20.04.2021 (Lecture 1)

Exercise:

Bonus, if 75% of bonus exercises are passed. (every two weeks)
No teams.
Tasks will be published on GitLab → fork project
Don't forget Name and Matriculation Number.
6-7 assignments overall.
Project 1, Sorting:

- file.h for helper-functions
- before writing to file, you need to resize to the desired size
- use make_temporary_file and open_file to circumvent memory restriction
- Deadline: 04.05.2021 15:30

27.04.2021 (Lecture 2)

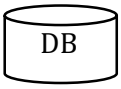
Storage:
- Data Independence (application is decupled from physical storage)
- Scalability (must scale to (nearly) arbitrary sizes)
- Reliability (must cope with hardware and software failures)

Layer-Architecture:
- it's not desirable to implement everything in one block
- rather use smaller tools for each task/layer (levels of functionality)

query layer → query translation and optimization (records)

access layer → managing records and access paths (pages)

storage layer → DB buffer and hardware interface



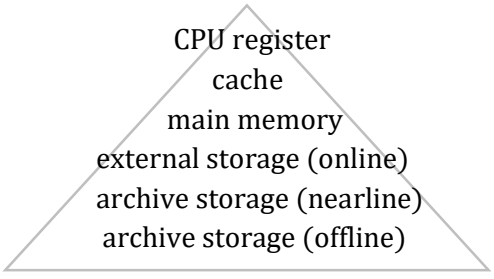
- This keeps the complexity of single layers reasonable.

- Recovery and transaction isolation might be layer-cutting

Hardware:

Moore's Law driving forward different parameters:
- main memory size
- CPU speed (no longer true!)
- HD capacity (density is getting very high. only capacity, not access time)

Memory Hierarchy:



	capacity:	latency:
CPU register	bytes	/ 1ns
cache	K-M bytes	/ <10ns
main memory	G bytes	/ 100ns
external storage (online)	T bytes	/ ms
archive storage (nearline)	T bytes	/ sec
archive storage (offline)	T-P bytes	/ sec-min

→ huge gaps between hierarchy levels:
- main memory vs. disk memory is most important

Hard Disk Access:

Hard Disks are still dominant external storage:
- seek time: 3ms
- transfer rate: 150MB/s
- huge imbalance in random vs. sequential I/O!

DBMS must take these effects into account:
- sequential access is much more efficient
- gap is growing instead of shrinking
- even SSDs are slightly asymmetric (and have other problems)

Techniques to speed up disk access:
- read larger chunks instead of single elements
- typical granularity: one page (4Kb/16Kb)

Page structure is very prominent within the DBMS
- granularity of I/O
- granularity of buffering/memory management
- granularity of recovery

Page is still too small to hide random I/O though
- sequential page access is important
- DBMSs use read-ahead techniques
- asynchronous write-back

Buffer Management:

Some pages are accessed very frequently
- reduce I/O by buffering/caching
- buffer manager keeps active pages in memory
- limited size, discards/write back when needed
- coupled with recovery, in particular logging

Basic interface:
1. FIX (pageNo,shared)
2. UNFIX (pageNo,dirty)

Pages can only be accessed (or modified) when they are **fixed**.

Buffer Frame:

Maintains the state of a certain page with the buffer
- pageNo the page number
- latch a read/write lock to protect the page (must not block unrelated)
- LSN LSN of the last change, for recovers (must force log before write)
- state clean/dirty/newly created, etc.
- data the actual data contained on the page

Buffer Replacement:

When memory is full, some buffer pages have to be replaced

- clean pages can be discarded
- dirty pages have to be written back
- discarded pages are replaced with new pages

We replace pages, which are better than random replacement, but probably not optimal

First In – First Out (FIFO)

- simple replacement strategy

Does not take locality into account

Least Recently Used (LRU)

- similar to FIFO, but frames are kept in a double-linked list (for constant lookup)
- remove from head
- when a frame is unfixed, move it to the end of the list
- "hot" pages are kept in the buffer

Very popular strategy. Latching requires some care.

Least Frequently Used (LFU):

- priority queue
- remembers the number of access per page

Sounds plausible, but too expensive in reality.

LRU is nice, but list might become the hotspot.
Idea: use simpler mechanism to simulate LRU

- one bit per page
- bit is set when page is unfixed
- set bits are cleared during the process
- strategy is called "**second chance**" or "**clock**"

Easy to implement, but a bit crude

2Q:

- many pages are referenced only once
- some pages are hot and referenced frequently
- maintain a separate list for those

1. Maintain all pages in FIFO queue
2. When a page is referenced again that is currently in FIFO move it to LRU
3. Prefer evicting form FIFO

Hot pages are in LRU, read-once are in FIFO

- tries to recognize read-once pages
- but the DBMS knows this anyway
- it could therefore give hints when unfixing
- e.g., *will-need*, or *will-not-need* (changes placement in queue)

Exercise:

Project 1 hints:

- try to only read in blocks
- couple of seconds is ok

04.05.2021 (Lecture 3)

Segments

- most data structures span more than one page (relations, indexes, etc.)
- a set of pages is called segment
- a segment doesn't offer linear address space by default

Mapping:

- static file-mapping (very simple, low overhead | resizing is difficult)
- dynamic block-mapping (maximum flexibility | administrative overhead, indirection)
- dynamic extent-mapping (can handle growth | slight overhead)

Dynamic extent-mapping:

- grows by adding new extents
- growing size of the extents exponentially bounds the number of extents (factor 1.25)
  → catalogue size is bounded

Segment Types:

- private vs. public
- permanent vs. temporary
- automatic vs. manual
- with recovery vs. without recovery

Standard Segments (low-level segments):

Segment inventory:

- keeps track of all pages allocated to segments
- keeps extent lists or page tables or …

Free space inventory:

- keeps track of free pages
- maintains bitmaps or free extents or …

High-level segments:

- schema
- relations
- temp segments (created and discarded on demand)
- index structures

Update Strategies:

Dimensions: steal & force

- usually preferred: steal, not force
- but then pages contain dirty data
- complicates recovery

Shadow Paging:

- dirty pages are story in a shadow copy

Advantages:

- clean data is always available on disk
- greatly simplified recovery
- can be used for transaction isolation

Disadvantages:

- complicates the page access logic
- destroys data locality

Delta Files:

- on change, pages are stored to a separate file
- a copied page can be changed in-place
- on commit discard the file, on abort, copy back

Two flavors:

- store a clean copy in delta
- store the dirty data in delta

Advantages:

- preserves data locality
- no mixture of clean and dirty pages

Disadvantages:

- more I/O
- abort (or commit) becomes expensive
- keeping track of delta pages is non-trivial

Still often preferable over shadow paging

Access Paths:

Needed data structures:

- free space management
- data itself
- unusually large data
- index structures to speed up access

Free Space Inventory:

Problem: where to we have space for incoming data?

Solution: Traditional free space bitmap (one nibble (half-byte) per page)

→ encode fill status in 4 bits

- data_size/ (page_size/16) loss of accuracy in lower range
- ceil(log_2(free_size)) logarithmic scale is often better

→ combination (logarithmic for lower, linear for upper)

Insertion:

compute required FSI entry -> scan FSI -> insert

Problem:

- linear effort
- FSI is small, for 16KB pages, 1 FSI page covers 512MB
- but scan is still not free
- only 16 FSI value, cache next matching page (range)
- most pages will be static (and full anyways)

- segments will mostly grow at the end
- cache avoids scanning most of the FSI entries

Allocation:

Allocating page benefits from application knowledge
- often larger pieces are inserted soon after each other
- e.g. a set of tuples
- or one very large data item
- should be allocated close to each other

Allocation interface is usually `allocate(min,max)`
- `max` is a hint to improve data layout
- some interfaces (e.g., segment growth) even implement over-allocation
- reduces fragmentation

Exercise:

Project 2: Buffer Manager
- Implement a buffer manager that uses the 2Q replacement strategy
- `Buffer Manager` class with `fix(page_id, exclusive)` and `unfix(page_id,dirty)`
- `page_id` is split into segment(16bits) and segment page(48bits)
- don't hold latches while doing I/O

11.05.2021 (Lecture 4)

Slotted Pages:
- put it in, if it fits, else create a slot and save the address of the page, where it's stored
- "forward" points towards the new location of the overflow record
- mostly only created, when size of an element is changed (updated), which is rare
- the chain will never get longer than two, instead, the forward" will be changed
- the link is only followed, when we come from an index. Not, if page is scanned

Tuples:

Page: header | slots -> … <- data
- Data grows from one side, slots from the other
- Problems: - updates/deletes complicate issues.
  - might require garbage collection/compactification

Header: (can be delayed until needed)

LSN:	for recovery
slotCount:	number of used slots
firstFreeSlot:	to speed up locating free slots
dataStart:	lower end of the data
freeSpace:	space that would be available after compactification

$$\text{effectiveFreeSpace} = \text{PageSize} - \text{HeaderSize} - \text{SlotCount} \times \text{SlotSize} - (\text{PageSize} - \text{DataStart}) \leq \text{freeSpace}$$

Slot:

offset: start of the data item
length: length of the data item
Special Cases:
free slot: offset $= 0$, length $= 0$
zero-length data item: offset > 0, length $= 0$

Problem:

1. transaction T_1 updates data item i_1 on page P_1 to a very small size (of deletes it)
2. transaction T_2 inserts a new item i_2 on page P_1, filling up P_1
3. transaction T_2 commits
4. transaction T_1 aborts (or T_3 updates i_1 again to a larger size)

TID concept $\Rightarrow$ create an indirection
but where to put it? Would have to move i_1 and i_2.

