# Data Management Systems

## 1 Intro

### 1.1 Physical and logical independence

User does not care about how the data is stored, and what format it's stored in memory - is it on RAM, is it NTFS, whatever.

Logical independence means the data is the data, and You can have different views of it.

I guess physical independence is nuts and bolts of actual storage, logical is how the data is presented to the user.

### 1.2 Query Optimization

Equivalent results may be achieved by different operators, and some are more efficient than others e.g. join and select v.s. select and join.

### 1.3 Data Integrity

Enforcement of legal values, so the database is intact/coherent.

### 1.4 Access Control

### 1.5 Concurrency Control

### 1.6 Recovery

### 1.7 Data vs Query shipping

### 1.8 Cons/Pros of Shared-Nothing

Pros: easy to maintain and to scale, sure.

Ideal when data can be sharded - a shard is a horizontal partition. Yeah I suppose in a horizontal case there won't be any need to do data shipping.

Basically works well as long as each node can function as it's own little island - this is broken if the data cannot be horizontally partitioned for some reason or if there are a lot of updates to the entire table, because operations that involve the entire table will probably do data shipping and that's no bueno.

## 1.9 Shared Memory

Not actually shared memory, but this is an abstraction that allows the node to pretend as if though all the nodes are sharing memory - so I guess the advantage here is that if a table is in some node's memory, it might as well be in Your memory. Simplifies design, and I guess if hardware allows for this it ought to be fast.

## 1.10 Shared Disk

Is more cloud oriented - the idea is that each node has access to some sort of giga-sized storage that contains everything, and local storage becomes more of a cache.

## 1.11 Exericses - Process Models

### 1.11.1 Thread v.s. Process

Process is a task in an OS - can basically think of it as an application. It's got it's own virtual memory, privileges, access to stuff, all that.

A process can have threads, these are I guess subtasks for the process. The main point here is that the threads share resources of the process - files, memory, whatever. So communicating between threads is fast, communicating between processes is slow and hard lol.

A DBMS client is, well, a client - it's a piece of software belonging conceptually to the user of the DBMS, and the client interacts with the DBMS through some established API.

The thing that the client interacts with is a worker - each client gets a worker. Picture a loop listening for connections on a socket. You can think of the DBMS server as the whole DBMS from the client's perspective - a client sends a request to a server, and within the server a worker is created to handle the client's requests.

## 1.12 Process per Worker

So the OS allocates a process for each conceptual worker.

Advantages are that it's easy to implement - every time You need something new, throw a process at it. OS handles parallelism, for better or for worse. Debugging processes is easier, since there's lots of support for that sort of thing from OS perspective.

Downsides are that DMBS workers need to communicate - locks for example. Not ideal for processes. Plus a process has it's own virtual memory, security stuff, basically overhead, so if You have a lot of workers it won't be super efficient.

## 1.13 Thread per worker

So You get a process for the DBMS I suppose, and within that process each worker gets a thread.

Advantage is that this is more lightweight, sharing is easier.

Disadvantage is that it can become a real mess, and You get to maybe do Your own management of resources instead of it being handled by the OS.

# 2 Storage Management

## 2.1 Memory Hierarchy

Registers, caches, memory, disk, external storage, archive.

Instant, 1ns, 100ns,

## 2.2 Storage Goals

Enhance temporal and spatial locality in layout.

Storage - contiguous storage, prefetching, caching.

Write back changes from fast layers to slower layers.

## 2.3 Logical and Physical Storage

### 2.3.1 Tablespace

Kay so it's a "logical data unit" so, it's *a thing* within the database. Easiest example is a table or a cluster of tables - they are logical objects in the database. I suppose tablespace is supposed to be a really broad grouping.

If something consists of a bunch of stuff and that something should be kept together, it's a tablespace. A space in which some related tables live?

Table maps to table space which maps to further constructs which finally yield the physical pages.

Logical locality.

## 2.4 Segments

A tablespace can have more than one segment, but usually has just one.

Segment is virtual memory, so I guess the segment is providing a unified view of memory to the tablespace.

Usually one segment per table, but You can partition these up.

But generally an object in the schema corresponds to a segment.

## 2.5 Extents

Contiguous memory blocks, divided into data blocks.

Not contiguous to each other.

Contiguous for spatial locality obviously/sequential access.

Extents are just straight up allocated - so say having extent size of 10GB might be wasteful.

Has a header, which contains information about which extent this is, who it belongs to, where the next extent is, what is stored in this extent.

We need an extent to contain blocks to manage space dynamically.

Alternatives: static file mapping. Sucks because a small table will not use the full file. Efficient in performance since You call malloc once and everything is massively contiguous.

Dynamic block mapping - allocate on a block level - shit for performance since everything is non-contiguous and You're calling malloc all the time. Great efficiency though.

Extents are in between.

## 2.6 Data Blocks

Could consists of several OS pages, but from the perspective of the database, this is as small as it gets.

Larger than OS page lol.

Block starts with a header of important metadata - who this belongs to, time of last update, type of content or whatever.

Table directory - structure of what is in the block. Homogenous items are stored in the block, for example rows of a particular table. The layout of that will be here.

