output tuples
Inequality Join (R.sid < S.sid)
→ Can use nested loop and sort merge join to save some cost. Can also use index nested loop join, but only for B+ trees!
→ Hash based joins cant be used because they only work for equality
→ Projection by default don't remove duplicates unless u mention distinct
→ Union/intersect/Except removes duplicates

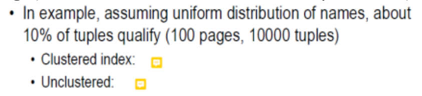
Processing Relational Algebraic Operators

- Selection – select tuples of R based on some condition c
- Selectivity → Size of result / ||R|| (basically ratio of number of results selected to the number of tuples in R)

Access Path
→ Ways to access a table to get records/entries
1. Table scan – no index, unsorted → scan the whole relation, cost is #pages in R
2. Index-only scan → Index must have all relevant information already (including the attributes in the WHERE condition!), then you can treat the index like a table.
3. Index search – We use the index to find qualifying data entries, then retrieve the corresponding data records.
→ We also perform RID lookups for index based access paths to retrieve data records

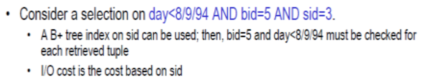
Cost/Selectivity of an Access Path
- Refers to the number of index/data pages retrieved to access the data records/entries
- Most selective access path is the one that retrieves the fewest pages. Usually, index-based access paths are better than table scans

Using an Index for Selection
- Cost will depend on number of qualifying tuples, and clustering.
E.g. (values in the brackets are already the 10%)


- In example, assuming uniform distribution of names, about 10% of tuples qualify (100 pages, 10000 tuples)
  - Clustered index:
  - Unclustered:

→ Clustered: 100 i/os
→ Unclustered: 10000 i/os

So for selection…
→ Use whichever way returns the most selective path
→ Find the most selective access path that uses the least I/Os
→ So once we find the most selective path, remember that if we have multiple conditions, we fetch **from the index** based on the most selective condition, the rest of the terms are used to discard extra tuples. (So i/os is based on the most selective attribute **for indexes.** Table scan I think can just use all the attributes at once)


- Consider a selection on day<8/9/94 AND bid=5 AND sid=3.
  - A B+ tree index on sid can be used; then, bid=5 and day<8/9/94 must be checked for each retrieved tuple
  - I/O cost is the cost based on sid

Projection Operation (Eliminates Duplicates)
→ Projection – operation to project some tuples in a relation, helps to eliminates duplicates as well

- $\pi^*_L(R)$ same as $\pi_L(R)$ but preserves duplicates

→ The * notation is a projection of attribute list L on table R that includes duplicates

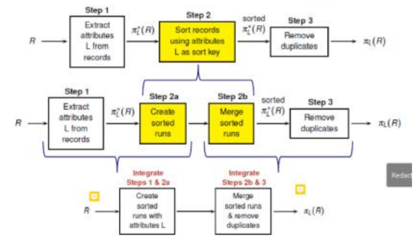
Projection: Sort based Approach
Naïve Approach:
1. Extract attributes L from records (Contains duplicates)
2. Sort records using attributes L as sort key
3. Remove duplicates to get the final projected tuples

→ In doing the naïve approach, cost breakdown:
1. Cost to scan records = numPages(R) (|R|), cost to output temp result = |pi*L(R)|
2.
- Cost to sort records = $2|\pi^*_L(R)|(\log_m(N_0)+1)$
- N_0 = number of initial sorted runs; m = merge factor

3. Cost to scan records, |pi*L(R)|, store answer (optional) = |piL(R)|

Optimized Sort-based Approach
→ However, realize that sorting step is basically the generating sorted runs + merging step.
→ Then, we can generate sorted runs and perform attribute extraction at the same time in memory to generate sorted runs with the attributes L!
→ We can also combine merging sorted runs step with removing duplicates, since the runs are all sorted, we just track the last added value, and don't add it again, we have removed duplicates while merging

Optimized Sort-based Approach



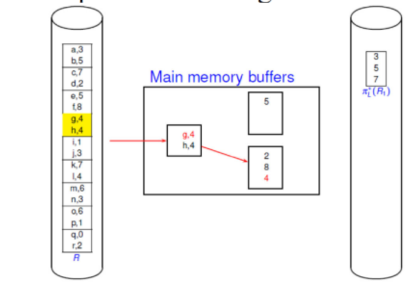
→ Runs produced are smaller than the input tuples (size ratio (size of output / table) depends on the number of fields and the size of the fields that were dropped)
→ Merging also results in number of tuples being less than the input, with the difference depending on the number of duplicates.

Cost Analysis of Sort Based Projection
- In phase 1, read original relation (size M), write out same number of smaller tuples
- In merging passes, fewer tuples written out in each pass

Projection - Hash-Based Approach
→ 2 phases:
1. Partitioning phase (partition table R into B-1 buckets)
2. Duplicate Elimination Phase: Eliminate duplicates from each (pi)*L(Ri)

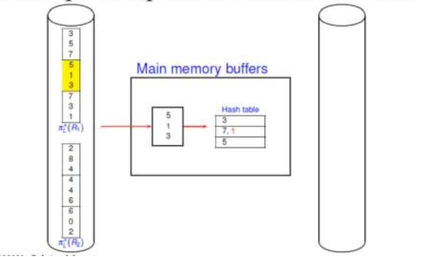
Partitioning Phase
- As usual, use one input, hash into B-1 buckets, reading one page at a time from R into input
- When we load one page of R into the input buffer, for each tuple t, **can also project out unwanted attributes from t to form t'**
- Apply hash function h on t' to distribute into respective buckets. So the tuples in each buckets already have their attributes projected. Similar to sort approach
- **Optimization:** If we happen to see duplicates in the buckets, we can remove it immediately before writing it out to disk, incurring more cpu cost but less i/o cost!

Example: Partitioning Phase



Duplicate Elimination Phase
- Initialize in memory hash table
- Read in R one page at a time, for each tuple t', check if its already in the hash table, if yes, we ignore, if not in the table yet we add it inside. Like a hash set.
- We can finally write out the tuples in the hash table to the results.

Example: Duplicate Elimination Phase



Projection Hash Based: Partition Overflow
→ What if the size of the bucket after the partition phase is larger than the available memory? We recursively apply hash-based partitioning to the overflowed partition (same as the grace hash join)

Cost Analysis of Hash Based Projection
→ We assume the h distributes the tuples in R uniformly
→ Each bucket Ri has |projectedDupTuples(R)|/(B-1)
→ Size of hash table for each bucket: Size of the bucket * fudge factor
→ B must be bigger than the value above
- Therefore, to avoid partition overflow, $B > \frac{|\pi^*_L(R)|}{B-1} \times f$
  - Approximately, $B > \sqrt{f \times |\pi^*_L(R)|}$

→ Basically, B > root(numPages(R)), kind of, but this time projected so maybe less
- Analysis:
  - Cost of partitioning phase: $|R| + |\pi^*_L(R)|$
  - Cost of duplicate elimination phase: $|\pi^*_L(R)|$
  - Total cost = $|R| + 2|\pi^*_L(R)|$

(ignore cost to write output)

→ Cost of partitioning: Load all R pages in + cost of writing out projected tuples (read + write)
→ Cost of duplicate elimination: Cost of read in the projected tuples

Index Based Projection
→ If the index contains all the wanted attributes, use an index scan! If the index and all the wanted attributes are ordered the same way, we can scan data entries in order, comparing adjacent entries for duplicates

Set Operations
- Set operations
  - Cross-product: R x S
  - Intersection: R ∩ S
  - Union: R ∪ S
  - Difference: R – S
- Intersection and cross-product: special cases of join
  - $R(A,B) \cap S(A,B) = \pi_{R.*}(R \bowtie_p S)$
    - $p = (R.A = S.A) \wedge (R.B = S.B)$
  - $R \times S = R \bowtie S$ with an empty join predicate
- Union (Distinct) and Difference are similar
  - Sorting based approach
  - Hash based approach

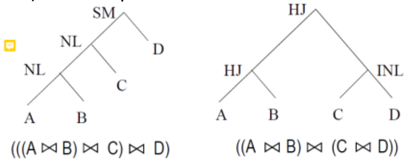
→ Cross product is obviously similar to join, just without any join predicate
→ Intersection is similar to join, because we need to compare the projected attributes
→ For Union, I guess we can do the sort based way by generating sorted runs/partitions with the wanted attributes only for both tables, then perform merging/tuple elimination
→ Difference can use sort/hash-based approach as well (In merge phase for sort, if we see they got similar attributes we skip them)

- Sorting based approach to union:
  - Sort both relations (on combination of all attributes)
  - Scan sorted relations and merge them
    - Implementation typically removes duplicates while merging (why?)
- Hash based approach to union:
  - Partition R and S using hash function h
  - For each S-partition, build in-memory hash table (using h2), scan corr. R-partition and add tuples to table while discarding duplicates
- Algorithms for R – S are similar

Aggregate Operations (COUNT, SUM, AVG, MIN, MAX)
Without grouping:
→ We can do index only scan if the attributes in the SELECT and WHERE clauses only concern those we have in an index
→ Otherwise we probably need to scan the entire table/relation
With grouping:
- With grouping:
  - Sort on group-by attributes, then scan relation and compute aggregate for each group
  - Similar approach based on hashing on group-by attributes
  - Given tree index whose search key includes all attributes in SELECT, WHERE and GROUP BY clauses, can do index-only scan; if group-by attributes form prefix of search key, can retrieve data entries/tuples in group-by order)(no sorting needed)

<u>**Query Optimization in Relational Database Systems**</u>
- Operations can be composed because each relational op returns a relation
- Queries may be composed in different ways – which is why query optimization is necessary for good performance.

Query Evaluation Plan (QEP)
- Each strategy can be represented as a QEP with choice of algos for each op.
- We want to find the best plan (or just avoid bad plans) that compute the same answer.



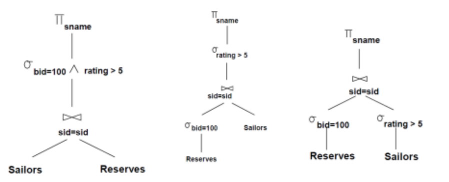
(We will use motivating example of reserves and sailors from here onwards)
- Reserves:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.
- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.

→ Many logical ways to plan a query.

```
SELECT S.sname
FROM  Reserves R, Sailors S
WHERE  R.sid=S.sid AND
   R.bid=100 AND S.rating>5
```

Example

→ Example below is a logical plan (plan without specifying what exact joins we are using)



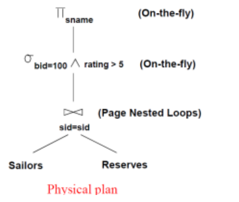
Example Physical Plan

```
SELECT S.sname
FROM  Reserves R, Sailors S
WHERE  R.sid=S.sid AND
   R.bid=100 AND S.rating>5
```

Example (Cont)

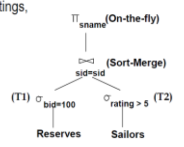
Query Evaluation Plan:

- Cost:
- Memory:



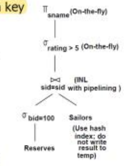
→ Do page nested loops, and once we perform join, on the fly filter based on selection condition, and project desired attribute

Alternative Plan:
→ With alternative plan, we can push selections down, reducing the number of tuples when we perform join, reducing i/o costs
- Main difference: push selections down
- Assume 5 buffers, T1 = 10 pages (100 boats, uniform distribution), T2 = 250 pages (10 ratings, uniform distribution)
- Cost of plan:
  - Scan Reserves (1000) + write temp T1 (10 pages, if we have 100 boats, uniform distribution)
  - Scan Sailors (500) + write temp T2 (250 pages, if we have 10 ratings)
  - Sort T1 (2*2*10), sort T2 (2*4*250), merge (10+250)
  - Total: 4060 page I/Os



→ Calculation assume buffer pages = 5

Alternative Plan 2 With indexes
- Clustered index on bid of Reserves
  - 100,000/100 = 1000 tuples on 1000/100 = 10 pages
- Hash index on sid (format 2). Join column sid is a key for Sailors
- INL with pipelining (outer is not materialized)
  - Project out unnecessary fields from outer doesn't help
- At most one matching tuple, unclustered index on sid OK
- Did not push "rating>5" before the join. Why?
- Cost?
  - Selection of Reserves tuples (10 I/Os); for each, must get matching Sailors tuple (1000*2.2); total 2210 I/Os


π_{sname} (On-the-fly)
$\sigma_{rating>5}$ (On-the-fly)
sid=sid
(INL with pipelining)
$\sigma_{bid=100}$
Sailors
Reserves
(Use hash index; do not write result to temp)

→ We have clustered index on reserves, so if bid = 100, 1% of all tuples = 1000 tuples = 10 pages.
→ We have hash index on sid, which is a key for sailors, so we will only retrieve 1 tuple per key.
→ Pipelining – don't store/materialize intermediate results, on the fly filter and project tuple
→ 2.2 because 1.2 i/o is the magic number to get pointer in hash index, 1 more i/o to get tuple. Total 2210 i/os.