Logic becomes much simpler if TID could be stored inside the slot
- borrow a bit from the TID (or have some other way to detect invalid TIDs)
- if the slot contains a valid TID, the entry is redirected
- otherwise, it is a regular slot
- this way, the slot itself gives space for the "forward"

Possible slot implementation:

| T | S | O | O | O | L | L | L |
1. if T ≠ 0xFF, the slot points to another record
2. otherwise, the record is on the current page
   2.1 if S $= 0$, the item is at offset O, with length L
   2.2. otherwise, the item was moved from another page
      - it is also placed at offset O, with length L
      - but the first 8 bytes contain the **original** TID

The original TID is important for scanning

Record Layout:

Possibility: serialize the attributes (Problem: accessing an attribute is $\mathcal{O}(n)$)
Solution:
- split tuple into two parts
- fixed size header and variable size tail
- header contains pointers to the tail (end of attribute is stored)
- allows for accessing any attribute in $\mathcal{O}(n)$

for performance reasons attributes should be reordered:
- split string into length and content
- reorder by decreasing alignment
- place variable length data at the end
- variable length has alignment 1

This gives better performance without wasting space on padding

What about NULL values?

- represent an unknown/unspecified value
- is a special value outside the regular domain

It's a good idea to declare columns and not-nullable

Multiple ways to store it:

- either pick an invalid value (e.g. INTMIN) (not always possible)
- or use a separate NULL bit

NULL bits allow for omitting NULL values from the tuple

- complicates the access logic
- but saves space
- useful if NULL values are common

Some DBMSs apply compression techniques to the tuples

- most of the time, compression is **not** added to save space!
- disk is cheap after all
- compression is used to **improve performance**!
- reducing the size reduces the bandwidth consumption

Some people really care about space consumption, of course.

But outside embedded DBMSs it is usually an afterthought.

Compression:

What to compress?

- the larger the data compressed chunk, the better the compression
- but: DBMS has to handle updates
- usually rules out page-wise compression
- individual tuples can be compressed more easily

How to compress?

- general purpose compression like LZ77 too expensive
- compression is about performance, after all
- most systems use special-purpose compression
- byte-wise to keep performance reasonable

Integer variable length encoding (add two bits to specify length):

- but now the integers are variable size
- store 4 length indicators as one byte in the fixed size part

Dictionary compression:

- stores strings in a dictionary
- stores only the string id in the tuple
- factors out common strings
- can greatly reduce the data size
- can be combined with integer compression

Exercise:

Happy Path:

1. Lock directory
2. Get frame from directory (careful! I/O -> release lock)
3. Lock frame (careful! Deadlocks)
4. Unlock directory

Best dictionary is Hash-Map (directory = dictionary that maps frame_id to frame)

- guarded by global mutex

Evict Frames:

1. Find page to evict
2. Try to lock the page (or restart)
3. If page is clean -> evict
4. Unlock directory
5. Write data to disk
6. Mark as clean
7. Lock directory
8. If no concurrent access wants to keep this page -> evict (or restart)
9. Remove from directory
10. Unlock

Tipp for exam:

use wolfram alpha to compute "`sum 1.25^n, n=1 to 24 (bytes)`"

18.05.2021 (Lecture 5)

Long Records:

- a tuple must fit into a single page (this limits the size of a tuple)
- what about large tuples? (SQL: BLOB/CLOB)

→ requires some mechanism to handle these large records

Spanning pages is not a good idea:

- complicates buffering
- updates that change size are complicated
- intermediate results during query processing

Instead, keep the main tuple size down:

- BLOBS/CLOBS are stored separate from the tuple
- tuple only contains a pointer
- increases the cost of accessing a BLOB but simplifies tuple processing

BLOBs can be stored in a B-Tree-like fashion:

- (relative) offset is search key
- allows for accessing and updating arbitrary parts
- very flexible and powerful
- but might be over-sophisticated
- SQL does not offer this interface anyway (growing or shrinking not supported)

Using an extent list is simpler:

- real tuple points to BLOB tuple
- BLOB tuple contains a header and an extent list
- in worst case the extent list is chained, but should rarely happen
- extent list only allows for manipulating BLOB in one piece
- but this is usually good enough
- hash and length to speed up comparisons

It makes sense to optimize for short BLOBs/CLOBs:

- users misuse BLOBs/CLOBs
- they use CLOB to avoid specifying a maximum length in varchar
- but most CLOBs are short in reality
- on the other hand, some BLOBs are really huge
- the DBMS cannot know
- so BLOBs can be arbitrarily large, but short BLOBs should be more efficient

Approach:

1. BLOBs smaller than TID are encoded in BLOB TID
2. BLOBs smaller than page size are stored in BLOB record
3. Only larger BLOBs use the full mechanism

Index Structures (B-trees):

Data is often indexed:

- speeds up lookup
- de-facto mandatory for primary keys
- useful for selective queries

Two important access classes:

- point queries: find all tuples with a given value (might be a compound)
- range queries: find all tuples within a given value range

Support for more complex predicates is rare.

B-Tree:

B-Trees (including variants) are the dominant data structure for external storage.

Classical definition:

- a B-Tree has a degree k
- each node except the root has at least k entries
- each node has at most $2k$ *entries*
- all leaf nodes are at the same depth

depth_tree $\leq \log_k(|\text{entries}|)$

$k = 1000$, since we want to be able to store one node per page ($2k$ entries)

RB-Tree vs. B-Tree:

B-Trees usually don't have a depth of more than 3-4 (k = 1000)
RB-Trees have a depth of around 20-40
→ much faster lookup time in B-Trees (even though single steps are slower)

B^+-Tree:

- k + TID only in leaf nodes
- inner nodes contain separators, might or might not occur in the data
- increases the fanout of inner nodes
- simplifies the B-Tree logic

Question: page 86 he said we could insert 49 behind 47, but then the node would have over 2k entries, so it wouldn't fit on a single page anymore…

Page Structure:

Inner Node:

- LSN — for recovery
- upper — page of right-most child
- count — number of entries
- key/child — key/child-page pairs
- … — …

Leaf Node:

- LSN — for recovery
- ~0 — leaf node marker (helps identifying leaf-nodes)
- next — pointer to the next leaf-node (improves range queries)
- count — number of entries
- key/tid — key/TID pairs
- … — …

Operations - Lookup:

Lookup a search key within the B^+-Tree:

1. start with the current root node
2. is the current node a leaf?
   → if yes, return the current page (locate the entry on it)
3. find the first entry ≥ search key (binary search)
4. if no such entry is found go to the *upper*,
   otherwise go to corresponding page
5. Continue with 2

Lookup can return concrete entry or just the position on the appropriate leaf page (depends on usage pattern).

Exercise:

Slotted pages:

free space inventory:

- find suitable pages more easily

TID redirect:

- only ever one redirection

Schema:

01.06.2021 (Lecture 6)

Operations – Insert:

Insert a new entry into the B^+-tree:

1. lookup the appropriate leaf page
2. is there free space on the leaf?
   → if yes, insert entry and stop
3. split the node into two, insert entry on proper side
4. insert maximum of left page as separator into parent
5. if the parent overflows, split parent and continue with 3
6. create a new root if needed

Operations – Delete:

Remove an entry from the B^+-tree:

1. lookup the appropriate leaf page
2. remove the entry from the current page
3. is the current page at least half full?
   → if yes, stop
4. is the neighbouring page more than half full?
   → if yes, balance both pages, update separator, and stop
5. merge neighbouring page into current page
6. remove the separator from the parent, continue with 3

Most systema simplify the delete logic and accept under-full pages

Operations – Range Scan:

Read all entries within a (start,stop(range

1. lookup the start value
2. enumerate subsequent entries on the current page
3. use the next pointer to find the next page
4. stop once the stop value is reached

Very efficient, in particular if leaf nodes are consecutive on disk.

Indexing Multiple Attributes:

Compound keys are compared lexicographically:

$$(a_1, a_2) < (b_1, b_2) \Leftrightarrow (a_1 < b_1) \vee (a_1 = b_1 \wedge a_2 < b_2)$$

Otherwise compound keys are quite similar to atomic keys.

- when all attributes are bound, difference is minor
- if only a prefix is bound, the suffix is specified as range

$$a_1 = 5 \Rightarrow (5, -\infty) \leq (a_1, a_2) \leq (5, \infty)$$

Indexing Non-Unique Values:

Users can create indexes on any attributes

- not necessarily unique
- in face might contain millions of duplicates
- main problem: index maintenance
- how to locate a tuple for update/delete?