Row directory - the block itself has like empty spaces in it, slots, and the slow directory tells You which tuple is in which slot.

Then some empty space,

Then the tuples in slots.

Percentage Free - how much of the data block is reserved to be free for updates.

Percentage Used - no new updates until the percentage of the block that is used is below this value. Prevents thrashing.

Info on top, data down below.

When to defragment? When there is sufficient free space in the block for insertion (so below percentage used), but everything is so fragmented that no insertion can be made.

## 2.7 How to allocate extents?

Simplest case: just manually set the size as a parameter.

Can allocate the same size every time.

Or maybe automate some changes, like repeated requests get more memory (*1.25).

## 2.8 Where to find space

The *segment* keeps a list of free blocks? Wild. Could be a bunch of extents in a segment so they keep free space at the segment level.

Ah okay segment is not tablespace, so segment contains extents contains blocks but segment keeps track of available blocks.

## 2.9 Tables ain't got order

Because maintaining it is expensive.

But can be done lol.

## 2.10 The art of writing to disk

Keep several copies of free lists for parallelization.

## 2.11 Shadow paging

Instead of writing updates on data, create copy of data and update that.

Once committed, mark the shadow block as the proper block and add the old unupdated block to the free list.

Expensive for large updates.

Complicates access.

Destroys locality.

Use when shadow paging is cheap i.e. on RAM.

Delta file - make a copy of the original rather than the changed version. Then perform the update on the original page, so it's in place.

Delta files favour commits.

## 2.12 Buffer Cache

We have stuff in memory, work with it there, when to move back to disk?

Disaggregated resources - usually CPU, memory, disk are couples, disaggregation splits them apart, like in the cloud.

## 2.13 Simple Structure

So You have a block number.

You has the number to get a hash bucket.

Then You check the latch for that hash bucket.

The has bucket will contain a link to a buffer header and then You can traverse that to try to find Your block.

If it ain't there You raise an error and fetch the block from disk.

## 2.14 Problems

Hot blocks - if an item is hot it'll get hit all the time and latches will block.

Scans block all the damn latches.

Similar operations happening hitting the same blocks.

## 2.15 Solutions

Create more latches.

Use multiple buffer pools.

Reduce amount of data in a block to distribute the load.

Use scans to avoid scans etc.

## 2.16 Buffer manager: Hash Buckets

So hashing the id yields a linked list of blocks.

To make lists smaller, increase number of buckets.

## 2.17 Buffer manager: Headers

A block in a linked list in a hash bucket will have tons of information - what's in the block, what segment the block belongs to, is the block dirty, last time of access etc.

## 2.18 Status of a block

Blocks can be pinned to memory if that's a good idea.

In some systems blocks will keep track of how often they've been accessed.

And of course whether the block is clean or dirty.

Gotta know all that stuff for replacement purposes.

## 2.19 What is in the linked list

Obviously You've got normal blocks.

But You may also have dirty blocks as mentioned before, or maybe delta blocks or shadow blocks, pinned blocks as well.

## 2.20 Replacement Policies

Least Recently Used - LRU - stuff that has not been accessed in a while gets dropped.

Every time an item is accessed it is put on top of the LRU list.

If dirty when dropped off list and out of memory, write it back.

Super shit for scans - Table Scan Flooding.

Index range scan - index organizes data (e.g. by age), and You're looking over a range of values, which produces a random access pattern since the tree structure will be all over the map.

## 2.21 Optimizations

Keep different pools - LRU might be ok for some access patterns, but You may also want a

Keep Buffer Pool - stuff pinned to memory.

Recycle buffer pool - stuff that don't matter.

etc.

Read ahead in blocks.

If You're going to need A, B and C tables, order Your fetches such that those are sequential.

You can do the same thing with a table that is out of order but contiguous.

Evict clean pages over dirty pages.

Ring buffers for scans.

Have different buffers for different size blocks.

Touch count. Weakness for touch count is that burst activity may make something look hot. Solution is to just update hotness periodically. Also periodically decrease counts with decay.

Clock sweep/second chance: run a garbage collector over the stuff in memory - if something is currently at 1 set it to 0, if it's at 0 toss it.

2Q: two queues. One is a first in first out queue, the other is least recently used. If something doesn't seem to be hot, it's put on the FIFO queue, and when memory is needed stuff from FIFO is dropped.

If a block is in FIFO and is accessed it's moved to LRU.

When that block is added to LRU, the LRU block gets bounced to FIFO (or evicted).

Can use automatic tuning to pin stuff that is often used.

## 2.22 Snowflake

Old systems designed to work with fixed hardware. Snowflake designed to work w/ cloud.

Everything is on S3.

Extents are replaced with micropartitions. Micropartitions contain tuples, but they are stored column-wise.

Metadata enables checking what is in micropartition without downloading it.

EC2 instances have local memory which can be used as a cache, and is used as an LRU cache.

## 2.23 Access Methods

### 2.23.1 Trade-offs

Everything is depending on what You need.

Computation v.s. space.

Analytic v.s. transaction workloads etc.

### 2.23.2 Finding pages

Tuple ID - block ID + offset.