→ As seen, there are many plans to pick from in running a query. Our goal is to find an optimal plan from a set of QEPs

Relational Algebra Equivalences

What about $\sigma_{p1 \vee p2 \dots \vee pn}(R)$?

- **Cascading of selections**: $\sigma_{p_1 \wedge p_2 \wedge \dots \wedge p_n}(R) \equiv \sigma_{p_1}(\sigma_{p_2}(\cdots(\sigma_{p_n}(R))\cdots))$
- **Commutativity of selections**: $\sigma_{p_1}(\sigma_{p_2}(R)) \equiv \sigma_{p_2}(\sigma_{p_1}(R))$
- **Cascading of projections**: $\pi_{L_1}(R) \equiv \pi_{L_1}(\pi_{L_2}(\cdots(\pi_{L_n}(R))\cdots))$, where $L_i \subseteq L_{i+1}$ for $i \in [1, n)$
- **Commutativity of cross-products**: $R \times S \equiv S \times R$
- **Associativity of cross-products**: $R \times (S \times T) \equiv (R \times S) \times T$
- **Commutativity of joins**: $R \bowtie S \equiv S \bowtie R$
- **Associativity of joins**: $R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$
- **Others**: $R \cup S = S \cup R$, $R \cap S = S \cap R$, $R \cup (S \cup T) = (R \cup S) \cup T$, $R \cap (S \cap T) = (R \cap S) \cap T$, etc.

- $\pi_L(\sigma_p(R)) \equiv \sigma_p(\pi_L(R))$ if σ involves only attributes retained by π
- $R \bowtie_p S \equiv \sigma_p(R \times S)$
- $\sigma_p(R \times S) \equiv \sigma_p(R) \times S$ if σ refers to attributes only in R but not in S
- $\sigma_p(R \bowtie S) \equiv \sigma_p(R) \bowtie S$ if σ refers to attributes only in R but not in S
- $\pi_L(R \times S) \equiv \pi_{L_1}(R) \times \pi_{L_2}(S)$ if $L_1 = L \cap attr(R)$ & $L_2 = L \cap attr(S)$
- $\pi_L(R \bowtie_p S) \equiv \pi_{L_1}(R) \bowtie_p \pi_{L_2}(S)$ if $L_1 = L \cap attr(R)$, $L_2 = L \cap attr(S)$, & every attribute in p also appears in L
- Others: $\sigma_p(R \cup S) = \sigma_p(S) \cup \sigma_p(R)$, etc.

$\sigma_p(R \bowtie S) \equiv \sigma_p(R) \bowtie S$ if σ refers to attributes only in R but not in S

→ If we are joining a table and selecting based on some predicate p, if the predicate only concerns the table R then we can just perform the selection first

Bags vs Sets

R = {a,a,b,b,b,c}
S = {b,b,c,c,d}
$R \cup S$ = ?

- SUM is implemented: $R \cup S$ = {a,a,b,b,b,b,b,c,c,c,d}
- Some rules cannot be used for bags
  - e.g. $A \cap_s (B \cup_s C) = (A \cap_s B) \cup_s (A \cap_s C)$

Let A, B and C be {x}
$B \cup_B C = \{x, x\}$ **$A \cap_B (B \cup_B C) = \{x\}$**
$A \cap_B B = \{x\}$ $A \cap_B C = \{x\}$ **$(A \cap_B B) \cup_B (A \cap_B C) = \{x, x\}$**

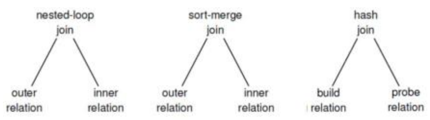
→ Relational algebra is based on sets, but in practice, in general dbms implements based on multi sets.

Query Optimizer

- We want to find the best plan. It comprises of:
1. The plan space – huge number of alternative but semantically equivalent plans
2. computationally expensive to examine all
3. Conventionally we just want to avoid bad plans
4. Enumeration algorithm – search through plan space, should be efficient (have low overhead)

Join Plan Notation

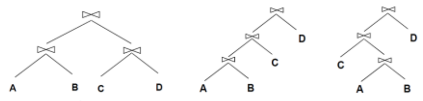
→ For nested-loop and sort-merge join, we call left table outer relation, right table inner relation (left table is the outer loop of nested for loop)
→ For hash join, we call left table build relation and right table probe relation (left table is in memory)



Plan Space

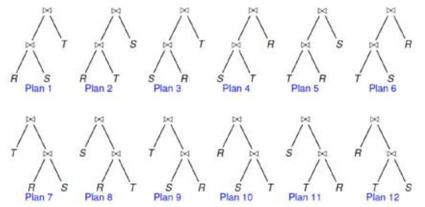
→ Left deep trees, right deep trees, deep trees, bushy trees

- Left-deep trees: right child has to be a base table
- Right-deep trees: left child has to be a base table
- Deep trees: one of the two children is a base table
- Bushy tree: unrestricted



Bushy tree Left-deep tree Deep tree

→ Plan space can be huge. For R join S join T there are the following ways, which hasn't even accounted for the algorithms

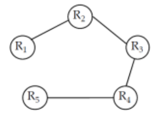


Search Algorithms for Query Optimization

- Brute force (Exhaustive, complete space). Enumerate all and pick the best
- Greedy techniques (polynomial time)
→ Get smallest relation next, smallest result next etc
- Randomize/transformation techniques
- System R (Dynamic Programming with Pruning)

Multi-Join Queries – Greedy Algorithms

- We focus on multi-join queries, pushing selections/projects down as early as possible.
- Heuristic 1: Smallest relation next. We pick the smallest relation as the base table, and keep joining with the next smallest
  - What if R1 < R5 < R3 < R2 < R4???

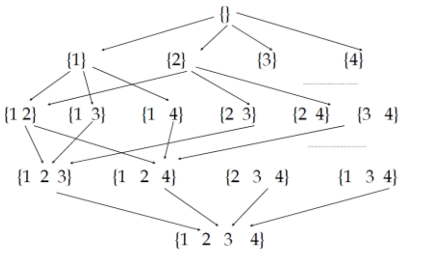


Another heuristic:
Smallest result next?

→ However, what if R5 and R1 are not being joined? We just performed a cross product which is the worst possible operation
→ One other possible heuristic would be to always take the table that would give us the smallest result next

System-R (Dynamic Programming)

- We start bottom up from 1 table, 2 tables..
- We plan for a set of cardinality i (e.g. tables {1,2,3}, {1,2,4}…) as extensions of the best plan for a set of cardinality i-1. We only keep 1 best plan among all possible plans of that cardinality.

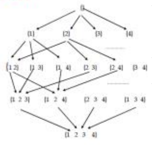


→ We prune the search space based on the principle of optimality: **If 2 plans differ only in a subplan, then the plan with the better subplan is also the better plan**
→ Reduce computation overhead due to overlapping subproblems
→ We only look at left deep trees.

accessPlan(R) – give me the best plan for relation (single table) R
joinPlan(p1, R) – Give me plan p2, which is the best possible way to join the partial join plan p1 with R
Optimal plans are stored in optplan() array and are reused rather than recomputed

Algorithm

```
for i = 1 to N
    optPlan({Ri}) = accessPlan(Ri)
for i = 2 to N {
    forall S subset of {R_1, R_2, … R_n} such that |S|=i {
            bestPlan = dummy plan with infinite cost
            forall Rj, Sj, |Sj| = i-1   such that S = {Rj} U Sj {
                    p = joinPlan(optPlan(Sj), Rj)
                    if cost(p) < cost(bestPlan)
                        bestPlan = p
            }
            optPlan(S) = bestPlan
    }
}
P_opt = optPlan{R_1, R_2, … R_n}
```



→ We init our dp array with the costs of accessing the base tables.
→ Then we slowly build up each plan of cardinality 2, 3, 4 of different combinations.
→ We do so by trying out all the possible combinations. E.g. if we want to make {1,2,3,4} we test {2,3,4} U {1}, {1,3,4} U {2}…

Dynamic Programming Example

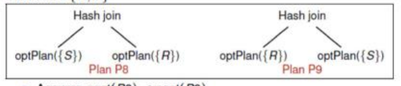
→ Suppose we have 3 tables S, R, T, we have B+ tree index on I_B, I_C, I_E. We only support hash join, avoid cross products, performing the following query:

$\sigma_p(R \bowtie_{R.A=S.X} S \bowtie_{R.D=T.F} T)$, $p = (R.B > 10) \wedge (R.C = 20) \wedge (T.E < 100)$

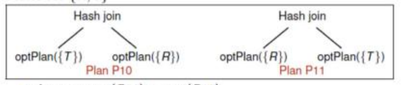
- **Plans for {R}**
  - Plan P1: Table scan with "$(B > 10) \wedge (C = 20)$"
  - Plan P2: Index seek with I_B & RID-lookups with "$C = 20$"
  - Plan P3: Index seek with I_C & RID-lookups with "$B > 10$"
  - Plan P4: Index intersection with I_B & I_C, and RID-lookups
  - Assume $cost(P3) < cost(P4) < cost(P2) < cost(P1)$
  - optPlan({R}) = P3
- **Plans for {S}**
  - Plan P5: Table scan of S
  - optPlan({S}) = P5
- **Plans for {T}**
  - Plan P6: Table scan of T with "$(E < 100)$"
  - Plan P7: Index seek with I_E & RID-lookups
  - Assume $cost(P7) < cost(P6)$
  - optPlan({T}) = P7

→ We enumerate all plans for r,s,t and save the best plan

- **Plans for {R, S}**

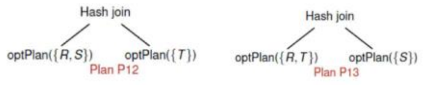


  - Assume $cost(P8) < cost(P9)$
  - optPlan({R, S}) = P8
- **Plans for {R, T}**



  - Assume $cost(P11) < cost(P10)$
  - optPlan({R, T}) = P11

→ We try and join the tables to get {R,S} and {R,T} and save the respective best plans



- Assume $cost(P12) < cost(P13)$
- optPlan({R, S, T}) = P12

→ Perform the final join and get the best plan.