Solution: only index unique values

- append TID to non-key attributes
- TID works as tie-breaker
- increases space consumption a bit
- but guarantees $\mathcal{O}(\log n)$ access.

Concurrent Access:

How to handle concurrent access?

- simple page locking/latching is not enough
- will protect against "simple" (single page) changes
- but pages depend upon each other (pointers)

The classical technique is *lock coupling*

- a thread latches both the page and its parent page
- i.e., latch the root, latch the first level, release the root, latch the second level etc.
- prevents conflicts, as pages can only be split when parent is latched
- no deadlocks, as the latches are ordered

But what about inserts?

- when a leaf is split, the separator is propagated up
- might go up to the root
- but we only have locked the parent

Lock coupling is not an option (deadlocks)

One way around it: "safe" inner pages

- while going down, check if the inner page has enough space
- if not, split it
- ensures that we never go up more than one step

Alternative: restart

1. first try to insert using simple lock coupling
2. if we do not have to split the inner node everything is fine
3. otherwise release all latches
4. restart the operation but now keep all latches up to the root
5. all operations can be executed safely now

- greatly reduces concurrency
- but should happen rarely
- simpler to implement, in particular for variable-length keys

B-link Trees:

- lock coupling latches two nodes at a time
- seems cheap, but effectively it locks hundreds (all children of the parent node)
- if would be nicer to lock only one page

For pure lookups that is possible when adding *next* pointers to inner nodes:

1. latch a page, find the child page, release the page
2. latch the child page
3. might have been split in between, check neighbouring pages

Requires some care when deleting.

Optimistic Lock Coupling:

- Problem: root is a bottleneck (bad for concurrency)
- Even when shared (read-locks are a problem in practice)
- but: conflicts are unlikely

→ optimize for case without conflict

Optimistic lock: Exclusive Bit | Version (instead of counter)

Also: initially only acquire read-locks on inner nodes. (restart if necessary)

Optimistic Read:

1. don't lock but make a local copy C_0 of the lock L_0
2. restart, if lock is changed to exclusive in the meantime (rare)
3. binary search on page (no locking)
4. validate C_0 and restart if version is not the same (or exclusive)
5. read L_1 and copy into C_1
6. binary search on page
7. validate C_0 and C_1 and restart if necessary
8. forget C_0

…

Usually not used for range queries, since the longer the query runs, the more likely it is, that the validation fails.

Bulkloading:

How to build a B-tree for a large amount of data?

- repeated inserts are inefficient
- a lot of random I/O
- pages are touched multiple times

Instead: *sort* the data before inserting

- now inserts become more efficient
- goo locality

But we can do even better.

To construct an initial B^+-Tree:

1. sort the data
2. spool data into leaf pages
   → fill the pages completely
   → remember largest value (separator) in each page in a temp file
3. Spool the separators into inner pages
   → fill the pages completely
   → remember largest value (separator) in each page in a temp file
4. Repeat 3 until only one inner page remains (root)

Produces a compact clustered B^+-tree

Existing B^+-trees are a bit more problematic but can still be updated:

1. sort the data
2. merge the data into the existing tree
3. form pages, remember separators, etc.
4. start a new chunk once a page would contain only entries from the original tree
5. merge in the separators above

Minimizes I/O but destroys clustering. Usually, a good compromise.

Partitioned B-Tree:

Bulk operations are fine if they are rare, but they are disruptive

- usually the B-tree has to be taken offline
- the new cannot be queried easily
- existing queries must be halted

Basic idea: *partition* the B-tree

- add an artificial column in front
- creates separate partitions with the B-tree

Benefits:

- partitions are largely independent of each other
- one can append to the rightmost partition without disrupting the rest
- the index stays always online
- partitions can be merged lazily
- merge only when beneficial

Drawbacks:

- no "global" order any more
- lookups have to access all partitions
- deletion is non-trivial ("anti-matter")

Variable Length Records:

So far, B-trees are defined for fixed-length keys

- all nodes have between k and 2k entries
- simplifies life considerably
- e.g., we "know" if an inner node is full

But in reality, entries can be variable length

- strings
- variable-length encoding, NULL
- compound of those

Usually, keys are *opaque*. The B-tree does not understand the structure. (One could special-case single strings)

Variable length keys are problematic.
example: alternating long and short keys
→ leads to long keys being separators
→ leads to a deep tree
⇒ we want short separators

Separator choice is crucial!
- affects fanout and space consumption
- all standard guarantees are off, if normal algorithms are used for variable length keys

Non-trivial issue
- greedy algorithms exist for bulkloading case
- idea: recursively pick the smallest separator such that the resulting group sizes vary within a constant factor
- can also be extended to the dynamic case (rebuild as need) but not trivial
- difficult, but gives good amortized bounds

Minimal support: modify the split logic
1. when a page overflows, build the sorted list of values
2. instead of picking the median as separator, pick the smallest value within 20% around the median
3. for ties, prefer value closer to the median

- pragmatic solution, but not optimal
- can still degenerate
- avoids worst mistakes

Exercise:

Indexing: B+-tree
Use buffer-manager to solve concurrency
- lock coupling
- safe inner nodes/ restarting

Lower Bound should have logarithmic search times

08.06.2021 (Lecture 7)

Prefix B^+-tree:

B^+-trees can contain separators that do not occur in the data.
We can use this to save space:
- use small artificial keys in inner nodes
- smallest possible separator to separate leaf nodes

Problem:
How to do this for arbitrary data types?
(Unicode collate → weight per symbol (not reversible))

Solution:

- Factor out common prefix (e.g., "http://www." for websites)
- But strings have to be reconstructed when reporting back
- Only one prefix per page
- the change to the lookup logic is minor
- the lookup key itself is adjusted
- sometimes only inner nodes, to keep scans cheap

The lexicographic sort order makes prefix compression attractive:

- neighbouring entries tend to differ only at the end
- a common prefix occurs very frequently
- not only for strings, also for compound keys
- in particular important in partitioned B-trees
- with big-endian ordering any value might get compressed

Big Endian storage is attractive for numerical entries

Index Structures (Hash-Based):

Example: 1Mil. Entries

Lookup:

RB-Tree:	depth 20-40	⇒	≈ 30 elements per lookup
B-Tree:	depth 2-3	⇒	≈ 21 elements per lookup
Hash-Table			1-2 elements

But hash-tables have a very bad worst case.

"You have to trust the statistics."

In main memory a hash table is usually faster than a search tree:

- compute a hash-value h, compute a slot
- promise $\mathcal{O}(1)$ access (if everything works out fine)

A DBMS could benefit from this, too, but:

- random I/O is very expensive on disk
- collisions are problematic
- re-hashing is prohibitive

But there are hashing schemes for external storage.

Implementations:

Chaining:

Every entry has a pointer to the next entry in the chain

Lookup: Iterate over chain until found

Problems:

- how large should n be? ("elements + a bit more" , ~70% full)
- modulo hashing is bad and you need prime n (adversary, …)
- modulo is computationally slow
- how to implement hash function? (uniform & independent)

Possibilities:

- make modulo cheap (magic numbers)
- you could use $n = 2^k$
  - fast!
  - fixed sizes for table & requires good hash function

Trick:

- use 16bits of 64bits to store unused hash-bits
- then, if you have a miss, you only have to look at one entry

Robin-Hood-Hashing:

If entry is taken, go to next entry

Make sure distance doesn't get too big:

- if distance is higher than of the current value in entry
  → move current entry instead of new

Hash Indexes are not as versatile as tree indexes:

- only support point query (no range query (very problematic))
- order preserving hashing exists, but is questionable
- quality of the hash function is critical

Consequently, mainly used for primary key indexes

- unique keys
- key collisions would be very dangerous
- how to delete a tuple with an indexes attribute if there are 1mil. other tuples with the same value?
- can be fixed by separate indexing within duplicate values (complicated)

Extendible Hashing:

A central problem of hashing schemes on disk is the table size

- hard to know beforehand
- too small ⇒ too many collisions
- chaining is expensive on disk
- too large ⇒ waste of space
- would have to grow over time
- chaining is bad, since we don't want to jump from bucket to bucket

Traditional solution for main memory: *rehashing*

- re-map items to hash table sizes
- involves touching every item
- poor locality, a lot of random I/O
- prohibitive for disk

Idea: Allow for growing the hash table without rehashing by *sharing* table entries.