Block ID - segment ID + offset.

### 2.23.3 Bottlenecks and solutions

So if You look for free space, first thing You'll need is a latch on the free list in the extent.

Which is bad, if there is only one list.

So then You can have a few and manage those etc.

You can also sort the available blocks by size, fill in smallest holes first etc.

### 2.23.4 Finding tuples within a page

You have slotted pages, slots of different sizes generated when the table was created.

Tuples will usually have a system-generated id which corresponds to the offset within the page.

### 2.23.5 Free List v.s. Used List

Free List contains addresses of blocks with less than % used parameter, so You can write in that block.

Used List contains blocks which are in memory and have stuff in them. So if You're looking for a tuple, You go to the used list and see if the block that has Your tuple is in memory.

### 2.23.6 Optimizing the record layout

Instead of storing tuples in length, item, length, item order, store offset, offset, offset, data order so that just the start of a tuple has to be accessed to get an attribute.

Also put variable length things at the end to make prefetching easier on constant size stuff at the start.

### 2.23.7 Corner cases

So what if You have a big 'ole attribute:

The idea is that they'll just sit there and not move, so just go ahead and store them, since to store them externally is to push away the complexity which can be annoying.

### 2.23.8 Data layout

If You are interested in reading a particular attribute that's like 4 bytes, You won't want to store row-wise since reading those 4 bytes will read 512 bytes at once, and all of that will be useless junk for You.

Also good for SIMD/AVX instructions.

### 2.23.9 Partition Attributes Across

So like Snowflake, I think.

Blocks contain the same tuples, but now it's just a column store within the block.

I suppose a column store goes the full-way for columns - pages contain one type of attribute.

#### 2.23.10 Compression

Primarily by dictionary:

Just map items to integers and store the integers.

## 2.24 Frame of reference

Make everything relative to some base - if all numbers hover around a billion, just store billion as a base and store the numbers as differences from that base.

You can go even farther and take every value as a delta.

### 2.24.1 Run length encoding

Instead of storing "2, 2, 2, 2", store that 2 happens 4 times.

### 2.24.2 Bitmaps

For every possible value an attribute can take, create a list as long as the table and store 1 if attribute is equal to one value and 0 otherwise. You'll get as many bitmaps as there are values.

### 2.24.3 Indexes

Our own data structures that We use to access data.

Indexes produce random-access patterns.

### 2.24.4 Hashing

Produces random access

Which can be good! If a "cluster" of blocks is hot, then breaking up that cluster will reduce congestion.

Good for transactions, bad for scans - only supports point queries, as in "get me tuple X".

So overall, You reserve some space with blocks, You hash the tuple id and that hash will tell You where in the block the tuple will go.

### 2.24.5 Extensible Hashing

You got two concepts, global depth and local depth.

Global depth is what dictates the number of possible values for the hash function. Global depth if two results in 4 possible values, depth of three in 8 and so forth.

Local depth is associated with individual configurations of bits

### 2.24.6 B-trees

If You have a bunch of duplicate values, do not hash on them lol.

Anyway, order $k$, any non-leaf contains at most $k - 1$ keys (I suppose for range indexing?)

B trees are not used, B+ trees are used. The difference is that in B+ the data is at leaves only. Or rather, pointers to data.

The leaves are blocks and are linked, so leaves can be traversed.

The keys in leaves may not be actual data - e.g. if You are looking for strings, keys might correspond to the first letter.

### 2.24.7 Clustered Index

But the linked list of leaves doesn't have to be contiguous.

But it will be if You use clustered indexing - basically forces leaves to be one after another in memory.

Best for tables that aren't modified all the time.

Maintaining indexes is expensive.

### 2.24.8 What to index

Stuff that has an ordering.

### 2.24.9 Index advantages

Yields data already sorted by index, and hopefully You don't have to go across the entire table.

### 2.24.10 Inserting when full

Add block, then split parent index.

### 2.24.11 Deleting

Delete tuple, balance with neighbour, and if neighbour half full and the current block is half full then merge.

### 2.24.12 Concurrent access

Lock coupling - lock parent.

If that doesn't work then abandon and lock everything.

### 2.24.13 Bulk Inserts

Create tree bottom up - sort everything, then create inner nodes.

#### 2.24.14 Optimizations

Don't use entire key.

Reverse key to break the sequential relationship if applicable.

Don't merge nodes when they are half empty - instead periodically rebuild entire index.

#### 2.24.15 How indexes are stored

Same as tables really - segments, extents, blocks.

Indexes can answer queries that are not dependent on the tuples themselves - only on the stuff that is in the index.

#### 2.24.16 Query Selectivity

Is kinda self explanatory.

#### 2.24.17 Clustered tables

Putting multiple tables on same segment - precomputed join really.

#### 2.24.18 Log structured file

Just store changes as opposed to values, ish.

Periodically compact.

Or rather keep all versions of stuff, and point to the newest version?

#### 2.24.19 Why no indexes for Snowflake

'cause Snowflake is for analytics, and indexes don't help there. They're massive.

You'd fetch the index, then according to index fetch data from cloud, and it's bandwith that costs (remember everything is compressed because of just this), so it don't make sense.