System-R Analysis

- Time & Space complexity: for k relations, left deep trees: $2^k - 1$ entries
- For bushy trees, $O(3^k)$
- Not always optimal, because of the notion of interesting orders.
→ For example, we have a plan that used hash join on R and S, and sort merge join for R and S. Hash join took less i/o and was chosen. However, choosing sort merge would have resulted in globally optimal, because sort merge might have sorted the attributes in a way that made it optimal to pick it (e.g. we have a order by clause on that attribute) over the hash join
→ Interesting orders: order by/group by
→ DP with interesting orders is optimal in the plan space we search (in this case left deep)

Randomized Techniques for Query Optimization

→ We have some state (i.e. a plan), with some cost associated with it determined by some cost model. A move is a perturbation applied to a state to another state.
→ 2 states are neighboring states if one more sufficies to go from one state to another
→ A move set is the set of moves available to go from 1 state to another. Any one move is chosen from this move state randomly, based on a probability associated with selecting some move
→ Randomized algorithms look at full plans across the search space.
→ Local minimum – state such that its cost is lower than that of all neighboring states
→ Global minimum – smallest of all local minimums, should only have 1
→ Move that takes one state to another with lower cost – downward move; otherwise called upward move

Algorithm

→ We start with a random plan, then we move to all neighbor states until we get a local minimum. Then we choose another random plan and repeat until we get a near-optimal minimum.

```
S = initialize()  // initial plan
minS = S // cost of plan S – currently the best
repeat {
    repeat {
        newS = move(S)  // move to a new plan
        if (cost(newS) < cost(S))
            S = newS
    } until ("local minimum reached")
    if (cost(S) < cost(minS))
        minS = S
    newStart(S);  // iterate with a different initial plan
} until ("stopping condition satisfied")
return (minS);
```

Repeat until a near-optimal minimum Is reached

By doing so repeatedly, a local minimum can be reached

A move is accepted if it is a downward move, i.e., has a lower cost

Issues on local optimization for randomized algo

→ How is each start state obtained? We should obtain is quickly. Some ways include: randomly, using greedy heuristics, or even allowing upward moves, hoping that it will help us jump out of the local minimum and eventually decrease cost

→ How do we know if we have local minimum? We cannot examine all neighbors to verify the one that was local minimum, so we base off random sampling, so if no one is lower in some process, we just consider it as local minimum.
- Number of neighbors to examine can be specified as a parameter and called the sequence length. Can also be time-based.

→ How do we determine stopping criterion of algorithm?
- Basically, how many times we execute the outer loop. Can be fixed or is given by sizeFactor * N, where sizeFactor is a parameter and N is the

number of relations. Depends on number of tables we are joining.

Transformation Rules for randomized algo
- We can possibly only restrict to left deep trees, then permutate by swapping the order of base tables
- 3Cycle heuristic: Select 3 relations, then cycle I,j,k with j,k,I or k,I,j etc

Comparison between Exhaustive, Greedy and Randomized Algo
- Search Space, plan quality and optimization overhead
- Exhaustive search largest space, but randomized might perform more search operations because it might repeat the search operation by chance
- Randomization can have higher optimization overhead if the parameters not set well

<u>Cost Models</u>
- Defined by combination of CPU and I/O costs
- Objective is to rank plans; the exact values are not important
- Relies on statistics on relations and indexes (how many tuples of some value we have..), formulas to estimate CPU and I/O cost, estimate selectivity of operators and intermediate results

Cost Estimation
- Estimate cost of each operation in plan tree. Depends on input (e.g. join cost of r1 and r2, r2 r3 etc), buffer size, availability of indexes, algos used
- Estimate the size of result for each operation in the tree, using information about the relations and making assumptions such as uniform distribution of data and independence of predicates.

Statistics and Catalogs
- Need info about the relations and indexes involved. We store them in Catalogs.
Catalogs typically contain:
- # tuples of R (||R||), #bytes in each R tuple (S(R))
- #blocks/pages to hold all R tuples (|R|)
- #distinct values in R for attribute A (V(R,A))
- N(pages) for each index
- Index height, min/max values for each tree index
→ Catalogs updated periodically

Estimation Assumptions
→ Uniform distribution (uniformity assumption)
→ Independent distribution of values in different attributes (independence assumption)

→ Inclusion Assumption - if the number of distinct values of A is R < B in S, we assume that the values of A in R is a subset of B in S, assuming a foreign key relationship of A to B
Example:

R

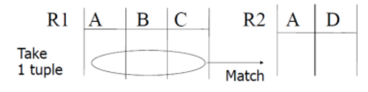
A	B	C	D
cat	1	10	a
cat	1	20	b
dog	1	30	a
dog	1	40	c
bat	1	50	d

A: 20 byte string
B: 4 byte integer
C: 8 byte string
D: 5 byte string

$||R|| = 5$ $\quad S(R) = 37$
$V(R,A) = 3$ $\quad V(R,C) = 5$
$V(R,B) = 1$ $\quad V(R,D) = 4$

Example: Computing Number of tuples in join operation

<u>Computing T(W)</u> when $V(R1,A) \leq V(R2,A)$



1 tuple of R1 matches with $\frac{||R2||}{V(R2,A)}$ tuples of R2

so $||W|| = ||R1|| \times \frac{||R2||}{V(R2,A)}$

If $V(R2,A) \leq V(R1,A)$ $\quad ||W|| = \frac{||R2|| \times ||R1||}{V(R1,A)}$

→ There are less distinct values of A in R1 than R2. So, when we join, each page in R1 matches with ||R2|| / numDistinct(R2,A).
→ Other way round if there are more distinct values of A in R2 than R1.
→ For complex expressions involving more than 2 tables, we will also need intermediate T,S,V results

E.g. $W = \underbrace{[\sigma_{A=a}(R1)]}_{\text{Treat as relation U}} \bowtie R2$

$||U|| = ||R1||/V(R1,A) \qquad S(U) = S(R1)$

Also need V (U, *) !!

R1

A	B	C	D
cat	1	10	10
cat	1	20	20
dog	1	30	10
dog	1	40	30
bat	1	50	10

$V(R1,A)=3$
$V(R1,B)=1$
$V(R1,C)=5$
$V(R1,D)=3$
$U = \sigma_{A=a}(R1)$

V(U,A) = ? V(U, B) = ? V(U,C) = ?

V(D,U) ... somewhere in between V(U,B) and V(U,C)

→ In the example above, we create U, which filters R1 based on A=a. So, V(U,A) = 1 because we filtered based on it, V(U,B) = 1 because all the same in the table. V(U,C) = ||U|| because all unique, V(U,D) is somewhere in between V(U,B) and V(U,C)

Estimation of Join Example

For Joins $\quad U = R1(A,B) \bowtie R2(A,C)$

$V(U,A) = \min \{ V(R1, A), V(R2, A) \}$
$V(U,B) = V(R1, B)$
$V(U,C) = V(R2, C)$

→ Reasoning: Preservation of value sets

$Z = R1(A,B) \bowtie R2(B,C) \bowtie R3(C,D)$

$\boxed{R1}$ $||R1|| = 1000 \quad V(R1,A)=50 \quad V(R1,B)=100$
$\boxed{R2}$ $||R2|| = 2000 \quad V(R2,B)=200 \quad V(R2,C)=300$
$\boxed{R3}$ $||R3|| = 3000 \quad V(R3,C)=90 \quad V(R3,D)=500$

$Z = R1(A,B) \bowtie R2(B,C) \bowtie R3(C,D)$
$||R1|| = 1000 \quad V(R1,A)=50 \quad V(R1,B)=100$
$||R2|| = 2000 \quad V(R2,B)=200 \quad V(R2,C)=300$
$||R3|| = 3000 \quad V(R3,C)=90 \quad V(R3,D)=500$

Partial Result: $\quad U = R1 \bowtie R2$

$||U|| = \frac{1000 \times 2000}{200}$

$V(U,A) = 50$
$V(U,B) = 100$
$V(U,C) = 300$

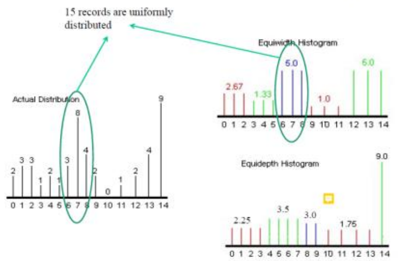
$Z = U \bowtie R3$

$||Z|| = \frac{1000 \times 2000 \times 3000}{200 \times 300}$

$V(Z,A) = 50$
$V(Z,B) = 100$
$V(Z,C) = 90$
$V(Z,D) = 500$

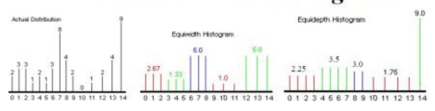
Error in Estimating Size of Plan
- Errors due to uniformity assumption, inability to capture correlation, inaccuracy of cost model, which are propagated to other operators at higher level of plan tree, affecting future results
Dealing with errors: Maintain more detailed statistics (at finer granularity), sample actual intermediate results at runtime and recalculate on the fly

Statistical Summaries of Data
→ More detailed info are sometimes stored e.g. histograms of values in some attributes.
→ Divides values on a column into k buckets, where k is either predetermined or computed based on space allocation
→ Equi-width vs Equi-depth histogram



→ Use information from histogram to get different estimation

Estimations with Histograms



Query Q: $\sigma_{A=6}(R)$

Actual value, $||Q|| = 3$
Without histogram, $||Q|| = 45/15 = 3$
Equiwidth histogram, $||Q|| = 15/3 = 5$
Equidepth histogram, $||Q|| = 14/4 = 3.5$

Query Q: $\sigma_{A=10}(R)$

Actual value, $||Q|| = 0$
Without histogram, $||Q|| = 45/15 = 3$
Equiwidth histogram, $||Q|| = 1$
Equidepth histogram, $||Q|| = 1.75$

Summary
→ Query optimization is NP hard. Instead of finding best plan we largely try to avoid bad plans. Many different strategies have been proposed, greedy heuristics are fast but might generate non-optimal plans.
→ DP is effective at expensive of high optimization overhead

<u>Transaction Management</u>
→ Sequence of reads and writes of database objects. Unit of work that must commit or abort as atomic unit
→ ACID: Atomicity (all actions in xact happen, or none), Consistency (if each xact is consistent, and the db starts consistent, it ends consistent), Isolation (Execution of one xact is isolated from that of other xacts), Durability (if a xact commits, its effects persist)
→ DBMS ensures atomicity and durability by using logging to perform recovery
→ DBMS maintains consistency by using integrity constraints
→ DBMS ensures isolation by performing concurrency correctly

- Concurrency Control
  - Provide correct and highly available data access in the presence of concurrent access by many users
- Recovery
  - Ensures database is fault tolerant, and not corrupted by software, system or media failure
  - 24x7 access to mission critical data

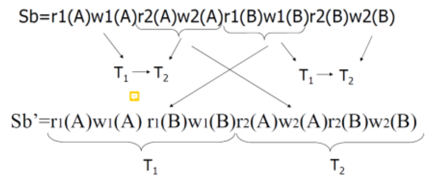
Transactions
→ Can be seen as a sequence of actions:
Ri(O): Ti reads an object O
Wi(O): Ti writes an object O
Ci: Ti completes successfully
Ai: Ti terminates unsuccessfully

Serial Schedule
- A schedule where one transaction starts and completes before another transaction does

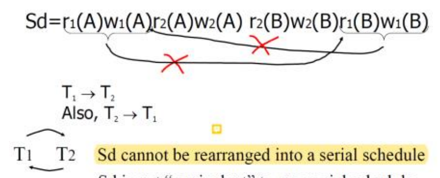
Schedule A: Serial Schedule

T1	T2	A	B
		25	25
Read(A); A ← A+100			
Write(A);		125	
Read(B); B ← B+100;			
Write(B);			125
	Read(A);A ← A×2;		
	Write(A);	250	
	Read(B);B ← B×2;		
	Write(B);		250
		250	250

→ Schedules can also overlap to be non-serial, but how do we know which non-serial schedules are good schedules?