- hash table size is always power of 2
- hash table points to buckets
- multiple table entries can point to the same bucket
- but always systematically (buddy system)
  elements in S are neighbouring, #S is power of 2, last log2(#S) bit is matching
- thus, the "depth" of the buckets varies

- when a bucket overflows, it is split
- if not at maximum depth, the depth is increased
- achieved by de-sharing slot entries
- one more bit becomes relevant
- items are distributed according to hash values

- if the depth cannot be increased, the table is doubled
- other buckets are unaffected
- entries are duplicated, resulting in new sharing
- new buckets are linked as usual

- once maximum table size is reached start chaining
- ideally occurs rarely, traversing chains is expensive
- optionally one can balance chaining vs. table growth via load factor

Advantages:

- ideally exactly two page accesses per lookup
- less than in a B-tree
- table can grow independent of existing buckets
- no need for re-hashing

Disadvantages:

- table growth is a very invasive operation
- large steps in space consumption
- what about hash collisions?
- same care is needed to avoid extreme table growth

Exercise:

Lock-Coupling + eagerly splitting
Critical Path: it is possible to lock only part of the pages

Sheet 3:

Record Layout:

- idea: fixed in the front, variable length in the end (with fixed pointer)
- reorder corresponding to alignment

Varchar(128); Varchar(128); int; smallint; int; bigint (primary key)

matriculation_no	8
major_program	4
minor	4
name	2
email	2
semester	2
nulls	1 (for minor) (this is usually the best way)
<name>	
<email>	

$2 + 2 + 2 + 4 + 4 + 8 + 1 = 23$ ← fixed part
$16 + 16 = 32$ ← variable part

Extendible Hashing:
X : split marker

A: 00101<u>0</u>	→	0
B: 011<u>001</u>	→	grow table
	→	A : 1 0 (001<u>010</u>)
	→	B : X 1
C: 011<u>110</u>	→	A : 1 0
	→	B : 0 1
	→	C : 1 0 (chained)
D: 010<u>010</u>	→	grow table
	→	B : X 0 1
	→	A, C, D : X 1 1
E: 101<u>000</u>	→	E : 000
	→	B : 001 (011<u>001</u>)
	→	A, C, D : X 1 1

Answers:

1. 8	(one larger than the data)
2. $2^4 = 16$	(4 bits used for table)

15.06.2021 (Lecture 8)

Linear Hashing

- avoids the exponential directory growth
- it starts with a regular hash table with buckets

- when a bucket overflows, it uses chaining
- degrades performance but ok, if chains are short

- triggered by load factor or chain length a bucket split
- chains are re-integrated into the bucket
- only one (i.e., the next) bucket is split, the rest remains untouched
- the range of buckets $[1, k[$ has been split, the range $[k, n[$ is unsplit (the range $[n, 2n + 2k - 2[$ contains the second halves)
- the directory grows page by page

- more buckets are split on demand
- at some point, all buckets have been split once
- then the cycle starts anew

Advantages:

- avoids the disruptive directory growth of EH
- index grows linearly
- amortized the index structure is nice

Disadvantages:

- chaining hurts performance
- it can take a while until chains are re-integrated
- page allocation for the directory is problematic

Multi-Level Extendible Hashing:

For uniform distributions, the EH directory is nice.
But data skew causes poor space utilization.
(many pointers point to the same bucket)
The directory size explodes.

- skew is an unfortunate reality

Basic idea: construct a tree of hash tables

- the next level uses the next k bits
- node size is page size
- additional page faults, but fanout is very large
- only the heavily used parts get additional levels

Problem: space utilization

- now, uniform is the worst case
- all buckets will overflow at the same time
- second-level hash tables will be nearly empty
- leads to poor space utilization (and poor performance)

Instead: share inner page between buddies

We can get additional bits by splitting the inner page
In fact, here we can even move buddies back up again

- uses a buddy system (boundaries are powers of 2)
- bit width etc. derived implicitly
- pointer structure contains enough information
- results in large fanout

It has some nice properties:

- naturally adapts to data skew
- directory growth is not a problem
- but additional page accesses
- tree is very shallow, however

Downside: complicated to implement

Bitmap Indexes:

Classical indexes do not handle unselective predicates very well

- large fraction of tuples is returned
- index access is more expensive than table scan
- but a combination of predicates might be selective
- index intersection would help, but is still expensive
- predicates might also contain disjunctions etc.

Example: $\sigma_{(a=2 \vee b=5) \wedge (c \neq 1)}(R)$

Could be answered by B-trees, but often not very efficient.

When the attribute domain is small, it can be indexed using
bitmap indexes

- one bitmap for every attribute value
- index intersections become bit operations
- very efficient
- remaining ones indicate matching tuples

- bitmap indexes are compact (one bit per tuple)
- (plus tid directory is needed)
- and are usually sparse
- can be compressed very well
- run-length encoding in particular attractive
- intersection can be performed in compressed form

Often outperforms other indexes for unselective attributes.

Small Materialized Aggregates

- data is usually stored in physical chunks
- pages, or chunks of pages
- clustering usually in insertion order
- older chunks tend to remain static
- we can cheaply pre-compute (i.e., cache) some info

Some useful aggregates:
min, max, sum, count

If, still, an update comes, mark the aggregates as invalid.
Re-compute the aggregates when the next query looks at the tuples, anyways.

Before scanning a chunk, we examine the aggregate:
- some predicates can be evaluated on the pure aggregate
- only for the current chunk, of course
- in particular skipping chunks is often possible
- aggregates can be directly re-used
- greatly saves I/O

What about updates?
- small materialized aggregates are like cache
- can be updated eagerly or lazily
- an invalidation flag is enough

Multi-Dimensional Indexing:

- huge field
- R-tree, grid file, pyramid schema, index intersection, …
- hundreds of approaches
- we do not discuss them here

But: remember the **curse of dimensionality**
- we can only index a relative low number or dimensions
- for higher dimensions, range queries/proximity queries fail
- scan becomes faster than index structures

Transactions and Recovery:

DBMSs offer two important concepts:
1. transaction support
   - a sequence of operations is combined into one compound operation
   - transactions can be executed concurrently with well defined semantics
2. recovery
   - the machine/DBMS/user code can crash at an arbitrary point in time, errors, etc.
   - the recovery component ensures that no (committed) data is lost (consistency)

Implementation of both is intermingled, therefore we consider them together.

Why Transactions?
Example: Bank Account (never lose money)

Operations:

- **begin of transaction (BOT)**
  - marks the begin of transaction
  - in SQL: `begin transaction`
  - often implicit
- **commit**
  - terminates a successful transaction
  - in SQL: `commit [transaction]`
  - all changes are permanent now
- **abort:**
  - terminates an unsuccessful transaction
  - in SQL: `rollback [transaction]`
  - undoes all changes performed by the transaction
  - might be triggered externally

All transactions either commit or abort.

ACID:

- Atomicity
- Consistency
- Isolation
- Durability

The concept of recovery is related to the transaction concept:

- the DBMS must handle a crash at an arbitrary point in time
- first, the DBMS data structures must survive this
- second, transaction guarantees must still hold
- Atomicity
  - in-flight transactions must be rolled back at restart
- Consistency
  - consistency guarantees must still hold
- Durability
  - committed transactions must not be lost, even though data might still be in transient memory

Sometimes the dependency is mutual

- Isolation
  - some DBMS use recovery component for transaction isolation

Technical Aspects:

The logical concept transactions and recovery can be seen under (largely orthogonal) technical aspects:

- concurrency control (I,C)
- logging (A,D)

As we will see, both are relevant for both logical concepts.

Multi User Synchronization

- executing transactions (TA) serialized is safe, but slow
- transactions are frequently delayed (wait for disk, user input, …)
- in serial execution, would block all other TAs
- concurrent execution is desirable for performance reasons

But: simple concurrent execution causes a number of problems:

- Lost Update (write of T1 is lost, because T2 read before and wrote after T1)
- Dirty Read (T1 works with value of T2, but T2 aborts)
- Non-repeatable Read (T1 reads twice, but values differ because of T2's write)
- Phantom Problem (T1 selects twice, but finds new tuple because of T2's insert)

Serial Execution:

These problems vanish with *serial* execution

- a transaction always controls the whole DBMS
- no conflicts possible
- but poor performance

Instead: execute transactions as if they were serial

- if they behave as if they were serial, they cause no problems
- concept is called *serializable*
- requires some careful bookkeeping

Exercise:

Lookup needs lock-coupling, as well.
memmove for moving elements in an array.