#### 2.24.20 Database Cracking

Don't build and maintain an index, instead incrementally build an index as You run queries.

E.g. queries iteratively sort the table? e.g. get entries where some value is between 10 and 14, then sort those entries and so forth.

# 3 Query Processing

## 3.1 Execution Models

### 3.1.1 Caching

Almost everything lol. Every intermediate part of a query that can be reused should be reused.

Client caching - cache results at the client.

Intermediate caching - there may be intermediate stages in query execution, basically.

### 3.1.2 Threads/Processes/Pools

OS process - basically an application, so it gets its own address space, state etc. Scheduled and managed by the OS.

Thread - belongs to a process, managed by kernel whatever that means.

Process per worker - easiest thing to do, but expensive, more difficult to do parallel processes.

Threads - gotta manage them, so more difficult, but lighter-weight and they can interact.

Postgres uses processes!

Can't have as many threads/processes as You want, instead create a pool to draw from. Makes it easier to parallelize since You just claim stuff from the pool.

### 3.1.3 Iterator Models

Iterator: scans one tuple at a time with next() calls originating one at a time from the root of the operator tree. Data flows up, control flows down. Generic interface, predictable access patterns. Also called "Volcano"

Materialization: compute entire result of operator and pass that. Downside is huge buffers. MonetDB, I think.

Vectorization Model: iterator model but with vectors, as in more than 1 tuple (vectors size of cache)

Push mode: instead of calling next, operators call push to produce their results. Event based programming, potentially more efficient due to improved function call locality. Hard to implement though.

For parallel execution introduce an EXCHANGE operator that pushes results to another process/machine.

### 3.1.4 Access to base tables

Row id access - when You know You want a particular tuple and You go get it.

Use index to get row id when the selectivity is very high.

In composite indexing, things depend on predicate selectivity.

In theory, slowest access pattern is a full table scan - one can cook up worse patterns, but none of them yield results additional to full table scan.

When to use full scan - low selectivity of predicate, no relevant index available ofc, self joins.

Full table scans can be improved by sharing scans across queries, compressing the data, and doing sample scans if approximate answers are good enough.

Other considerations: scanning index may be esspessive, but it returns data in order, so that's nice.

### 3.1.5 Sorting and Aggregation

External sort - since tuples don't fit into memory.

Also allows a larger number of queries to run in parallel, neat.

Two phase external sort - phase 1 loads tuples into available buffers and sorts them, writing out the sorted version to disk. A sorted chunk is called a "run". Phase 2 then merge-sorts by reading from the runs.

All fine and good unless the number of pages available as buffer is less than the square root of the number of pages of the input table, then multiple phases of merging will be necessary.

Must reserve at least one block for writing stuff out, though in practice 2 is better.

Sorting v.s. hashing - sorting is pretty expensive, hashing is cheaper, so unless You really need the sorted data it might be better to just hash it and aggregate it that way.

### 3.1.6 Optimization 1: Heuristics and Rewriting

Views: materialized view meaning there's an actual table corresponding to the view. Virtual meaning the DB just replaces the view with relevant base table columns.

Materialized views can be used as materialized results.

Rewriting: depending on how You write the query, the optimizer may catch on to simple optimizations.

Predicate transformation: figure out what the predicates are actually looking for and rewrite it with that, for example analyze the actual ranges dictated by predicates.

Predicate augmentation: add additional predicates that are logical consequences of the existing predicates to give the optimizer more options to work with.

Predicate pushdown: rewrite queries with views to use base tables, yep.

Heuristics: predicates are commutative and associative, selections are commutative and associative, filter then join is the same as join then filter

### 3.1.7 Optimization 2: Cost estimates

Got to choose between a bunch of possible query execution plans. Goal is to choose fastest one (or least resource intensive?)

So that means figuring out what implementation to use for each operator in the computation tree.

Histograms: equi-width is Your standard histogram with equal sized buckets. Equi-height or equi-depth sizes the buckets such that each bucket gets the same number of entries. Equi-width is good for hotspots.

Frequency histogram is basically a scatterplot - each item in database gets a frequency.

Zone maps are an interesting combo of the two centered around blocks lol. Sort of like equi-height histograms I suppose.

Attribute cardinality - how many different unique values does the attribute have.

Table cardinality - number of attributes in table.

Operator cardinality - the number of tuples that have to be processed for the operator to yield a result. So the cardinality of a full scan is just the number of tuples?

Predicate cardinality - how many tuples satisfy the predicate. So not unique values, just how many.

Selectivity - fraction of total data that the operator will yield.

Selectivity of disjunctions - sum indvidual selectivities and subtract conjunction.

Combine table cardinality with access pattern for cost estimate.

Rule-based optimizer: no stats, just use transformation methods, schema.

Heuristics are good, honed by time, but data is ignored so.

System-R: consider only left deep join trees. Left deep keeps appending tables to the join result

# 4 Transaction Processing

## 4.1 Concurrency and Control

ACID:

Atomicity, Consistency, Isolation, Durability.

Consistent state: state of database is a result of correctly applying transactions. Up to use to issue correct commands.