Sb=r1(A)w1(A)r2(A)w2(A)r1(B)w1(B)r2(B)w2(B)
T1 → T2
T1 → T2
Sb'=r1(A)w1(A) r1(B)w1(B)r2(A)w2(A)r2(B)w2(B)
T1
T2

no cycles ⇒ Sb is "equivalent" to a serial schedule Sb' (in this case T_1,T_2)
→ For objects A and B, transaction 2 reads from transaction 1. So no cycle and is a good schedule
→ If some schedule overlaps where A reads from B and B reads from A, then it's a bad schedule and there is no isolation



Sd cannot be rearranged into a serial schedule
Sd is not "equivalent" to any serial schedule
Sd is "bad"

→ Any 2 actions on the same object where one of them is a write action would result in a conflicting action that needs to be examined carefully

Serial vs Serializable Schedule
- Serial: No interleaving of actions from different transactions at all
- Serializable: A schedule whose effect on any consistent DB instance is guaranteed to be identical to some serial schedule. "can be serialized"

Conflicting Actions Lead to:
1. Dirty Read Problem: When you read a value from an object that was updated by another transaction and not yet committed. You can see an inconsistent DB state

- Dirty read problem (due to WR conflicts)
  - T_2 reads an object that has been modified by T_1 (which has not yet committed)
  - T_2 could see an inconsistent DB state!

T_1	T_2	Comments
		$x = 100$
R(x)		100
$x = x + 20$		
W(x)		$x = 120$
	R(x)	120
	$x = x \times 2$	
	W(x)	$x = 240$

- For every serial schedule, the final value of x is 200

2. Unrepeatable Read Problem
→ When you read from some object, it has value x. Then, you read again, the value becomes x', because some other transaction went to write to that object. However, in a serial schedule, both values of reads by me should be the same

- Unrepeatable read problem (due to RW conflicts)
  - T_2 updates an object that T_1 has just read while T_1 is still in progress
  - T_1 could get a different value if it reads the object again!

T_1	T_2	Comments
		$x = 100$
R(x)		100
	R(x)	100
	$x = x - 20$	
	W(x)	$x = 80$
R(x)		80

- For every serial schedule, both values read by T_1 are the same

3. Lost Update Problem

→ I want to update some value in an object, so I read from it. Then I want to update it to some value, after which other values can update the value based on my new value. However, in the midst of concurrency, some other transaction also read the older value and based their operations on that value. My update is now gone

Lost update problem (due to WW conflicts)

- T_2 overwrites the value of an object that has been modified by T_1 while T_1 is still in progress
- T_1's update is lost!

T_1	T_2	Comments
		$x = 100$
R(x)		100
	R(x)	100
$x = x + 20$		
	$x = x \times 2$	
W(x)		$x = 120$
	W(x)	$x = 200$

- For schedule (T_1, T_2), the final value of x is 240
- For schedule (T_2, T_1), the final value of x is 220

Conflict Equivalent Schedule

- S1 and S2 are conflict equivalent if S1 can be transformed into S2 by a series of swaps on non-conflicting actions

→ S1 and S2 order every pair of conflicting actions of 2 committed Xacts in the same way

→ A schedule is conflict serializable if it is conflict equivalent to some serial schedule

→ Note: Some serializable schedules are not conflict serializable!

- S1: w1(Y); w1(X); w2(Y); w2(X); w3(X)
  - Serial schedule
- S2: w1(Y); w2(Y); w2(X); w1(X); w3(X)
  - Serializable?
- S1, S2 conflict equivalent?
- What is the problem?
  - In the schedule, what we essentially have are blind writes, i.e., a write on an object O that did not read O prior to the write

→ S2 is serializable cus all the writes are blind writes, and Y is last written by xact2, X last written by xact3 correctly

→ But they are not conflict equivalent because in conflict equivalence we cannot swap the positions of w1(X) and w2(Y) because they are conflicting actions

Checking for conflict serializability

→ Use precedence graph

Precedence (Conflict Serializability) graph P(S) (S is schedule)

Nodes: transactions in S

Arcs: $T_i \rightarrow T_j$ whenever

- $p_i(A)$, $q_j(A)$ are actions in S
- $p_i(A) <_S q_j(A)$
- at least one of p_i, q_j is a write

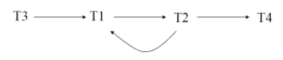
→ Draw arrow from t1 → t2 if t1 comes before t2 and one of t1/t2 is a write

→ If the precedence graph, P(S), is acyclic, S is conflict serializable.

→ S1, S2 conflict equivalent => P(S1) = P(S2)

→ However, if P(S1) => P(S2) they may not be conflict equivalent

- What is P(S) for
  S = w3(A) w2(C) r1(A) w1(B) r1(C) w2(A) r4(A) w4(D)



$$\begin{tikzcd} T3 \arrow[r] & T1 \arrow[r] \arrow[r, bend right, dash] & T2 \arrow[r] & T4 \end{tikzcd}$$

- S1 = w3(A) w2(C) r1(C) r1(A) w2(B) w1(B) w2(A)
- S2 = w3(A) r1(A) w2(B) w1(B) r1(C) w2(C) w2(A)

→ These 2 schedules have same precedence graph but they are not conflict equivalent

View Equivalent Schedules

→ 2 schedules S and S' are view equivalent if they satisfy the following conditions:

1. **Initial Read:** If Tx reads the initial value of A in S, it must also read the initial value of A in S'
2. **Updated Read:** If Tx reads a value of A written by Ty in S, Tx must also read the value of A written by Ty in S'
3. **Final Write:** For each data object A, the xact that performs the final write on A in S must also perform the final write on A in S'

→ A schedule is view serializable if S is view equivalent to some serial schedule over the same set of Xacts.

S	S_1
$R_1(x)$	$R_1(x)$
$R_2(y)$	$R_1(y)$
$W_3(x)$	$W_1(z)$
$W_3(z)$	$R_2(y)$
$R_2(x)$	$R_2(x)$
$R_1(y)$	$W_2(z)$
$W_1(z)$	$W_3(x)$
$W_2(z)$	$W_3(z)$

Not view equivalent

S	S_2
$R_1(x)$	$R_1(x)$
$R_2(y)$	$R_1(y)$
$W_3(x)$	$W_1(z)$
$W_3(z)$	$W_3(x)$
$R_2(x)$	$W_3(z)$
$R_1(y)$	$R_2(y)$
$W_1(z)$	$R_2(x)$
$W_2(z)$	$W_2(z)$

View equivalent/serializable

→ For all the read's just make sure in both schedules, they are reading from the same source

→ Make sure the same xact perform the final write

Test for View Serializability

- Polygraphs – generalization of precedence graphs for testing view serializability.

Theorem: Polygraph(S) is acyclic => S is view serializable

Polygraph of a schedule S

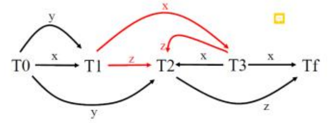
1. A node for each transaction. We have 2 additional nodes, T0 and Tf. T0 writes the initial values for all objects and Tf reads final values from all objects

2. Edges – For each action $r_y(A)$ with source Tx, add an arc from Tx to Ty

3. Look at all the write actions in the schedule. Look at all arrows that concern the same object. We connect all these write actions either to the start of the pair or from the end of the pair.

→The polygraph approach works only when each transaction only writes once on an object

- Let T_j be the source of a read $r_i(X)$, i.e., $T_j \rightarrow T_i$
- Let T_k be another transaction that also writes X
- We cannot allow T_k to intervene between T_j and T_i
- So, T_k must be before T_j or after T_i
- Add edges $T_k \rightarrow T_j$ and $T_i \rightarrow T_k$ to the polygraph
  - Intuitively, only one of these edges is "real"; and we can choose either of them when we try to make the polygraph acyclic at the end of the process
  - Special cases:
    - If T_j is T_0 then it is not possible for T_k to appear before T_j so we need only to add one edge $T_i \rightarrow T_k$
    - If T_i is T_f then it is not possible for T_k to appear after T_i so we need only to add one edge T_k to T_j



$$\begin{tikzcd} T0 \arrow[r, "x"] \arrow[r, bend left, "y"] \arrow[rr, bend right, "y"'] & T1 \arrow[r, "z"] \arrow[rr, bend left, "x"] & T2 \arrow[rr, bend right, "z"'] & T3 \arrow[l, "x"'] \arrow[l, bend right, "z"'] \arrow[r, "x"] & Tf \end{tikzcd}$$

S
$R_1(x)$
$R_2(y)$
$W_3(x)$
$W_3(z)$
$R_2(x)$
$R_1(y)$
$W_1(z)$
$W_2(z)$

The graph is acyclic – with the serial order of T1; T3; T2
It is view serialializable!

Conflict vs View Serializability

→ A schedule which is conflict serializable is also view serializable

→ View serializability: conflict serializable that allows blind writes

→ If S is view serializable and S has no blind writes, it is also conflict serializable.

Cascading Aborts

If Ty read from Tx, then Ty must abort is Tx aborts.

→ Tx's abort cascaded to Ty

→ Recursive aborting is called cascading abort

Recoverable Schedule

A schedule S is said to be recoverable if Tx writes to some object O and Ty reads from it, Tx commits before Ty commits

→ This schedule is not recoverable because T2 read from T1 and T1 hasn't committed but T2 did

T_1	T_2
$W_1(x)$	
	$R_2(x)$
	$W_2(y)$
	$Commit_2$

Cascadeless (Avoid Cascading Abort) Schedules

→ Recoverable schedules guarantee that committed xacts will not be aborted, but there can still be cascading aborts.

→ Cascading aborts are expensive to bookkeep and there is a performance penalty incurred

→ DBMS should also permit reads only from committed Xacts.

→ A schedule S is a cascadeless schedule if for every Wi(O) in S, until Ti either commits or aborts, there are no reads

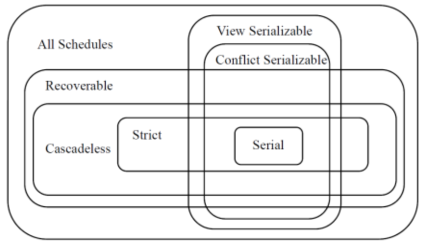
→ Cascadeless schedules are also recoverable schedules

Strict Schedules

A schedule S is a strict schedule if for every Wi(O) in S, until Ti either commits or aborts, there are no reads or writes on it.

→ A strict schedule is also a cascadeless schedule

Relationship between schedules



<u>Concurrency Control</u>

→ We want to enforce serializable schedules to increase concurrency

CC Algorithms: Pessimistic CC (Lock-based, timestamp-based CC), Multiversion CC, optimistic CC

Pessimistic CC

- We have a locking protocol with lock and unlock actions on some object e.g. $l_i(A)/u_i(A)$

CC Rules

1. Transactions must be well-formed, i.e.

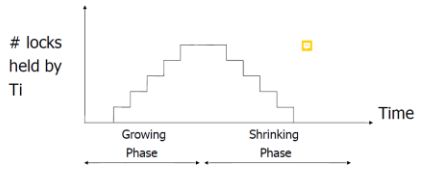
$$Ti: \; \ldots \; li(A) \ldots pi(A) \ldots ui(A) \ldots$$

2. Legal Scheduling, if someone grabs lock on A, no one else can

$$S = \ldots\ldots. \; \underbrace{li(A) \;\ldots\ldots\ldots.. \; ui(A)}_{\text{no } lj(A)} \;\ldots\ldots..$$

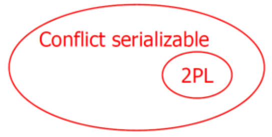
3. 2 phase locking (2PL) for transactions

$$Ti = \underbrace{\ldots\ldots.}_{\text{no unlocks}} \; li(A) \;\ldots\ldots\ldots.. \; ui(A) \; \underbrace{\ldots\ldots..}_{\text{no locks}}$$



→ Transaction must have growing and shrinking phase, if release lock, then u cannot grab anymore new locks

→ Ensures conflict serializability if rules 1,2,3 satisfied



Example:

Schedule G

T1	T2
l1(A);Read(A)	
A ← A+100;Write(A)	
l1(B); u1(A)	
	l2(A);Read(A)
	A ← Ax2;Write(A);
Read(B);B ← B+100	
Write(B); u1(B)	
	l2(B); u2(A);Read(B)
	B ← Bx2;Write(B);u2(B);

delayed

→ However, 2PL can cause deadlocks

T1	T2
l1(A); Read(A)	l2(B);Read(B)
A ← A+100;Write(A)	B ← Bx2;Write(B)
delayed	delayed

→ deadlocked transactions are aborted and rolled back

Improving Concurrency – Shared Locks

→ We want to improve concurrency by allowing laxer rules in interleaving. So, we establish 2 kinds of locks, a shared and exclusive locks (S/X)

→ S for read actions, X for write actions.