4 Bonus Points:

Intersect
IntersectAll
Except
ExceptAll

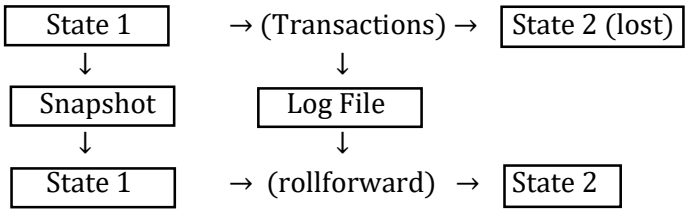
22.06.2021 (Lecture 9)

Recovery:

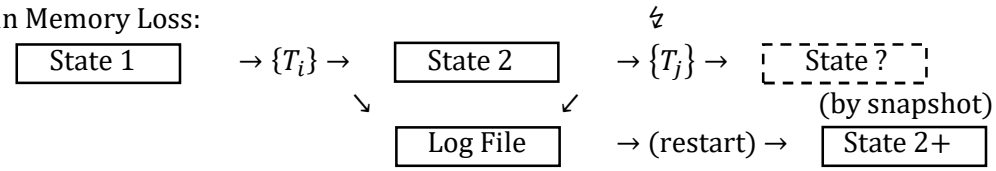
- a DBMS must not lose any data in case of a system crash
- main mechanisms of recovery:
  - database snapshots (backups)
  - log files

- a *database snapshot* is a copy of the database at a certain point in time
- the *log file* is a protocol of all changes performed in the database instance
- obviously the main data, the database snapshots and the log files should not be kept in the same machine…

System Failure:



Main Memory Loss:



- problem: some TAs in $\{T_j\}$ were still active, some committed already
- restart reconstructs state 2 + all changes by committed by TAs in $\{T_j\}$

Aborting a Transaction:

- log files can also ne used to undo the changes performed by an aborted TA
- the functionality is needed anyways (system crash)
- can be used for "normal" aborts, too

Classification of Failures:

- local failure within non-committed transaction
  - effect of TA must be undone
  - *R1* recovery
- failure with loss of main memory
  - all committed TAs must be preserved (*R2* recovery)
  - all non-committed TAs must be rolled back (*R3* recovery)
- failure with loss of external memory
  - *R4* recovery

Storage Hierarchy:

DBMS Buffer vs. External Memory

- Replacement strategies for buffer pages
  - ¬*steal:* pages that have been modified by active transactions must not be replaced
  - *steal:* any non-fixed pages can be replaced if new pages have to be read in
- write strategies for committed TAs
  - *force:* changes are written to disk when TA commits
  - ¬*force:* changed pages may remain in buffer and are written back at some later point in time

simplest: *force* / ¬*steal*

but we want: ¬*force* / *steal* (for performance)

Update Strategies:

- Update in Place
  - each page corresponds to one fixed position on disk
  - the old state is overwritten
- twin-block approach
  - two positions per page
  - one is in the committed state
  - when we are done, the roles of the pages are swapped
- shadow pages
  - only changes pages are replicated
  - less redundancy than the twin-block approach

In the following we assume a system with the following configuration:

- steal
- ¬force
- update-in-place
- fine-grained locking

ARIES Protocol:

- The ARIES protocol is a very popular recovery protocol for DBMSs
- The log file contains:
  - Redo Information: contains necessary information to re-apply changes
  - Undo Information: contains necessary information to undo changes

- the log information stored is written two times
  - log file for fast access: R1, R2 and R3 recovery
  - log archive: R4 recovery

- organization of the log ring-buffer:
  - when buffer is full, then the other end gets written to disk
  - this is done with one inserter and one writer

• **W**rite **A**head **L**og Principle
  - before a transaction is **committed**, all corresponding log entries must be written to disk
  - before a modified page is written back to disk, all log entries involving this page must have been written to disk

• this is called *forcing* the log

Required for Durability.

Some care is needed when writing the log to disk

- disks are not byte addressable
- larger chunks, usually 512 bytes
- remember, the system may crash at any time
- partial writes to the block are dangerous
- might require additional padding then forcing the log
- related problem: partial page writes

Some of these issues can be solved by hardware.

Restart after Failure:

- *winner* transactions: must be replayed completely
- *loser* transactions: must be undone

Restart Phases

- Analysis:
  - determine the *winner* set of transactions
  - determine the *loser* set of transactions
- Repeating History:
  - all operations contained in the los are applied to the database instance in the original order
- Undo of Loser Transactions:
  - the operations of *loser* transactions are undone in the database instance in reverse order

Structure of Log Entries:

[LSN, TA, PageID, Redo, Undo, PrevLSN]

- Redo:
  - physical logging: after image
  - logical logging: code that constructs the after image from the before image
- Undo:
  - physical logging: before image
  - logical logging: code that constructs the before image from the after image

ARIES: Redo is physical, Undo is logical

- LSN (Log Sequence Number):
  - unique number identifying a log entry
  - LSNs must grow monotonically
  - allows for determining the chronological order of log entries
  - typical choice: offset within the log file (i.e., implicit)
- TA:
  - transaction ID of the transaction that performed the change
- PageID:
  - the ID of the page where the update was performed
  - if the change affects multiple pages, multiple log records must be generated
- PrevLSN:
  - pointer to the previous log entry of the corresponding transaction
  - needed for performance reasons

Note: often there is a certain asymmetry:
physical redo (one page), logical undo (multiple pages)

The Phases – Analysis

- the log contains BOT, commit and abort entries
- the log is scanned sequentially to identify all TAs
- when a commit is seen, the TA is a winner
- when abort is seen, TA is a loser
- TAs that neither commit nor abort are implicitly loser

Winners have to be preserved, losers have to be undone.

The Phases – Redo

Redo brings the DB into a consistent state

- some changes might still be in main memory at the crash (¬force)
- changes can be incomplete (e.g., B-tree split)
- but the log contains everything

Redo is often done by one forward pass

- all log entries contain the affected page
- the pages contain LSN entries
- of the LSN of the page is less than the LSN of the entry, the operation must be applied
- the LSN is updated afterwards!
- allows for identifying the current state

Afterwards, the DB has a known state.

The Phases – Undo

Eliminates all changes by *loser* transactions

- during analysis, DBMS remembers last LSN of each transaction
- transactions that aborted on their own can be ignored (no “last operation”, all undone)
- active TAs have to be rolled back

Log is read backwards

- last LSN pointers are used for skipping
- all encountered operations are undone
- produces new log entries (redo the undo)

Idempotent Restart

- CLRs (compensating log records) for undone changes
- log records, which will undo certain redos
- we will have to write less records even when there are multiple crashes
- a CLR is structured as follows
  - LSN
  - TA
  - PageID
  - Redo Information
  - PrevLSN
  - UndoNxtLSN (pointer to the next operation to undo)
- no undo info, only redo
- prevLSN/undoNxtLSN could be combined into one
  (prevLSN is not really needed)

Partial Rollback

- undo some actions and continue with another
- *this is supported by CLRs*
- necessary to implement save points within a TA

Checkpoints

Problem with restart: reading large parts of data
Solution: take a time slice and record who is running during this time

- transaction consistent
  only create checkpoint when no transaction is running
  → *analysis, redo and undo never have to go beyond this point*
- actions consistent:
  stop current transaction for a moment and create checkpoint
  → *analysis and redo until checkpoint, undo goes until MinLSN*
- fuzzy checkpoint:
  start creating checkpoint and record all active transactions. When you are done, check active transactions again.
  → *analysis until checkpoint, undo until MinLSN, redo until MinDirtyPageLSN*

Fuzzy Checkpoints:

- modified pages are not necessarily forced to disk
- only the page ids are recorded
- Dirty Pages = set of all modified pages
- MinDirtyPageLSN:
  the minimum LSN whose changes have not been written to disk yet

Set-Oriented Query Processing:

Motivation:

During query processing, the DBMS tries to process whole sets of data items at a time

- "manual" programming is usually record oriented
- e.g., compare two records
- easy to understand, but this does not scale

Consider: intersecting two lists

- breaking it into record-level operators is inefficient
- compares each record with each other record
- $\mathcal{O}(n^2)$
- considering the complete lists in one step is more efficient
- $\mathcal{O}(n \log n)$

Set-oriented processing has several advantages:

- data can be pre-processed before processing
- sorting/hashing/index structures etc.
- amortizes over the set
- leads to more efficient algorithms
- easier to cope with memory limitations etc.
- easier parallelism
- …

Algorithms tend to become more scalable, but also more involved.

The Algebraic Model:

Query processing is usually expressed by relational algebra

- operators consume zero or more relations and produce one output relation
- inherently set (or rather: bag) oriented

29.06.2021 (Lecture 10)

Timeline:

T1: Compile DB

T2: Compile Query

T3: Execute Query

It's important at T1 to implement operators, such that they can handle arbitrary queries.

Implementing the Algebraic Model:

Operators are specified in a query agnostic manner:

- intersect
  - left
  - right
  - compare

Operator does not understand the query semantic. It only knows:

- *left* will produce a result set
- *right* will produce a result set
- *compare* compares two elements

Note: a scalable implementation will need more (e.g., hashLeft, hashRight), we ignore this for now.

The algebraic operators define the **abstract logic** of query processing primitives. The query specific parts are hidden in **subscripts**.

In particular:

- operators do not "know" the data types or byte size of input tuples
- they do not "understand" the content of a tuple
- they only specify the data flow and the control flow
- all query dependent operations are delegated to helper subscripts
- keeps the operator itself very generic

Note: sometimes operators are hinted with query specific info (e.g., a fixed tuple size) for performance reasons, but this is only a minor variation.

Operator Composition:

- each operator produces a set (bag/stream) of result tuples
- operators consume zero or more input sets
- usually assume nothing about their input
- therefore can be combined in an arbitrary manner
- very flexible

Option 1: Full Materialization

Every operator materializes its output. The input is always read from a materialized source.

Advantages:

- easy to implement
- can handle surprises concerning intermediate result sizes
- advanced techniques like parallelization, result sharing, etc. are simple

Disadvantages:

- materialization is expensive
- in particular if data is larger than memory

Few systems use this approach, but some do (MonetDB).