Correctly: satisfying constraints and as if though they executed independently.

Operations: inserts, updates, deletes.

Atomicity - transactions either occur or do not.

Isolation - transactions execute as if they are the only ones taking place.

Durability - persistentcy - power outage won't wreck things.

Transaction: system commit - transaction has been comitted lol.

Read: $r_1[x]$ writing item x by transaction 1. Writing is $w_i[x]$.

There's a whole history graph thing - higher items happen later.

## 4.2 History

Serial history - all operations from $T_i$ happen before any other $T_j$.

Equivalent histories - same transactions and operations and if there are conflicts, then the ordering among conflicts is identical between histories. Aborted stuff can be ordered any way, since, yano, it's a-borted lol.

Serializably history - equivalent to a serial history.

A history is serializable if it's serializability graph is acyclic.

## 4.3 Recovery

Recovery procedures: just abort/undo transaction.

Redo changes from committed transaction e.g. dirty page in memory was lost, use log to redo operation.

Similarly maybe You have something in disk that needs to be undone.

If lost disk - read snapshot, play out log on snapshot.

SQL phenomena - dirty read - reading uncommitted stuff. Obviously bad.

Non-repeatable read - within a transaction two read operations give different results.

Phantom reads - affects aggregates.

## 4.4 Transactions

2PL - 2 Phase Locking.

Shared locks and exclusive locks.

Growth and shrink of transactions. Those are the two phases. Ensures serializability basically.

Transaction manager:

Transaction table has a table containing all current transactions. Queries are elsewhere.

Transaction table contains a pointer to a handler. The handler has a list of stuff to be locked, as well as a pointer to a log of the stuff that the transaction will execute.

Lock table: hash table id, go to some bucket, You'll find a header and create a lock request. Your request will be added to a linked list.

Locking done at tuple level, ish.

## 4.5 Recovery

Physical v.s. logical logging.

Steal policy - dirty pages are allowed on persistent storage. Requires undo

Force policy - transaction not committed before changes in persistent storage

Steal requires redo, no force requires redo.

So, write - create log entry with before and after images, then write.

Commit - add commit to log.

Abort - restore log images.

Recovery - start at end of the log and sort all entries to undone and redone items. You're done when all items are in either category or You reached the beginning of the log.

Kay so then because You keep before and after images, You go through the list backwards and if redone, put after image on the after image list, same for undo. You only need to do one restore per item.

Advantages: easy buffer management, only forced IO in logs.

More complicated recovery relative to other approaches.

Undo no redo - as part of commit flush to disk.

Transactions now take longer, smaller log, faster recovery.

No undo, redo: read dirty pages from memory, otherwise read the entry from disk.

On on write, do the write to some temporary buffer, flush only when committed.

No undo no redo is a pipe dream since You can't atomically update 10 blocks.

Log record: stored on small blocks, 512 bytes ish.

SCN - timestamp.

Log Sequence Number keeps track of all logged bits for a particular transaction.

LSN orders transactions.

## 4.6 Types of locks

Shared lock - anyone who wants to read an item gets one.

Update lock - only one person at a time gets to hold this one, and it says 'ey, I'm gonna update this value. Blocks all other locks I think.

The scenario it prevents is that two transactions read the same item, and then both try to write and therefore secure a write lock, but ya can't get a write lock when there's a shared lock, so one gets the update lock and the other gets to wait, yeet.

Exclusive lock - good 'ol write lock.

Intent lock - transaction intends to get a shared/exclusive lock on table. Intent locks exist on the level of tables, so that if You want to know if something's going on in the table, You can just check this.

Schema locks - don't change schema, obviously.

Key-range locks - prevents anyone messing with a range of key values, e.g. You can't insert into locked range, so it prevents phantom reads. Imagine a transaction that reads the same range twice but someone has inserted into it.

## 4.7 Transaction Layer

Transaction table - keeps track of transactions, sure.

Transaction handler - honestly kind of ambiguous - information needed for transaction execution?

Lock table with buckets, and log.

# 5 An Evaluation of Buffer Management Strategies for Relational Database Systems

QLSM - query set locality model.

Domain separation - give different stuff different buffers. This is inflexible and dumb. It doesn't take into account relative priorities that may change dynamically.

GLRU - grouped LRU - partitions buffers so that different partitions have different priority. If You're looking to evict someone, start w/ lowest priority and take it from there..

Hot set - stuff being looped over is hot, so keep it in memory.

Hot point - if You can't keep the hot set in memory, suddenly page faults come in and ruin the performance. These discontinuities are called hot points.

Hot stuff only really applies to LRU though - MRU will just MRU.

MRU - most recently used.

Quary Locality Set Model - QLSM: You've got different query patterns. You've got Straight Sequential - ze SS, which is just a scan. Then You have a Clustered Sequential, where You might scan over some parts of the range multiple times, and then You have Looping Sequential which, well, are looped scans, and You want to keep everything that touches in memory, and not use LRU but instead opt for MRU. Looping Hierarchical is for looping over hierarchical structures, and You may access roots more often etc.