Compatibility Matrix:

Compatibility matrix

		New request	
		S	X
Lock already held in	S	True	False
	X	False	False

2PL with shared locks

→ For rule 3 with S/X locks, if we upgrade S → X lock, it is allowed (even though we are releasing read lock). If is S → {S,X}, this is also allowed. However, downgrading for X → S is not allowed in 2PL

→ Rule 1,2,3 fulfilled for shared locks => conflict serializable schedules.

Strict 2PL protocol

→ Same as 2PL, except a Xact must hold on to locks until Xact commits or aborts.

→ Strict 2PL schedules are strict and conflict serializable (and conflict serializable = serializable), no dirty reads, can deadlock, is view serializable, no unrepeatable reads

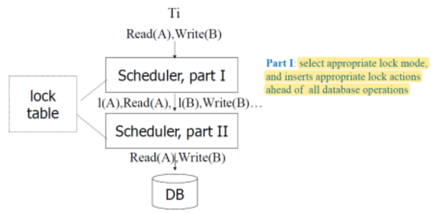
Locking in practice / commercial DBMS

Start with a sample locking system:

→ Don't trust transactions to request/release locks. Hold all locks until transaction commits.

→ This way, all transactions will be strict. However, you might be acquiring / holding onto locks the transaction no longer needs.

Architecture of locking Scheduler

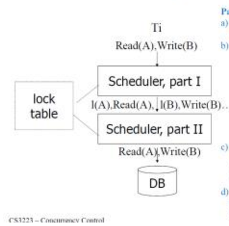

Ti
Read(A),Write(B)
Scheduler, part I
lock table
l(A),Read(A),l(B),Write(B)…
Scheduler, part II
Read(A),Write(B)
DB
Part I: select appropriate lock mode, and inserts appropriate lock actions ahead of all database operations

→ Part 1: Select appropriate lock mode and insert appropriate lock actions ahead of operations

→ Part 2: Execute the operations. This determines whether locks should be granted. If not, transactions are delayed.

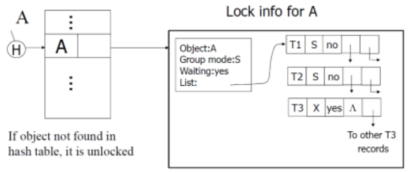
- Once transactions are granted and is not delayed, if the action is a normal opr, we send it to dbms.
- If action is a lock opr, then check if lock can be granted. If yes, update lock table. If not, block transaction's progress, but update lock table that xact is waiting
- When transaction commits/aborts, part I is notified and releases all locks. Part 2 will be notified if there are xacts waiting.
- Part 2 determines which xacts to be given locks, processing those that acquire locks

Architecture of a Locking Scheduler



Part II:Execute the operations
a) It determines if lock should be granted; if not, then transaction is delayed.
b) If transaction is not delayed,
- If action is a normal opr, then send it to the dbms
- If action is a lock opr, then check if lock can be granted
  * if so, update lock table
  * if not, delay transaction but update lock table to reflect transaction waiting
c) When a transaction commits/aborts, Part I is notified and releases all locks. Part II will be notified if there are transactions waiting.
d) Part II determines next transactions to be given the released locks. Those that acquired locks can be processed. 78

Lock Table (Hash table)



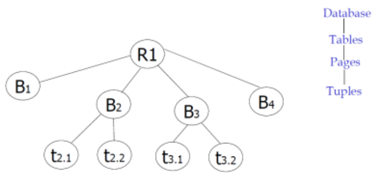
What are the objects we lock?

→ If we lock an entire relation, we need few locks but there will be very little concurrency.

→ If we lock at low granularity (tuples), then need more locks but have more concurrency.

Hierarchy of DB Elements

→ DB, tables, pages, tuples



Warning Protocol

Warning Protocol

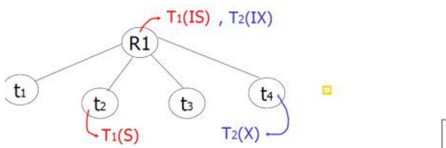
SIX mode: Like S & IX at the same time

Requestor

Holder	IS	IX	S	SIX	X
IS	T	T	T	T	F
IX	T	T	F	F	F
S	T	F	T	F	F
SIX	T	F	F	F	F
X	F	F	F	F	F

Does it make sense to have XIS?

Multiple Granularity: Warning Protocol

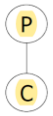


IS – Intent to get S lock(s) at *finer granularity*

IX – Intent to get X lock(s) at *finer granularity*

→ Introduce IS, IX, SIX locks as well. Can acquire stricter locks as we go to finer granularity

Parent locked in	Child can be locked in
IS	IS, S
IX	IS, S, IX, X, SIX
S	[S, IS] not necessary
SIX	X, IX, [SIX]
X	none



<u>Rules</u>

(1) Follow multiple granularity comp function
(2) Lock root of tree first, any mode
(3) Node Q can be locked by Ti in S or IS only if parent(Q) locked by Ti in IX or IS
(4) Node Q can be locked by Ti in X,SIX,IX only if parent(Q) locked by Ti in IX,SIX
(5) Ti is two-phase
(6) Ti can unlock node Q only if none of Q's children are locked by Ti

→ Lower level nodes must be unlocked before you can unlock higher level nodes.

Some examples on 2 level hierarchy

→ Suppose a T1 scans R and wants to update a few tuples. => T1 gets a SIX lock on R, then gets X lock on tuples that are updated

→ T2 uses an index to read only a part of R => T2 gets an IS lock on R, repeatedly gets an S lock on tuples of R

→ T3 reads all of R => T3 either gets an S lock on R, or can behave like T2 and escalate locks as it comes to the tuple/page. But this could mean acquiring too many low level locks

Locks on Insert/Delete operations

1. Get exclusive lock on A before deleting A
2. If Ti inserts, Ti is given exclusive lock on A

Concurrency Control Anomalies: Phantom Read Problem

→ When a transaction re-executes some query, the set of rows satisfying the condition has changed due to another recently committed transaction, i.e. Differing return results due to new rows being added or rows being deleted

Phantom Read Problem

Accounts

account	name	balance
100	Alice	5000
200	Bob	800
300	Carol	1000

→

Accounts

account	name	balance
100	Alice	5000
200	Bob	800
300	Carol	1000
400	Dave	3000

```
begin transaction;
select   name
from     Accounts
where    balance > 1000;
                                begin transaction;
                                insert into Accounts
                                   values (400,'Dave',3000);
                                commit;
select   name
from     Accounts
where    balance > 1000;
commit;
```

```
R1(100,Alice,5000)
R1(200,Bob,800)
R1(300,Carol,1000)
Output: {Alice}
W2(400,Dave,3000)
Commit2
R1(100,Alice,5000)
R1(200,Bob,800)
R1(300,Carol,1000)
R1(400,Dave,3000)
Commit1
Output: {Alice, Dave}
```

Phantom Update Problem

T1: Insert <99,Gore,...> into R
T2: Insert <99,Bush,...> into R

T1	T2
S1(o1)	S2(o1)
S1(o2)	S2(o2)
Check Constraint	Check Constraint
-- no such key, ok to insert	-- no such key, ok to proceed
⋮	⋮
Insert o3[99,Gore,..]	
	Insert o4[99,Bush,..]

Violation of key constraint!

CS3223 – Concurrency Control

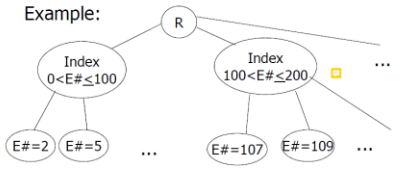
→ Violation of pkey constraint because both try to insert tuples with same pkey

→ Solution? Use multiple granularity tree, and before inserting of node Q, we have to lock parent of Q in X mode.

T1: Insert<99,Gore>	T2: Insert<99,Bush>
T1	T2
X1(R)	
	X2(R) ← delayed
Check constraint	
Insert<99,Gore>	
U(R)	
	X2(R)
	Check constraint
	Oops! e# = 99 already in R!

→ Can also lock index on R

Example:



→ Lock the node 100<E#<= 200, basically locking all R in the range of those values. No other transactions can modify the values within this range or subtrees under this index node, which solves the update problem

ANSI SQL Isolation Levels

ANSI SQL Isolation Levels

Isolation Level	Dirty Read	Unrepeatable Read	Phantom Read
READ UNCOMMITTED	possible	possible	possible
READ COMMITTED	not possible	possible	possible
REPEATABLE READ	not possible	not possible	possible
SERIALIZABLE	not possible	not possible	not possible

- SQL's SET TRANSACTION ISOLATION LEVEL command

```
BEGIN TRANSACTION;
SET TRANSACTION ISOLATION LEVEL
          { READ UNCOMMITTED |
            READ COMMITTED |
            REPEATABLE READ |
            SERIALIZABLE };
          ……
COMMIT;
```

- In many DBMSs, the default isolation level is READ COMMITTED
- Isolation level: per transaction and "eye of the beholder"

<u>**Deadlocks**</u>

- Cycle of xacts waiting for locks to be released by each other.

E.g. T1 request S lock on A, T2 request S lock on B, T1 request X lock on B, T2 request X lock on A

→ Solutions? Simple timeout mechanism (not effective), wait-for graph (detection), wait-die/wound-wait (prevention).

Deadlock Detection

→ Wait-For graph

→ Nodes represent active xacts.

→ We add an edge from T2 → T1 if T2 is waiting for T1 to release lock

- Add an edge when a lock request is queued, and the edges are updated dynamically when a lock request is granted. If a cycle is found, there is a deadlock

→ Possibility of livelock

What happens after detection?

→ Break a deadlock by aborting a xact in cycle. We determine the victim either: randomly, finding the most connected node, or workload/time-based (e.g. node that is most recently admitted into the system)

Deadlock Prevention

- Xacts are given a timestamp when they arrive, denoted as ts(Ti) An older Xact has a smaller time stamp

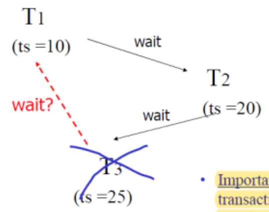
If Tj now requests for a lock that is held by another Ti, then we have 2 possible policies:

Wait-die: Higher priority xact waits if he needs a lock from a lower priority xact. Lower priority xact dies if he needs a lock from a higher priority xact.

Wound-wait: Higher priority xact wounds the lower priority xact if he needs the lock. Lower priority xact waits if he needs the lock from the higher priority xact.

Prevention Policy	T_i has higher priority	T_i has lower priority
Wait-die	T_i waits for T_j	T_i aborts
Wound-wait	T_j aborts	T_i waits for T_j

Wait Die Policy



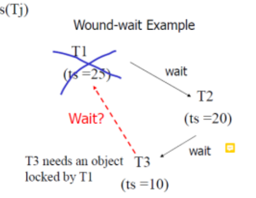
- Important detail: If a transaction re-starts, make sure it gets its original timestamp. Why?

→ If youre the higher priority xact, you wait for the lower priority one to finish if you need his lock. Else you if youre the lower priority xact you die. **However, after you die and restart, you get your original timestamp! Else, you might get starved and keep dying;** eventually youre processed

Wound-Wait Policy

- Ti wounds Tj if ts(Ti)< ts(Tj)
  else Ti waits

"Wound": Tj rolls back and gives lock to Ti



→ If im the higher priority transaction and I need your lock, I will wound you. Else, if im the lower priority transaction I die

→ Transaction that dies will also get original timestamp to prevent starvation

Optimistic & Lock Free Concurrency Control
- Idea: Deadlocks don't happen that much, we don't need so much overhead in dealing with it, just deal with it as it comes.
- Validation-Based Protocol: Breaking down transactions into its 3 phases (Read, validate, write), we use a validation based protocol for CC

(1) Read
- All DB values read/written
  - Make a copy in temporary (local) storage
  - All updates/writes on local storage (no in-place updates/writes)
- No locking
- Maintains *read-set* (RS) and *write-set* (WS)

(2) Validate
- Check if schedule so far is serializable (ensure no conflict between RS/WS sets of transactions)

(3) Write
- If validate ok, write to DB values of WS; if not, "roll-back"

→ Validation: "if i validate this schedule, will it result in non serializable schedule?"