Option 2: Iterator Model

Each operator produces a tuple stream on demand. The input is iterated over.

Advantages:

- data is pipelined between operators
- avoids unnecessary materialization
- flexible control flow
- easy to implement

Disadvantages:

- millions of virtual function calls
- poor locality

The standard model. Widely used.

The iterator model usually offers the following interface:

- open
- next
- close
- *first (often, a lot of computation is done before the first tuple is computed)*

Repeated calls to *next* produce the output stream.

Ideally, operators maintain a complex state to offer the iterator interface.

How to pass data from one operator to the other?

- the data itself is opaque
- as a consequence, it cannot be passed (easily) by value

Alternative 1: pass tuple pointers

- the real data resides on a page/in the buffer
- operators are only passed pointers to the data

Alternative 2: not at all

- there is a global data space ("registers")
- subscript functions operate on these registers
- the operators never touch the data directly

Alternative 2 is more generic and can cope better with computed columns.

Option 3: Blockwise Processing

Each operator produces a tuple stream, but not tuple-by-tuple but as a stream of larger chunks.

Advantages:

- far fewer function calls
- better code and data locality
- *vs. full materialization: use chunks which fit into cache*

Disadvantages:

- additional materialization overhead
- consumes memory bandwidth
- control flow not as flexible

Option 4: Pushing Tuples Up

Each operator pushes produced tuples towards the consuming operators.

Advantages:

- operator logic is concentrated in a few loops
- good code and data locality
- pipelining etc. still possible
  pipelining: make only few copies of the data
- support for DAG-structured plans
  DAG: directed acyclic graph

Disadvantages:

- some restrictions in control flow
- code generation more involved

Note: For CPU processing reasons, it's better to use predicate execution (without if-statements) like $w = w + p(R'[w])$ *for moving the write pointer when a match is found.*

Exercise:

Operators Project:

Use Alternative 2: not at all (register) of the Iterator model.
Only call `get_output_values` once in `open`, then save the register pointers and blindly assume they will be populated whenever child operator returns `true` on `next()`

```
open:
      rout = right.get_output()

bool next:
      if not hashtable_built:
            ht = std::unordered_multimap{}
            lout = left.get_output()
            while(left.next()):
                  key = *lout[keyID] //Register Pointer
                  ht.append(key,*lout)    // load every value of the
                                          // left output
            hashtable_built = true
      if not previous_match.empty():
            leftoutput = *previous_match
            previous_match++
            return true
      while (right.next()):
            key = rout[key]
            previous_match = ht.find(key) // range of keys
            if not match.empty():
                  leftoutput = *previous_match
                  match++
                  return true
      return false
```

`Intersect` -> Set

Eliminate Duplicates? (`unordered_map` instead of `multimap`)
necessary to remember output tuples -> bool flag

`Intersect_All` -> Bag

use counter where bool flag is used in intersect

LLVM Code Generation (Bonus Project):

not standard stack for DBMS
implementing subscripts
practical course next semester:
~200lines
Deadline 13.07.2021

06.07.2021 (Lecture 11)

Additional Functionality:

We ignored the *close* function so far
- releases allocated resources

Other functionality implemented or used by operators
- rewind/rebind
- memory management
- spooling intermediate results

Implementing Subscripts:

The operators are query independent, but the subscripts are not
- cover the query-specific parts of the query
- attribute access (e.g., x.a)
- predicates (e.g., a=b)
- computations (e.g., sum(amount*(1+tax)))
- …

Must be implemented, too:
- different for every query
- but usually relatively simple
- complexity much lower than for operators

Option 1: **interpreter objects**

Subscripts are assembled from interpreter objects.
- very flexible
- easy to implement
- widely used
- but: many virtual function calls

```
Val AccessInt::eval(char* ptr)
  return *((int*)(ptr+ofs));

Val CompareEqInt::eval(char* ptr)
  return left->eval(ptr).intValue==right->eval(ptr).intValue;
```

Option 2: **virtual machines**

Subscripts are compiled into instructions for a virtual machine

- more efficient than interpreter objects
- but also more complex
- requires a compiler to byte code

```
while (true) switch ((++op)->cmd){
  case Cmd::AccessInt:
    reg[op->out]=*((*int)(ptr+op->val);
    break;
  case Cmd::CompareEqInt:
    reg[op->out]=reg[op->in1].intValue==reg[op->in2].intValue;
    break;
  …
}
```

Option 3: **pre-compiled fragments**

Subscripts are expressed as combination of pre-compiled fragments.

- each fragment performs a number of operations
- quite efficient (vectorization)
- but usually only applicable for column stores

```
CompareEqInt(unsigned len,int* col1,int* col2,bool* result)
  for (unsigned index=0;index!=len;++index)
    result[index]=col1[index]==col2[index]
```

Option 4: **generated machine code**

Subscripts are at runtime compiled into native machine code.

- the most efficient alternative
- but also the most difficult
- portability is an issue
- we will look at this in the section Code Generation

```
…
movq  72[%rsp), %rax
molv  (%rax,%r12,4) %r13d
movq  120(%rsp), %rax
movl  (%rax,%r12,4), %edi
cmpl  %r13,4,%edi
…
```

Pipelining:

As mentioned, most approaches try to avoid copying data between operators:

- this is called *pipelining*
- operators that do materialize their input are called *pipeline breakers*
- operators that consume their input completely before processing are called *full pipeline breakers* (e.g., sort)
- some binary operators are pipeline breakers on only one side

This behaviour has implications regarding other operators.

Some effects of different pipeline behaviour:

- if a pipeline break is between source and sink, the original data is no longer accessible
  - relevant for lazy attribute access/TID join/string representations etc.
  - the system must plan defensively
- if a pipeline breaker is between two operators, both are decoupled
  - the full pipeline break breaks the plan into fragments
  - can be executed independent from each other
  - relevant for scheduling
- …

The code generation must know the pipeline behaviour of operators.

How can we exploit multiple cores during query processing?

- inter-query parallelism is simple
- intra-query parallelism is much harder
- independent parts of the query can be executed in parallel (see: full pipeline breaker)
- parallelizing individual operators is more difficult
- usual strategy: partition the input

We will discuss this later in more detail.

Algebraic Operators:

Queries are translated into relational algebra:

- therefore a DBMS must offer implementations for all algebraic operators
- often, more than one implementation
- different implementations are tuned for different usage scenarios

Complexity varies:

- a few operators are very simple
- but most are more complex
- pipelining operators tend to be simple
- pipeline breakers tend to be complex

Table Scan:

The most basic operator

- produces all tuples contained in a relation
- conceptually very simple
- implementation complexity varies from simple to complex
- has to navigate the physical representation of the relation
- additional complexity from deferred updates, snapshot isolation, etc.

Selection:

A selection σ_p

- filters out all tuples that do not satisfy p
- a very simple operator
- many systems do not even implement it as separate operator
- instead, piggybacked onto other operators

Map:

A map $\chi_{a:f}$

- computes a new column by evaluating f
- another very simple operator
- like selections, often piggybacked

Join:

A join $e_1 \bowtie_p e_2$

- a very complex operator
- one if the most important operators
- several different implementations exist

Candidate implementations depend on the join itself:

1. If e_2 depends upon e_1, nested loop join must be used (i.e. dependent join $e_1 \bowtie e_2$)
2. otherwise, if p has the form $e_1.a = e_2.b$, any join algorithm can be used (equi-join)
3. Otherwise, wither nest loop or blockwise nested loop can be used

<u>Nested-Loop Join:</u>

The nested loop join $\bowtie_p^{NL}$ is the most flexible, but also most simple and inefficient join

- evaluates the right hand side for every tuple on the left side
- pairwise comparison, suitable for any kind of predicate
- the right hand side is evaluated very frequently

Blockwise-Nested-Loop Join:

The blockwise nested loop join $\bowtie_p^{BNL}$

- loads as many tuples from the left side as possible
- evaluates the right side and joins
- and repeats this with additional chinks from the left side
- like a NL join, suitable for any predicate (but not for $\rtimes$)
- greatly reduces the number of passes over the right hand side
- potentially speeds up execution by orders of magnitude

Sort-Merge Join:

The sort-merge join $e_1 \bowtie_p^{SM} e_2$

- assumes that p has the form $e_1.a = e_2.b$, that e_1 is sorted on *a*, and that e_2 is sorted on b
- some implementations actually perform the sort, too, but we consider it as separate operator
- performs a linear pass over the input to find matching entries
- very fast and efficient (after sorting)
- **but:** only simple for the 1 : N case!

Node: a SM join is simple in the iterator mode, but not in the push model (control flow). We materialize the left hand side here to simplify the code, which is usually not needed.