Or once more:

Quary Locality Set Model: basically analyzes each operation in a database and observed that they each have their localities. They just go through operations one by one to find access patterns lol. They've done this for a bunch of queries I suppose, and they've found common trends:

Straight Sequential: reference once and move on to next page. Just a scan. Ideal strategy is MRU, or LRU or whatever - You only need one page at a time.

Clustered Sequential: same as straight sequential but You jump back in the loop now and then. This means that some pages are re-used, and they form a cluster, so try to keep them in memory. LRU makes sense.

Looped sequential: same as clustered sequential but the cluster is the entire object, whatever it may be. Either fit the file in memory, or use MRU and keep as much of it in memory as possible.

Independent Random: what it says on the tin lol.

Clustered random: random access but You bump into clusters occasionally.

Straight Hierarchical - just get one tuple via traversing an index from root to leaf.

Hierarchical with Straight Sequential and Hierarchical with Clustered Sequential are the same, but the starting point is found via index, so nothing much changes.

Looping Hierarchical - You are finally looping over the index to find different leaves. Gotta keep the index nodes in memory of possible. Prioritize nodes closer to root.

DBMIN - gives buffers per file instance, as in per open file, even if the file is a duplicate. Locality set is the set of pages is buffer for a file.

So basically each file gets a buffer and that buffer is managed according to QLSM.

And I guess broadly speaking that's the main bit - it's individualizing buffers based on access pattern.

Locality set - the set of pages allocated for a file.

And now what does the locality set look like for each type of query:

SS - size 1, since nothing will be reread.

CS - Size of the maximum cluster, which makes sense, since it'll be clusters You'll be going back over, and replacement strategy is what - FIFO deletes old stuff and keeps new stuff in buffer, so when looping over a cluster it'll get rid of stuff before the cluster and LRU will preserve this, as it so happens.

LS - as much as You can and MRU, since really You just want to keep max amount of stuff in memory and do as few shuffles as possible.

IR - forget about it.

Clustered Random - same as clustered sequential.

Straight Hierarchical - don't matter.

Looping Hierarchical - as much of the tree as possible.

# 6 Snowflake Elastic Data Warehouse

## 6.1 Motivation

Trad systems: fixed systems so tough time balancing all online power, just designed for a smaller scale, highly/complexly tuned. Basically the servers used to be smaller and fixed where as now they are massive and ever growing.

Change in data: data no longer just internal, no longer a semi-constant stream.

## 6.2 Key qualities

Snowflake not based on Hadoop, PostgreSQL etc.

Snowflake: 100% software as a service. No management required by the user.

Relational support.

Support for semi-structured data, so stuff that breaks relational rules.

Elastic obviously since it's on the cloud.

Highly available, again the cloud.

Durable.

Cost-efficient - all table data is compressed! You pay for the storage and compute You use.

Secure - everything is end to end encrypted. Also has SQL-like users.

## 6.3 Storage v.s. Compute

Shared nothing - elegant, easy to scale, tightly couples storage with compute - You wanna compute something, it has to be on some node.

Things shared-nothing fucks up:

Heterogenous workload - the hardware is homogenous and partitioned, so.

Changing stuff costs - if You take nodes offline to replace them or upgrade or whatever, You need to move the data that is on the node. This is again just another coupling consequence.

On the cloud none of this holds - You have different nodes, nodes die all the damn time.

So then separate storage into S3 and shared-nothing compute nodes that cache some stuff.

So to avoid these drawbacks, Snowflake decouples storage from compute. Both are scalable independently.

Storage - S3, compute - Snowflake's own code.

If data is on a compute node, it's hot, so in that sense it's efficient.

## 6.4 Architecture

Storage - S3.

Virtual Warehouse of I guess compute nodes.

Cloud services - user facing API, basically all the other database stuff.

### 6.4.1 Storage

Table partitioned horizontally into blocks.

Uses PAX to store stuff in chunks.

Since S3 supports getting parts of files, stuff in blocks is grouped with the head the of the block containing metadata for offsets.

S3 is also used for storage required by operators e.g. huge joins, so it's got an infinite amount of storage lol.

## 6.5 Virtual Warehouse

Clusters of EC2 clusters. Individuals in the EC2 cluster are called Worker Nodes.

Kay so You have VWs, which has a set of EC2 instances. New query comes in, and workers spawn a new process for that query. Process dies at end of query.

## 6.6 Elasticity and Isolation

WVs are expendable - should only exist when queries are running.

One node = one WV.

Query comes in, gets a WV, then each worker in the EC2 cluster spawns a process for the query, which dies the moment the query is done.

WV's give isolation, wee. So long as data not changed?

Cloud elasticity is used - the price for 1 WV running 16 hours is the same as 16WV's running for 1 hour - if You parallelize, then You can improve performance without cost due to cloud resoruces being there anyway.

## 6.7 Local Caching and File Stealing

Worker nodes cache file headers and individual columns, so basically everything they can lol.

Caches live for the duration of worker node (so the query?). No, a WV can live longer than the duration of a query.

File stealing - overachiever worker nodes can do labour for lagging nodes. The peer downloads the data from S3 rather than from the lagging node, to avoid choking the node even further.

## 6.8 Execution Engine

Columnar - works on columns lol.