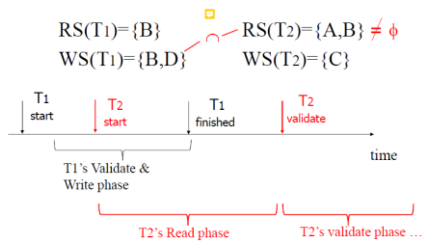
Validation-Based Protocol
→ No dirty reads, no deadlock, no unrepeatable reads, no cascading abort, can starve
3 Timestamps: start(T), validate(T), finish(T)
Start(T): Start of read phase of T
Validate(T): start of validate phase of T
Finish(T): end of write phase

Timestamp of transaction is given by validate(T) for validation based protocol, so, the transactions are serialized based on the timestamp of validation.
If validated, the schedule is:
→ Conflict equivalent to serial order
→ Cascadeless, no locking/deadlocks.
→ **However, rolled back transactions restart with new timestamps and have new validate timestamps**

Validation Tests
- We go in the sequence of validation timestamps. The first xact T1 will always be validated
1. If T2 starts after T1 finishes, we are good.
2. If T1 finishes when T2 is in between start(T2) and validate(T2), then we must check that the write-set(T1) does not overlap with read-set(T2)
3. If T1 finishes after T2 starts and validates, then we must make sure that the write-set(T1) does not overlap with the read-set(T2) and write-set(T2)
→ If any one 1,2,3 is true validation passes! First rule basically says that the schedule is serial, and second/third rule basically says there are no conflicts

Example of what validation must prevent:

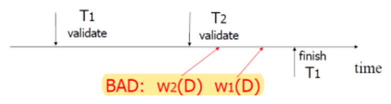
$RS(T_1)=\{B\}$ $\quad RS(T_2)=\{A,B\}$ $\not= \phi$
$WS(T_1)=\{B,D\}$ $\quad WS(T_2)=\{C\}$



→ T1 finish in between T2 starting and validating: Rule 2. Since there is overlap validation of T2 fails

Another thing validation must prevent:

$RS(T_1)=\{A\}$ $\quad RS(T_2)=\{A,B\}$
$WS(T_1)=\{D,E\}$ $\quad WS(T_2)=\{C,D\}$



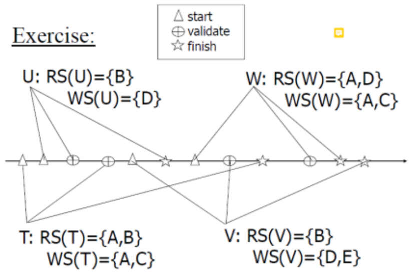
→ Example of Rule 3 where T1 finishes after T2 validates. So, the read and write set of T2 cannot overlap with write set of T1, but as we see, it does so validation fails.

Validation Algorithm
- Algorithm maintains 2 sets: FIN and VAL, where FIN holds all the xacts that have finished, VAL holds all the xacts that have validated
- At each Tj, we finish ignore all the transactions in the FIN set (i.e. they have finished, we don't need to compare them)
- Then, we check if Tj is valid (i.e. Valid(Tj), we talk more about this later). If yes, we add Tj to VAL set, and after write phase, add Tj to FIN set.

Valid(Tj):
- For each transaction Ti that is validated but not finished (i.e. in write phase), we check if our read set overlaps their write set. If Ti is not finished and we are writing and our write set overlaps with their write set, then we return false. Else we return true

```
Valid(Tj):

  For $T_i \in$ VAL - IGNORE ($T_j$) DO
     IF [ $WS(T_i) \cap RS(T_j) \neq \emptyset$ OR        "if they wrote and we read"
          ($T_i \notin$ FIN AND $WS(T_i) \cap WS(T_j) \neq \emptyset$)]
              THEN RETURN false;                   Ti is not done and we both wrote to the same items"
  RETURN true;
```



→ U first to validate, success.
→ T validates and finishes when U is still writing. So we compare RS(T) and WS(T) to WS(U), no overlap, pass.
→ V validates next. U finishes when V is still validating, so we compare WS(U) with RS(V), no overlap. T finishes after V validates. So, we need to compare WS(T) with RS(V) and WS(V). No overlap, V passes
→ W starts after U finishes. T finishes while W is validating. We compare WS(T) with RS(W). Overlap. Fail! Lets look at V also. V finishes after W validates, so we compare WS(W) with RS(V) and WS(V), No overlap.

Concurrency Control For Indexes
Latches: Use latches instead of a heavy-weight lock, where we can unlatch almost immediately. S and X latches for read and writes

B+ Tree Concurrency Control
Problems:
1. Two xacts/threads modify the contents of a node at the same time
2. One xact/thread traversing the tree while another splits/merges nodes

Simple Idea: B+Tree Latching
Latch paths. Base case: We get a latch on the root. From there, get a latch on the appropriate child C of N.
→ However, if we latch this way, no other threads can access the root and we basically locked the entire tree. Poor performance

Latch Coupling ("Crabbing")
- Get a latch on the root.
- Assuming you have a latch on some node N, we can get the latch on the appropriate child C of N, whereby we can release latch on N if "safe" (how do we know if safe?) If not safe then we have to hold the latch.
- A node is safe when it will not be split or merged when updated, i.e. **it is not full for insertion, it is more than half-full for deletion.**

- **Benefits?** Another transaction who might need to access the root and traverse a completely different path can do so

Read and Write for Crabbing / Latch Coupling
- Read
  - Start at root and go down; repeatedly,
    - Acquire R latch on child
    - Then unlatch parent
- Insert/Delete
  - Start at root and go down, obtaining W latches as needed
  - Once child is latched, check if it is safe
    - If child is safe, release all latches on ancestors

→ For insert/delete, we must check if it is safe to unlatch parent before doing so.
→ Reading is always safe.

Any Better B+ Tree Latching Schemes?
- Everytime we insert/delete, we apply a W latch on the root node, which is bad for concurrency. If split/merge operations are uncommon, then maybe we can acquire R latch until leaf level. If we know that we need to split or merge at leaf level, then we perform the original match coupling. We waste the first pass to leaf, but it doesn't happen often anyway
→ if we only split once every 250 insertions then we only repeat once every 250 times

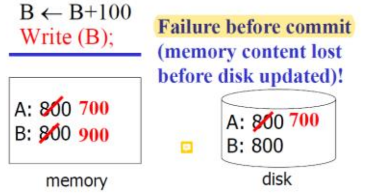
B+ Tree Latching: Deadlocks?
→No deadlocks because every xact traverses from tree top down. So it either acquires latch, **or waits.**
→ However, if a xact traverses from one leaf node to another (front and backwards), there might be deadlocks
→ For latches, we adopt a "no-wait" scheme. If deadlock, All xacts/threads will abort it held and restart

Summary
→ Latch coupling for B+ tree concurrency.
→ If we want to increase CC, we can violate deletion's minimum utilization requirement to do deletion more efficiently!

Log-Based Recovery Scheme
→ We have integrity/consistency constraints to maintain, even if the db instance crashes
E.g. Predicates to satisfy (x is key of relation R, domain(x) = {red, blue, green}…)
→ However, DB may not be always consistent within a transaction. At some point before a commit, there might be instances when the DB is not consistent.
→ What could be the reasons for failure of a DB?
1. Transaction Errors (logical errors/deadlocks)
2. System crash (power failure)
3. Disk failure (Disk head crashes)
→ However, we hope to achieve stable storage and never lose data.

Key Problem: Unfinished Transaction
→ In performing a transaction, usually, we perform operation in memory before it is updated on disk.

```
T1:  Read (A);
     A ← A-100
     Write (A);
     Read (B);
     B ← B+100
     Write (B);
```


Failure before commit (memory content lost before disk updated)!
A: ~~800~~ 700
B: ~~800~~ 900
memory
A: ~~800~~ 700
B: 800
disk

→ Supposed the operation above, where the DB is consistent in memory. But the system crashes before disk is written, memory value is lost and now db will be in an inconsistent state
→ Now, when the system is booted up, we don't know if the original disk value of A was 700 or 800? Was B updated already?

Recovery Manager
→ Guarantee atomicity and durability of Xacts.
Undo: Removes effects of aborted Xact to preserve atomicity
Redo: Reinstall effects of Xacts to preserve durability.
→ Processes 3 operations to maintain ACID properties of DB:
- Commit(T) – install T's updated "pages" into disk
- Abort(T) – restore all data that T updated to their previous values
- Restart – recover DB to consistent state from system failure, abort all active xacts at the time of system failure, install updates of committed xacts.
Ideally, we want it to add little overhead to processing of xacts, and can recover quickly from failure

Questions to answer to achieve Recovery
1. Can a dirty page updated by a xact T be updated to disk before T commits?
If the answer to this is yes, then the recovery manager needs to remember old values of the dirty pages updated by xact T, so that if a system failure happens before commit, it is able to restore the old values in disk.

2. Must all dirty pages written by T be written to disk when T commits?
If the answer to this is yes, then the recovery manager needs to make sure that all dirty pages are flushed to disk.
If the answer is no, then the recovery manager needs to log new values, so that if the new values are not yet updated after a commit, it can perform these updates to ensure durability of xacts

Recovery schemes: Design options

	Force	No-force
Steal	Undo & no redo	Undo & redo
No steal	No undo & no redo	No undo & redo

No-steal policy ⇒ No undo
Force policy ⇒ No redo

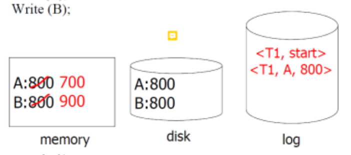
→ Steal means I can write to disk before committing
→ Force means I force flush to disk when commit. Then I don't need to redo

Log Based Recovery
- Logs down the history of actions executed by DB, contains a log record for writes, starts, commits, abort
- Each log has an LSN (Log sequence number)
- Log stored as sequential file of records in stable storage
→ LSN of log record = address of log record
→ Logs usually on separate disk, else everytime we write to logs and to db content, random i/os are incurred

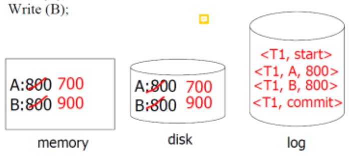
Undo Logging
→ Steal + Force policy – DB is allowed to write updates to disk before commit.
→ However, to ensure consistency, **logs must be written in disk before disk values are updated**
→ In our logs, we save the old value in case we need to recover it

T1: Read (A); A ← A-100
Write (A);
Read (B); B ← B+100
Write (B);

Undo log: <TID, Object, oldValue>

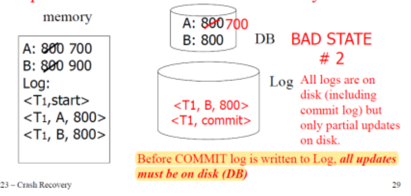


Write (B);



→ Once committed on the logs, it means that all values have been updated and are consistent, and during recovery process, we don't need to undo this transaction
→ **Possible Complication:** If we write everything to our logs before we actually perform the disk update, there might be complications. For example, if we added commit log to the disk without actually writing all the values to disk, then if the system crashes, based on our logs, we don't need to restore the values on disk. The values on disk are now inconsistent

- Updates are not written to disk on every action



Summary of undo logging rules
(1) For every action generate undo log record (containing *old* value)
(2) Before *x* is modified on disk, log record pertaining to *x* must be on disk (write ahead logging: WAL)
(3) Before commit is flushed to log, all writes of transaction must be reflected on disk

Recovery Rules: Undo logging
→ Only look at transactions that have started but not yet committed/aborted. In reverse order, restore all values to disk.
→ Once the values are restored to the old one, add a <Ti, aborted> log

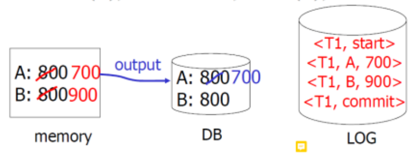
```
(1)Let S = set of transactions with <Ti, start> in log, but
   no <Ti, commit> (or <Ti, abort>) record in log
   • What about those with Commit/Abort?
(2) For each <Ti, X, v> in log,
      in reverse order (latest → earliest) do:
    - if Ti ∈ S then  - { X ← v
                      - { Update disk
(3) For each Ti ∈ S do
    - write <Ti, abort> to log
```

→ What if during recovery process, DB crashes? Its okay, because we can just do the recovery process again, it does not matter (idempotent)

Redo Logging (No steal/No force policy)
→ In a no force policy, when the transaction commits, it is not necessary to flush all the updated values from memory to disk.
→ If the system crashes before the new values are updated to disk, we need the logs to redo the disk writing process
→ Remember the new values in redo logging. If system crashes, we check those transactions that have committed and rewrite the new values saved in the logs to disk.