- non-standard, as left hand side is already materialized
- one usually tries to avoid this
- better: parallel scans through both sides
- materialization only if an $n:m$ match is found

Hash-Join:

The hash join $e_1 \bowtie_p^{HJ} e_2$

- assumes that p has the form $e_1.a = e_2.b$
- builds a hash table from e_1, and probes the hash table with e_2
- in-memory case and external memory case
- real implementations offer both in one (1st in-memory, external when needed)
- we split both cases to simplify the code

- for the combined case, consumeFromLeft would check for overflows
- switch to external memory hash join when memory is full

Grace-Hash-Join:

- left side and right side are partitioned
- recursive re-partition might be needed
- once partitions fit, partitions can be joined in-memory

- left side is materialized in memory
- when memory is full, data is written to corresponding partitions on disk
- here: sort to produce sequential I/O

- breaks large partitions into finer partitions
- until the partition fits into main memory
- overflow partitions are a special case

- materializes right side, just like the left side
- but can use the proper partition boundaries now

- partitions are joined pair-wise
- due to the equal join, join partners can only be found in corr. partitions
- the overflow case needs some special care

- P is known to fit for the left side
- brings it to the in-memory case

- partition did not fit (identical values)
- similar to a blockwise-nested-loop join
- but uses a hash-table to speed up finding matches

Singleton Join:

A singleton-join is a join $e_1 \bowtie_p^{1J} e_2$

- where $|e_1|$ is guaranteed to be ≤ 1
- relatively common, e.g., after group-by, scalar subqueries, etc.
- nested loop join would work, of course
- but singleton is even simpler

Non-Inner Joins:

So far, we have only considered inner joins. But:

- $\bowtie$, $\bowtie$, $\bowtie$ have to produce non-matching tuples, too
- $\ltimes$, $\rtimes$ have to produce only matching tuples without multiplicity
- $\triangleright$, $\triangleleft$ have to produce only non-matching tuples without multiplicity

Similar mechanism but requires additional bookkeeping.

Idea: non-inner joins *mark* tuples with a join partner

- outer joins: additional pass, produce non-marker tuples with NULL values
- semi joins: produce only if not marked yet
- anti joins: additional pass, produce only non-marked tuples

Problem: where to store the marker bit?

Marking the left side is usually simple:

- left tuples in-memory for potential join partners
- reserve a bit in the memory block/hash table/…
- bit can be examined before flushing the memory

Marking the right side is more problematic:

- ok if a right hand tuple is only considered once
- then, store marker in a register
- but nested loop joins etc. are difficult
- maintain extra data structure (interval compression)

Sort:

Sorting is useful for a number of other operations (sm-join, aggregation, duplicate elimination, etc.)

Basic strategy:

1. load chinks of tuples into memory
2. sort in-memory, write out sorted runs
3. merge sorted runs
4. recurse if needed to handle merge fanout

Implementation varies a bit.

Exercise:

Exercise 1: Implement `Except All` in the iterator model in Pseudo Code:

```
open(){
  left.open()
  right.open()
  output = left.output.copy
  loaded_left = false
  loaded_right = false
  out_iter = {}
  ht = {}
}

next(){
  if(!loaded_left){
    while(left.next()){
      i = ht.get_or_default(left,0)
      i += 1
    }
    loaded_left = true
  }
  if(!loaded_right){
    while(right.next()){
      ht[right]--
    }
    loaded_right = true
    out_iter = ht.begin()
  }
```

```
  while(out_iter->second <= 0){
    out_iter++
    if(out_iter == ht.end()) return false
  }
  if(*out_iter > 0){
    out_iter->second -= 1          // this is the value
    out_reg = out_iter->first      // this is the register
    return true
  }
  return false
}

close(){
}
```

Exercise 2: Generate pseudo code for the following operator tree using the push model.

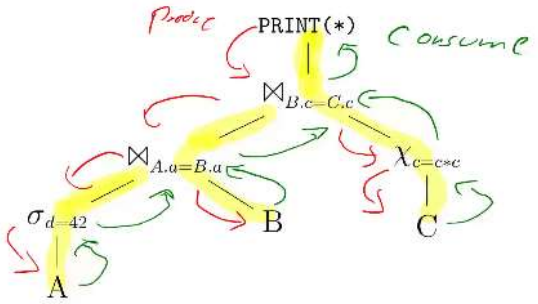
Push: produce (pre-order) and consume (in-order)

Exercise 2

Generate (pseudo-)code for the following operator tree using the push model.

Use the following schema.

- A(a, d)
- B(a, b, c)
- C(a, c)




```
ht1={} (Multiset),ht2={}
for(a in A):
  if(a.d=42):
    ht2.insert(a.d,a)
for(b in B):
  if(ht2.contains(b.a)):
    a = h2.get(b.a)
    ht1.insert(b.c,a,b)
for(c in C):
  d=c.c*c.c
  for(match in ht1.matches(c.c)):
    print(match.a,match.b,c,d)
```

13.07.2021 (Lecture 12)

General Sort Algorithm:

```
Sort::produce()
  input.produce()
  flush memory
  for each t in merged partitions
    consumer.comsume(t)

Sort::consume(t)
  if not enough memory to store t
    flush memory
  materialize t in memory
```

But how to implement flush and merge?

Easy solution for flushing:

- sort the in-memory tuples using quick sort
- write out all of them, release all memory
- fast sort algorithm, simple

More complex solution:

- use heap sort with replacement selection
- write out only when needed
- produces longer runs
- for sorted input: one run
- variable-sized records are difficult to handle

Merging is usually performed on the fly:

- read all runs in parallel
- retrieve always the smallest element
- tree of losers, or some other priority queue

Problem: what if the number of runs is too large?

- perform a partial merge
- reduces the number of runs
- repeat until merge is feasible

Group By:

Aggregation can be implemented in two ways:

Sort based:

- input is sorted by group-by attributes
- all tuples within one group are neighboring
- aggregation is largely trivial

Hash based:

- aggregate into hash table
- spill to disk if needed
- merge spilled partitions, similar to hash join partitioning

```
InMemoryGroupBy::produce()
  initialize an empty hash table
  input.produce()
  for each group t in hash table
    consumer.consume(t)

InMemoryGroupBy::consume(t)
  if hash table[t|A] exists
    update the group hash table[t|A]
  else
    create a new group in hash table[t|A]
```

Variable-length aggregates require some care.

Set Operations:

The DBMS also offers set operations:

- `union / union all`
- `intersect / intersect all`
- `except / except all`

Somewhat similar to joins but have a very specific behavior.

`union all:`

The only trivial set operation:

- concatenates both tuple streams
- attribute rename required
- otherwise nearly no code

`union:`

Like a union, but without duplicates:

- can be implemented as `union all` followed by duplicate elimination
- $e_1 \cup e_2 = \Gamma_A(e_1 \,\overline{\cup}\, e_2)$
- or: directly write into one hash table
  (slightly more efficient, but nearly identical)

`intersect:`

Similar to a semi-join:

- $e_1 \cap e_2 \approx e_1 \ltimes e_2$
- only true if e_1 is duplicate free
- can be checked during the build phase
- or: use a strategy like for `intersect all`

`Intersect all:`

Intersection with bag semantics:

- defined via characteristic functions
- group by sides into one hash table
- count occurrences on the left and right side
- intersect result is minimum of both
- standard group-by algorithm (including external memory etc.)
- for in-memory case can prune right side

`except:`

Similar to anti-join:

- $e_1 \setminus e_2 \approx e_1 \triangleright e_2$
- only true if e_1 is duplicate free
- can be checked during the build phase
- or: use a strategy like for `except all`

`except all:`

Set difference with bag semantics:

- defined via the characteristic functions
- group by sides into one hash table
- count occurrences on left and right side
- except result is $\max(0, l_c - r_c)$
- standard group-by algorithm (including external memory etc.)
- for in-memory case can prune right side

Code Generation:

Motivation:

For good performance, the operator subscripts have to be compiled

- either byte code
- or machine code
- generating machine code is more difficult but also more efficient

Machine code has portability problems:

- code generation frameworks hide these
- some well known kits: LLVM, libjit, GNU lightning, …
- greatly simplify code generation, often offer optimizations

LLVM is one of the more mature choices.

LLVM:

Distinct characteristics:

- unbounded number of registers
- SSA form
- strongly typed values

In LLVM we can just act as if we had unbound number of registers.

Compiling Scalar Expressions:

- all scalar values are kept in LLVM registers
- additional registers for NULL indicator if needed
- most scalar operations (=, +, -, etc.) compile to a few LLVM instructions
- C++ code can be called for complex operations (`like` etc.)
- goal: minimize branching, minimize function calls

The real challenge is integrating these into set-oriented processing.

Data-Centric Query Execution:

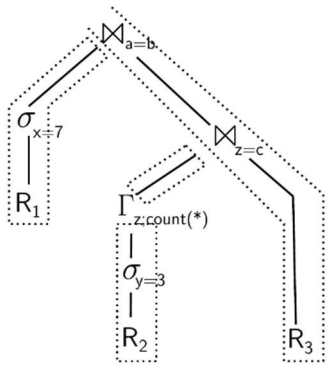
Why does the iterator model (and its variants) use the operator structure for execution?