Vectorized - works on many thousands of rows at a time. Doesn't do intermediate materialization.

Push-based execution - so I guess instead of processes continuously polling or "pulling" for results, the generating processes pushes them, is what I'm getting.

No transaction management since all files are immutable (what about race conditions?), no buffer pool due to scans.

## 6.9 Cloud Services

Workers are ephemeral, so something has to stick around semi-permanently to handle the database stuff.

Hence cloud services.

Snowflake doesn't use indexes?

Keeps track of worker performance for monitoring.

## 6.10 Pruning

So indexes are indeed not used, since the access pattern doesn't fit and it goes against the whole "user doesn't need to do jack" approach since it's the user that would have to create the indexes.

So these guys use zone mapping a.k.a. keeping coarse track of what is where.

# 7 Data page layouts for relational databases on deep memory hierarchies

NSM - n-ary Storage Model, A.K.A. slotted pages, stores tuples.

PAX (Partition Attributes Across) simply keeps attributes together, so I suppose it's columnar. Stores IDs for all of them, yuck.

"Paradox" - the size of requested chunk of data by user is not what is loaded in by engine, and the stuff that is loaded is largely irrelevant (if You just want an attribute and You get a tuple).

Decomposition storage model (DSM) splits everything into columns, but it sucks because You need to do joins to get tuples.

PAX stores what they call "minipages", which are just columns lol. If You want a tuple, it'll still be in the same page/block.

There are different kinds of minipages.

For records of fixed length, You have an F-minipage. The header for the minipage stores a bit vector to indicate nulls. No need for pointers due since You know where things are since everything is fixed length.

V minpage is basically just a page. Variable length entries, pointers to those entries.

# 8 Modern B-Tree Techniques

B tree contains data in branch nodes as well as leaf nodes, B+ trees just in the leaves.

Usually 100s of children so only like 1% of nodes are branch nodes, so keep those together.

Leaves use PCTFREE and PCTUSED type stuff - recall PCTFREE is the amount of space left for updating, and PCTUSED is the threshold such that when the amount of used space falls below it, the block is open for business again.

Insertion: in a normal tree, You might just replace a leaf with a branch when inserting. In B-trees, when the leaf is too full for an insert, the parent gets split (and then the parent's parent if necessary etc.), so the tree remains balanced.

Underflow can be actively managed, but studies suggest period compacting is better.

B trees great since there aren't that many branching nodes, and they can be kept in memory.

Optimal node size is calculated by taking the bandwith with the latency.

Interpolate between binary search and interpolation search.

## 8.1 B-Trees v.s. Hash Indexes

Branch nodes are few, getting to children is fast and B-Trees support ordering.

## 8.2 Normalized keys

Are interesting.

So You get a key, and instead of just storing the item with that key You go through the key and set various relevant bits, e.g. if the first bit in they is set, then the value stored is legit, if the second bit is set then blah blah

## 8.3 Prefix trees

If all keys start with "okay", then don't store okay 1 million times, just create a dictionary keeping track of prefixes. Basic compression.

# 9 The Design and Implementation of Modern Column-Oriented Database Systems

## 9.1 Intro

Implicit IDs store fixed-size items, so the position of the item in memory is basically an ID.

Late Materialization - don't join until You have to, since it's expensive in a column store, as in join columns to form tuples.

Writing to these might suck, since a tuple will be split into entries and those entries will go off and touch a lot of blocks. Solution is just to pool them.

Data is kept compressed.

A lot of stuff in column DBs is mapped over from row DBs, but where as column stuff was kind of appended on to the row implementations, the column DBs were designed with columnar storage from the get-go.

## 9.2 History

N-ary storage model, NSM, a.k.a. slotted pages with points to tuples.

DSM, decomposition storage model, splits columns into different pages, I think?l They get sub-keys in DSM, as in each sub-page now gets sub-keys which sub-keys map to tuples within the page, which increases space requirements.

PAX, ze hybrid, has it's own paper, see above.

Memory became way slower than processors very quickly, and the time take to retrieve an item v.s. time taken to process it went from 1:1 to 100:1

## 9.3 Column-store Architectures

### 9.3.1 C-Store

Each column gets a file, the columns are individually compressed based on data, and they are all sorted by the same attribute. Stuff stored like this is known as the Read Optimized Store, or ROS.

There's also WOS, which is just an uncompressed row-store.

WOS acts as a buffer, makes sense.

Can group columns and sort them by some other attribute - these are duplicate and called projections.

In other words, everything is stored as a column, but if it turns out that two columns are sorted on the same attribute, they form a group.

Usually at least one projection is just everything lol, as in all the columns.

Projection notation $(a, b, c|d)$ means a projection containing a, b and c columns sorted on d.

No secondary indexes, but does support sparse indexes - just metadata on starting value of attribute in page, given that the data is sorted.

### 9.3.2 MonetDB

Execution engine works on column-at-a-time.

DOES NOT COMPRESS COLUMNS lol.

Queries are optimized at runtime.

Prioritizes avoid cache misses.

Uses Database Cracking.

Read-only queries bypass transaction management hence no overhead.

They really do execute stuff column at a time, as in a whole ass column.