- **No steal policy:** Updated values in transactions cannot be written to disk until xact commits. This means that updates have to be buffered in memory!

T1: Read(A); A ← A-100; write (A);
Read(B); B ← B+100; write (B);



→ Log file have finished writing all the new values, but DB is not updated yet. If crash at this point, we can take logs of T1 and update DB

Redo Logging Rules
(1) For every action, generate redo log record (containing new value)
(2) Before X is modified on disk (DB), **ALL** log records for transaction that modified X (**including commit**) must be on disk

Redo Logging Recovery Steps

```
(1) Let S = set of transactions with
        <Ti, commit> in log
(2) For each <Ti, X, v> in log, in forward
    order (earliest → latest) do:
    - if Ti ∈ S then { X ← v
                     { Update X on disk
```

→ Redo is also idempotent, so its okay if it fails, just redo again.

Key Drawbacks of Undo Logging and Redo Logging
→ Undo Logging requires more disk i/o, because dirty pages can be written to disk before a commit. I/O is also incurred on writing to logs.
→ Redo Logging: Need to keep all blocks in memory before committing

Solution: Undo/Redo Logging (Steal/No force policy)
→ Store both old value and new value in logs.
→ Steal: You CAN write to disk before commit
→ No force: You don't need to write to disk when you commit.
→ So, if a transaction is not yet committed when crash, you can undo, and if a transaction is committed when crash, you can redo.

→ There is flexibility on when you want to write the new values from memory to disk. You are not restricted to do it before or after committing.

Recover Process: Undo/Redo Logging
- Backwards pass
  - construct set S of committed transactions
  - undo actions of transactions not in S
- Forward pass
  - redo actions of S transactions

Checkpointing
→ For example, when we want to perform redo logging, how do we know where we are going to start from? We need some form of checkpointing to know where to start the redo logging from
→ When you add a checkpoint start, it means that at that point, all dirty pages have been flushed to the disk

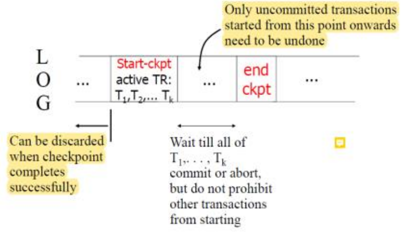
Quiescent Checkpointing

Solution: Checkpoint (simple version)

Periodically:
(1) Do not accept new transactions
(2) Wait until all (active) transactions finish
(3) Flush all log records to disk (log)
(4) Flush all buffers to disk (DB)
(5) Write "checkpoint" record on disk (log)
(6) Resume transaction processing

Non-Quiescent Checkpointing
→ Processing continues in the midst of checkpointing
→ We actually don't need to do this that often because failure doesn't occur so often. By doing less checkpointing we incur less overhead

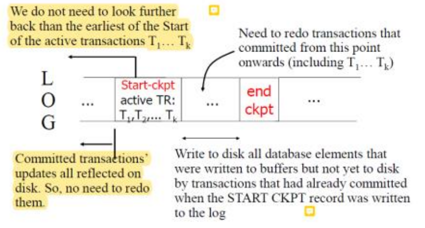
Non-Quiescent Checkpointing: Undo Log



Only uncommitted transactions started from this point onwards need to be undone
LOG
...
Start-ckpt
active TR:
$T_1, T_2, \ldots T_k$
...
end ckpt
...
Can be discarded when checkpoint completes successfully
Wait till all of $T_1, \ldots, T_k$ commit or abort, but do not prohibit other transactions from starting

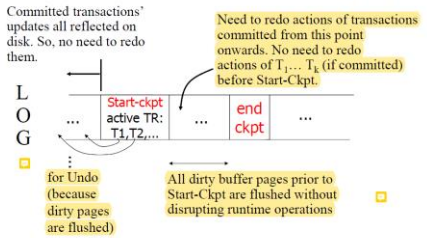
→ For undo logging, we check within our checkpoint all the transactions that have not committed or aborted. We undo those all the way till the start of those transactions.
→ Once we add a checkpoint end, means those active transactions have all committed/aborted. We don't have to undo them if fail. So, if we see a committed transaction it just means 'all updates have been written to disk'.

Non-Quiescent Checkpointing: Redo Log
→ Start a new checkpoint
→ Take all the transactions that were committed from this point onwards and redo them. Once done, add the ending checkpoint.
→ We don't need to redo all the committed transactions BEFORE the start checkpoint, because they were already written to disk.
→ Checkpoint end here means "I am done writing all updates listed at checkpoint start to disk". So, there is the guarantee that updates before checkpoint start have been written to disk. We only write committed transactions to disk.

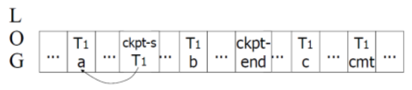


We do not need to look further back than the earliest of the Start of the active transactions $T_1 \ldots T_k$
Need to redo transactions that committed from this point onwards (including $T_1 \ldots T_k$)
LOG
...
Start-ckpt
active TR:
$T_1, T_2, \ldots T_k$
...
end ckpt
...
Committed transactions' updates all reflected on disk. So, no need to redo them.
Write to disk all database elements that were written to buffers but not yet to disk by transactions that had already committed when the START CKPT record was written to the log

Non-Quiescent Checkpointing: Redo/Undo Logging
→ Look at all the transactions that have not been checkpointed
→ Start a checkpoint. We try to write all the updates of the transactions to disk.
→ If an <end checkpt> exists, it means that all updates before <start checkpt> have been flushed to disk.
→ If a crash happens after <end checkpt>, we look at all logs after start checkpoint. If any of these transactions commit, we redo those. Those that did not commit, we undo those.
→ If there is no <end checkpoint>, then there is no guarantee and we need to undo/redo from the previous <start checkpt> <end checkpt> pair onwards.
→ Remember for this, we have a steal/no force policy, so the dirty mem pages may or may not be flushed to disk. So, we just undo all uncommitted and redo all committed
→ Add checkpoint end



Committed transactions' updates all reflected on disk. So, no need to redo them.
Need to redo actions of transactions committed from this point onwards. No need to redo actions of $T_1 \ldots T_k$ (if committed) before Start-Ckpt.
LOG
...
Start-ckpt
active TR:
T1,T2,...
...
end ckpt
...
for Undo (because dirty pages are flushed)
All dirty buffer pages prior to Start-Ckpt are flushed without disrupting runtime operations

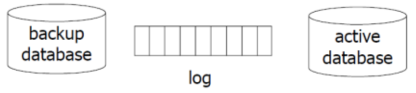


Redo T1: (redo b,c)

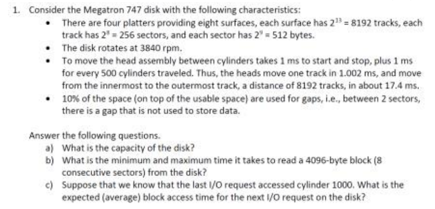
→ T1 committed, since a was before check point, don't need to redo. B and c needs to be redone

Media Failure
→ What if non-volatile storage fails? Make copies of data to prevent failure!
→ **Triple Modular Redundancy:** Keep 3 copies on separate disks. When reading, we can either look at just one of them or compare between all 3 and take the majority value

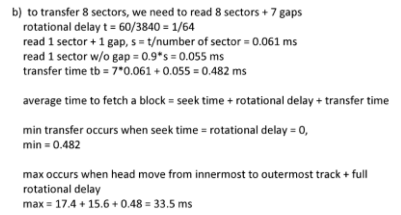
DB Dump + Log
→ Restore active DB from backup database, bring database up to date using redo entries in log file



Block Access Time = Seek Time + Rot delay + transfer time.
→ Average rotational delay = half a round
→ Expected seek time = calculate based on the fact that it is equally likely to travel to every track, calculate average number of tracks we will move
→ Transfer time of 1 block of 8 sectors – if there are gaps, then we are reading 8 sectors and 7 gaps. So calculate 1 without gap and 7 with gap


1. Consider the Megatron 747 disk with the following characteristics:
   - There are four platters providing eight surfaces, each surface has 2^{13} = 8192 tracks, each track has 2^8 = 256 sectors, and each sector has 2^9 = 512 bytes.
   - The disk rotates at 3840 rpm.
   - To move the head assembly between cylinders takes 1 ms to start and stop, plus 1 ms for every 500 cylinders traveled. Thus, the heads move one track in 1.002 ms, and move from the innermost to the outermost track, a distance of 8192 tracks, in about 17.4 ms.
   - 10% of the space (on top of the usable space) are used for gaps, i.e., between 2 sectors, there is a gap that is not used to store data.

   Answer the following questions.
   a) What is the capacity of the disk?
   b) What is the minimum and maximum time it takes to read a 4096-byte block (8 consecutive sectors) from the disk?
   c) Suppose that we know that the last I/O request accessed cylinder 1000. What is the expected (average) block access time for the next I/O request on the disk?

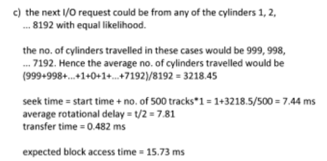

b) to transfer 8 sectors, we need to read 8 sectors + 7 gaps
rotational delay t = 60/3840 = 1/64
read 1 sector + 1 gap, s = t/number of sector = 0.061 ms
read 1 sector w/o gap = 0.9*s = 0.055 ms
transfer time tb = 7*0.061 + 0.055 = 0.482 ms

average time to fetch a block = seek time + rotational delay + transfer time

min transfer occurs when seek time = rotational delay = 0,
min = 0.482

max occurs when head move from innermost to outermost track + full rotational delay
max = 17.4 + 15.6 + 0.48 = 33.5 ms

→ Max rotational delay is 1 full rotation (3840 rpm spin = 0.1562s for 1 spin)
→ Transfer time affected by spin and number of sectors per track. 1 sector + 1 gap is time to spin 1 full rotation / number of sectors


c) the next I/O request could be from any of the cylinders 1, 2, ... 8192 with equal likelihood.