- it is convenient, and feels natural
- the operator structure is there anyway
- but otherwise, the operators only describe the data flow
- in particular, operator boundaries are somewhat arbitrary

What we really want is **data centric** query execution:

- data should be read/written as rarely as possible
- data should be kept in CPU registers as much as possible
- the code should center around the data, no the data move according to the code
- increase locality, reduce branching

Example plan with visible boundaries:



- data is always taken out of a pipeline breaker and materialized into the next
- operators in between are passed through
- the relevant chunks are the pipeline fragments
- instead of iterating, we can push up the pipeline

Corresponding code fragments:

```
initialize memory of Join_(a=b), Join_(c=z), and Gamma_z
for each tuple t in R1: if t.x = 7: materialize t in hash table Join_(a=b)
for each tuple t in R2: if t.y = 3: aggregate t in hash table of Gamma_z
for each tuple t in Gamma_z: materialize t in hash table of Join_(z=c)
for each tuple t_3 in R3:
      for each match t_2 in Join_(z=c)[t_3.c]:
            for each match t_1 in Join_(a=b)[t_3.b]:
                  output t_1 o t_2 o t_3
```

Basic strategy:

1. the producing operator loops over all materialized tuples
2. the current tuple is loaded into CPU registers
3. all pipelining ancestor operators are applied
4. the tuple is materialized into the next pipeline breaker

- tries to maximize code and data locality
- a tight loops performs a number of operations
- memory access is minimized
- operator boundaries are blurred
- code centers on the data, not the operators

Producing the Code:

Code generator mimics the produce/consume interface

- these methods do not really exist, they are conceptual constructs
- the *produce* logic generates the code to produce output tuples
- the *consume* logic generates the code to accept incoming tuples
- not clearly visible within the generated code

Parallel Query Execution:

Why parallelism?

- multiple users at the same time
- modern server CPUs have dozens of CPU cores
- better utilize high-performance IO devices

Forms of parallelism:

- inter-query parallelism: execute multiple queries concurrently
  - map each query to one process/thread
  - concurrency control mechanism isolates the queries
  - except for synchronization that parallelism is "for free"
- intra-query parallelism: parallelize a single query
  - horizontal (bushy) parallelism: execute independent sub plans in parallel (not very useful)
  - vertical parallelism: parallelize operators themselves

Vertical Parallelism: Exchange Operator

- optimizer statically determines at query compile-time how many threads should run
- instantiates one query operator plan for each thread
- relational operator can remain (largely) unchanged
- often (also) used in a distributed setting

Exchange Operator Variants:

- Xchg(N:M) N input pipelines, M output pipelines

Many useful variants:

- XcghUnion(N:1) specialization of Xchg
- XchgDynamicSplit(1:M) specialization of Xchg
- XchgHashSplit(N:M) split by hash values
- XchgBroadcast(N:M) send full input to all consumers
- XchgRangeSplit(N:M) partition by data ranges

Aggregation with Exchange Operators (3-way parallelism):

Standard: $R \rightarrow \sigma \rightarrow \Gamma$
1. Option: $R_{1-3} \rightarrow \sigma \rightarrow \Gamma \rightarrow \text{Xchg(3:1)} \rightarrow \Gamma$
2. Option: $R_{1-3} \rightarrow \sigma \rightarrow \text{XchgHashSplit(3:3)} \rightarrow \Gamma \rightarrow \text{Xchg(3:1)}$
3. Option: $R_{1-3} \rightarrow \sigma \rightarrow \Gamma \rightarrow \text{XchgHashSplit(3:3)} \rightarrow \Gamma \rightarrow \text{Xchg(3:1)}$

Option 1, if there are few groups after the group-by.
Option 2, if there are many groups after the group-by.
Option 3 is a combination of Option 1 and Option 2.

Disadvantages of Exchange Operators:

- static work partitioning can cause load imbalances (probl. with many threads)
- degree of parallelism cannot easily be changed mid-query (workload changes)
- overhead:
  - usually implemented using more threads than CPU cores (context switching)
  - hash re-partitioning often does not pay off
  - exchange operators create additional copies of the tuple

Parallel Query Engine:

- alternative to Exchange Operators: parallelize operators themselves
- requires synchronization of shared data structures (e.g., hash tables)
- allows for more flexibility in designing parallel algorithms for relational operators

Morsel-Driven Query Execution:

- break input into constant sized work units (“morsels”)
- dispatcher assigns morsels to worker threads
- # worker threads = # hardware threads
- *result is written to private memory of thread*

Dynamic Scheduling:

- the total runtime of a query is the runtime of the slowest thread/core/machine
- when dozens of cores are used, often a single straggler is much slower than the others (e.g., die to other processes in the system or non-uniform data distributions)
- solution: don't partition input data at the beginning, but use dynamic work stealing:
  - synchronized queue of small jobs
  - threads grab work from queue
  - the `parallel_for` construct can provide a high-level interface

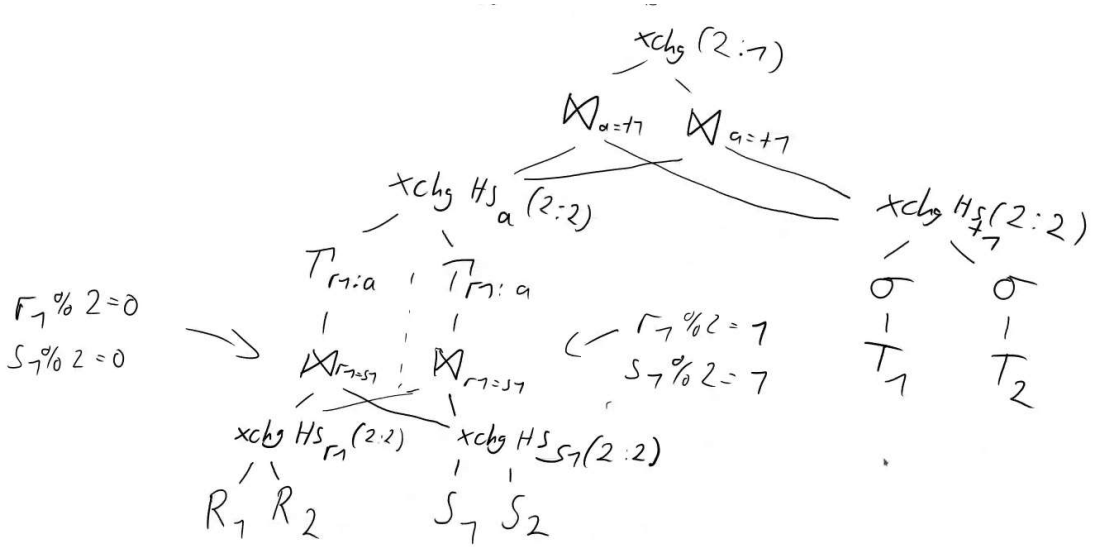
Parallel In-Memory Hash Join:

1. build phase:
   1.1 each thread scans part of the input and materializes the tuple
   1.2 create table of pointers of appropriate size (tuple count sum of all threads)
   1.3 scan materialized input and add pointers from array to materialized tuples using atomic instructions
2. probe phase: can probe the hash table in parallel without any synchronization (as long as no marker is needed)

END

Exercise:

Parallelization with Exchange Operators:



Recap of Lecture Contents:

Layered Architecture: (take a look at hierarchy slides)

Storage → Access → Query

Storage:
- disk read/write
- Memory hierarchy (Tape → … → Registers (know magnitudes))
- Buffer Management (Interface, Concurrency, Replacement Strat.)
  a lot of possible questions about understanding
- Segments (storage allocation) *extent mappings*
- Dirty Page Handling (shadow paging, delta)

Access:
- Data Structures
- Free Space Inventory
- Slotted Pages (allocation, reallocation ~ redirects)
  many questions about understanding
- Record Layout (alignment, padding, variable length data)
- Long Records (BLOBs → BLOB Trees →extent lists)
  (Text ~ short string optimization (prefix))
- B-Trees (concurrent access, locking, deadlock avoidance)
  (non-unique keys (record TID)=
  (Prefix queries)
  (Bulk-Loading)
  (Variable Length Records ~ prefix compression)

  possible questions about implementation and understanding
- Hash Tables (Extendable, linear, multi-level)
- Bitmap Indexes, Small materialized aggregates

Transactions and Recovery:
- *skipped a lot of the advanced stuff*
- Update Strategies (in-place, shadow pages (copy on write))
- ARIES Recovery
  - write ahead logging (WAL)
  1. Analysis (winners and losers)
  2. Redo everything
  3. Undo losers
  - checkpoints

  no implementation given, basically only ARIES

Skipped many of the slides, here.

Queries:
- Operators
- Iterator Model (Operations: Join, Select, Group-By, Set-Ops, Sort)
  very implementation-focused
- Push Model (- " -)
- Implementations (NestedLoop, BNL, SM, HJ)
- Subscripts (conditions on join, etc.) ~ Expressions
  get a bit comfortable with LLVM code

- Pipelines (identify distinct pipelines)
- Parallel Execution

Basically, if there is implementation, there can be many questions about understanding.
Closed book, on site exam!
Time Pressure is a tool against cheating (that's why live exam should be better)