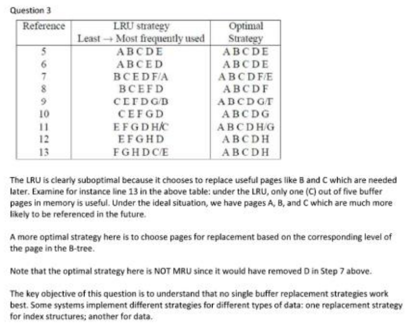
the no. of cylinders travelled in these cases would be 999, 998, ... 7192. Hence the average no. of cylinders travelled would be (999+998+...+1+0+1+...+7192)/8192 = 3218.45

seek time = start time + no. of 500 tracks*1 = 1+3218.5/500 = 7.44 ms
average rotational delay = t/2 = 7.81
transfer time = 0.482 ms

expected block access time = 15.73 ms

→ Calculate expectation based on summing total number of possible moves / number of tracks
→ Qn might ask us to calculate the number of sectors per track by giving us the capacity of the disk (e.g. 10GB), number of platters (e.g. 20 → 512MB per platter), number of cylinders (e.g. 256 → 2MB per track), size of 1 block/sector (e.g. 4096 bytes → 512 blocks/sectors).
→ Transfer rate of disk is proportional to rotation speed (double rot speed, double xfer speed)
→ double number of tracks/number of surfaces will double the number of bits on the disk

Page Replacement Policies


Question 3

Reference	LRU strategy Least → Most frequently used	Optimal Strategy
5	A B C D E	A B C D E
6	A B C E D	A B C D E
7	B C E D F/A	A B C D F/E
8	B C E F D	A B C D F
9	C E F D G/B	A B C D G/F
10	C E F G D	A B C D G
11	E F G D H/C	A B C D H/G
12	E F G H D	A B C D H
13	F G H D C/E	A B C D H

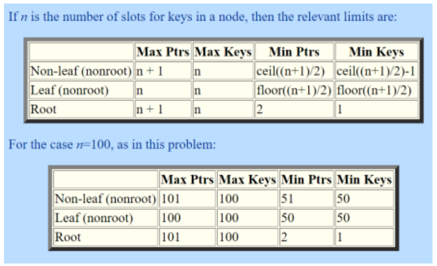
The LRU is clearly suboptimal because it chooses to replace useful pages like B and C which are needed later. Examine for instance line 13 in the above table: under the LRU, only one (C) out of five buffer pages in memory is useful. Under the ideal situation, we have pages A, B, and C which are much more likely to be referenced in the future.

A more optimal strategy here is to choose pages for replacement based on the corresponding level of the page in the B-tree.

Note that the optimal strategy here is NOT MRU since it would have removed D in Step 7 above.

The key objective of this question is to understand that no single buffer replacement strategies work best. Some systems implement different strategies for different types of data: one replacement strategy for index structures; another for data.

→ No one page replacement policy works best, but typically LRU is bad because it will kick the internal nodes (which are accessed frequently) out of the cache


If *n* is the number of slots for keys in a node, then the relevant limits are:

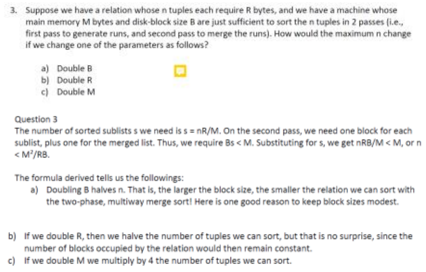
	Max Ptrs	Max Keys	Min Ptrs	Min Keys
Non-leaf (nonroot)	n + 1	n	ceil((n+1)/2)	ceil((n+1)/2)-1
Leaf (nonroot)	n	n	floor((n+1)/2)	floor((n+1)/2)
Root	n + 1	n	2	1

For the case *n*=100, as in this problem:

	Max Ptrs	Max Keys	Min Ptrs	Min Keys
Non-leaf (nonroot)	101	100	51	50
Leaf (nonroot)	100	100	50	50
Root	101	100	2	1

→ Pointers at root do not include the pointer to the adjacent leaf node

Relationship between n, R, M, B


3. Suppose we have a relation whose n tuples each require R bytes, and we have a machine whose main memory M bytes and disk-block size B are just sufficient to sort the n tuples in 2 passes (i.e., first pass to generate runs, and second pass to merge the runs). How would the maximum n change if we change one of the parameters as follows?
   a) Double B
   b) Double R
   c) Double M

Question 3
The number of sorted sublists s we need is s = nR/M. On the second pass, we need one block for each sublist, plus one for the merged list. Thus, we require Bs < M. Substituting for s, we get nRB/M < M, or n < M^2/RB.

The formula derived tells us the followings:
a) Doubling B halves n. That is, the larger the block size, the smaller the relation we can sort with the two-phase, multiway merge sort! Here is one good reason to keep block sizes modest.

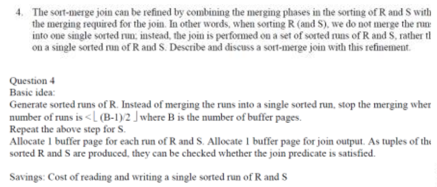
b) If we double R, then we halve the number of tuples we can sort, but that is no surprise, since the number of blocks occupied by the relation would then remain constant.
c) If we double M we multiply by 4 the number of tuples we can sort.

Nested Loop Join Sequential vs Random IOs

At the top level, when we try to retrieve the outer table, it should always be random i/os per block, but within the block, first page is random, subsequent pages are sequential i/os

For the inner table, if for example we are retrieving from a clustered index, then first i/o is random, subsequent is sequential

Combine Merge phase of each table into 1 merge phases in joining 2 tables using sort-merge join

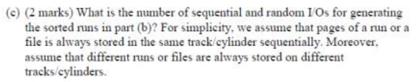

4. The sort-merge join can be refined by combining the merging phases in the sorting of R and S with the merging required for the join. In other words, when sorting R (and S), we do not merge the runs into one single sorted run; instead, the join is performed on a set of sorted runs of R and S, rather th on a single sorted run of R and S. Describe and discuss a sort-merge join with this refinement.

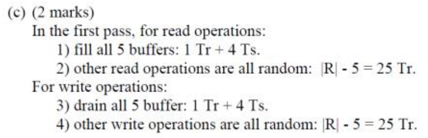
Question 4
Basic idea:
Generate sorted runs of R. Instead of merging the runs into a single sorted run, stop the merging wher number of runs is < ⌊ (B-1)/2 ⌋ where B is the number of buffer pages.
Repeat the above step for S.
Allocate 1 buffer page for each run of R and S. Allocate 1 buffer page for join output. As tuples of the sorted R and S are produced, they can be checked whether the join predicate is satisfied.

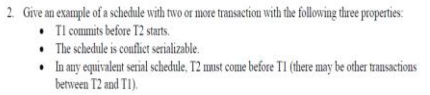
Savings: Cost of reading and writing a single sorted run of R and S

→ Merge each individual table until the number of runs < floor((b-1)/2), then we can move on to the merge phases, where we allocate 1 buffer page for each run of R and S, then start joining.

Replacement Selection Sequential and Random i/os (5 buffer pages, 30 pages in R)

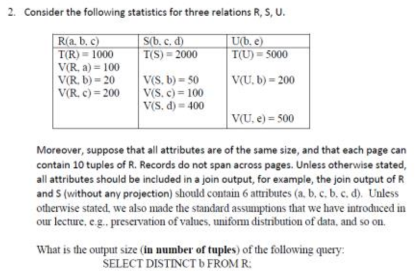

(c) (2 marks) What is the number of sequential and random I/Os for generating the sorted runs in part (b)? For simplicity, we assume that pages of a run or a file is always stored in the same track/cylinder sequentially. Moreover, assume that different runs or files are always stored on different tracks/cylinders.


(c) (2 marks)
In the first pass, for read operations:
1) fill all 5 buffers: 1 Tr + 4 Ts.
2) other read operations are all random: |R| - 5 = 25 Tr.
For write operations:
3) drain all 5 buffer: 1 Tr + 4 Ts.
4) other write operations are all random: |R| - 5 = 25 Tr.


2. Give an example of a schedule with two or more transaction with the following three properties:
   - T1 commits before T2 starts.
   - The schedule is conflict serializable.
   - In any equivalent serial schedule, T2 must come before T1 (there may be other transactions between T2 and T1).

→ w3(A) r1(A) r2(B) w3(B)
serial schedule:
r2(B) w3(A) w3(B) r1(A)

Join with histogram of odd and even values

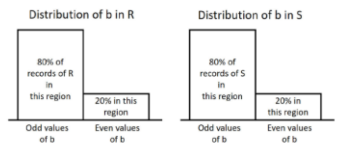

2. Consider the following statistics for three relations R, S, U.

R(a, b, c)	S(b, c, d)	U(b, e)
T(R) = 1000 V(R, a) = 100 V(R, b) = 20 V(R, c) = 200	T(S) = 2000 V(S, b) = 50 V(S, c) = 100 V(S, d) = 400	T(U) = 5000 V(U, b) = 200 V(U, e) = 500

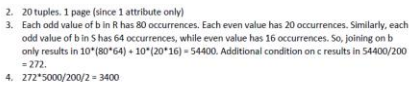
Moreover, suppose that all attributes are of the same size, and that each page can contain 10 tuples of R. Records do not span across pages. Unless otherwise stated, all attributes should be included in a join output, for example, the join output of R and S (without any projection) should contain 6 attributes (a, b, c, b, c, d). Unless otherwise stated, we also made the standard assumptions that we have introduced in our lecture, e.g., preservation of values, uniform distribution of data, and so on.

What is the output size (**in number of tuples**) of the following query:
SELECT DISTINCT b FROM R;

3. Consider the same setting as Question 2. Now, suppose we have the following information on the distribution of the distinct values of b in R and S:

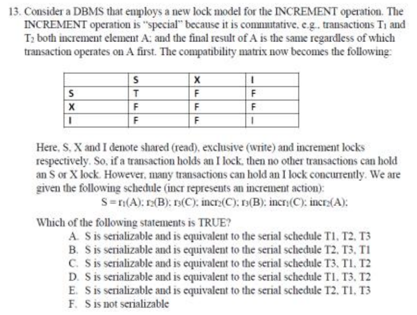


What is the output size (**in number of tuples**) of the following query:
SELECT * FROM R, S WHERE R.b = S.b and R.c = S.c;


2. 20 tuples. 1 page (since 1 attribute only)
3. Each odd value of b in R has 80 occurrences. Each even value has 20 occurrences. Similarly, each odd value of b in S has 64 occurrences, while even value has 16 occurrences. So, joining on b only results in 10*(80*64) + 10*(20*16) = 54400. Additional condition on c results in 54400/200 = 272.
4. 272*5000/200/2 = 3400

2 attribute join

→ if we join by one attribute, say b, then we have r*s/max(ra,sa).
- denote this as RS and size rs.
- we can now do a selection on two columns with the condition R.b=S.b. this is the same as saying R.b=1 and S.b=1 and R.b=2 and S.b=2, ... So, there are min(rb,sb) values. And the size is:
min(rb,sb)*[rs/(rb*sb)] = rs/max(rb,sb).
putting everything together (replacing rs by the first expression), we actually have
r*s/[max(ra,sa)*max(rb,sb)].


13. Consider a DBMS that employs a new lock model for the INCREMENT operation. The INCREMENT operation is "special" because it is commutative, e.g., transactions T_1 and T_2 both increment element A; and the final result of A is the same regardless of which transaction operates on A first. The compatibility matrix now becomes the following:

	S	X	I
S	T	F	F
X	F	F	F
I	F	F	I

Here, S, X and I denote shared (read), exclusive (write) and increment locks respectively. So, if a transaction holds an I lock, then no other transactions can hold an S or X lock. However, many transactions can hold an I lock concurrently. We are given the following schedule (incr represents an increment action):
$S = r_1(A); r_2(B); r_3(C); incr_2(C); r_3(B); incr_1(C); incr_2(A);$

Which of the following statements is TRUE?
A. S is serializable and is equivalent to the serial schedule T1, T2, T3
B. S is serializable and is equivalent to the serial schedule T2, T3, T1
C. S is serializable and is equivalent to the serial schedule T3, T1, T2
D. S is serializable and is equivalent to the serial schedule T1, T3, T2
E. S is serializable and is equivalent to the serial schedule T2, T1, T3
F. S is not serializable

in strict 2pl we release locks after committing, so we know for sure that any other conflicting actions (reads and writes) are only done after the first xact commits, so it is conflict serializable