Strength reduction - replace an operation with an equivalent and less costly version.

Array blocking - I guess it's grouping parts of an array, which I guess in practice will be a column, such that it fits into a cache? Make array in cache friendly blocks.

Loop pipelining - starting execution of the next before the current is finished? Basically loops are all waiting for each other to finish, but there is no need to wait for one to completely finish, You can use intermediate results and get started.

They implement their own operators which saturate the CPU with sequential data, allowing for many in-flight instructions, increasing throughput.

Uses BATs - Binary Association Table. Key value pairs, basically. Thing is, the ID is usually virtual - signified by position in memory.

Operators consume and produce BATs.

So basically BATs are just arrays lol.

Operators themselves do not change in MonetDB - if You have some complicated join condition for example, the DB will just chain pre-made operators. This improves efficiency, allegedly.

"RISC approach to database query languages"

Some people even want to compile custom queries at runtime - trim all "if/else" branches for example.

MonetDB uses full materialization, wtf?

Does not use compression, which is wild.

Full materialization of intermediate results.

### 9.3.3 VectorWise

Almost nothing there really.

Late materialization, processing stuff blocks at a time.

## 9.4 Other

Frequency Partitioning - arranging data so there is minimal heterogeniety per block.

## 9.5 Column-store internals and advanced techniques

### 9.5.1 Vectorized Processing

Okay so classically We have two ways of processing a information - all at once, as in the entire table or whatever, or just one tuple at a time.

Or rather, operators call next() to get one tuple at a time, or You can have an operator consume everything and output one complete result. Downside is there may be large intermediate results.

e.g. calculate average - tuple at a time will shove, well, one tuple at a time to the average calculator. Full materialization will perform a full scan and give all the relevant tuples and then average - the second will have a larger memory footprint (but might be faster?)

VectorWise uses vectorized execution, which instead of doing one tuple at a time does N tuples at a time. Similar to doing BATs but at a small scale.

Great because it reduces the number of function calls (since now there isn't a call per tuple).

Obviously cache locality, particularly L1 cache, L1 being fastest, L3 being slowest.

Compiler optimizations thanks to optimizing loops

Oooh it's better for checking blocking - if You're checking whether something is blocked once per tuple, You'll be doing that an awful lot. Checking it once ever N tuples addresses this.

Computing N things at a time creates parallel cache misses, so the machine is aware of what is needed and can be fetching it (where as with iterative per-tuple execution, misses arrive sequentially so who knows what the next tuple will bring in terms of cache misses).

### 9.5.2 Compression

Compress one column at a time - exploits similarity of data in columns.

Improves performance since more data can be fetched quicker.

If compressed values can be processed, allows for parallel processing of a larger number of items.

Savings in space due to compression can be used to store projections.

### 9.5.3 Run-length Encoding

Sort by some key, then compress by storing triples of (value, start position, frequency). Start position keeps track of what belongs where.

Using RLE makes some things a little trickier - processing in blocks doesn't really work, since there isn't really data there, You're dealing with a more abstract representation.

### 9.5.4 Bit Vector Encoding

So, You have a column, and that column has some length.

Then, for each unique value in the column, You create a sequence of bits of length equal to the length of the column.

Then, for that unique value, set bitstring[i] = 1 if unique value occurs at index i in the column.

Do this for all unique values and You've got Your encoding.

### 9.5.5 Dictionary encoding

Instead of storing values that occur all the time, make a dictionary and store references to the dictionary instead, e.g. the string "death" can just be replaced with an index to a dictionary for "death" lol.

Main advantage is that it can take variable-length stuff and make it fixed-length, and all the good stuff that comes with that.

### 9.5.6 Frame of Reference

Ahh this is about storing a base - if all Your entries are five billion + stuff, just store the five billion as a base and truncate and store offsets from that.

A similar concept is delta-encoding, where to get the current value You take the value at the previous index and apply the current "delta" stored - so don't store the value, store the change needed to apply to the previous value to get current value.

### 9.5.7 The Patching Technique

Outliers mess with frame of reference as well as dictionaries, so split the data in two - stuff to be compressed and the wonky outlier block. Outliers just get to sit there uncompressed.

### 9.5.8 Computation without decompression

Upside is: more data, faster processing, more efficient cache utilization etc.

Downside is: code more complex - operators have to be aware of compression schemes to be able to operate on them.

The way this is done is that data properties are abstracted - get next item, get size etc.

### 9.5.9 Late Materialization

Naively, You can just store columns as columns and change nothing else if You load the data in column by column and join in memory, but that's crap since You're doing extra work. This is early materialization.

If You want to process columns independently, You'll need to keep track of what belongs to what tuple.

Good things about late materialization: if You delay things until You need them, You may avoid unnecessary intermediate completions.

Avoiding actual construction also permits data to remain compressed.

Cache is more effectively utilized since it isn't clogged up with data from other tuples.

And finally it's easier to parallelize execution of data in column form (which is yielded by delaying materialization).

Not always the best thing to do, though, late materialization. Because You're avoiding joining things too early, You'll have to keep track of indexes as mentioned before, keeping track of what part of what column belongs to what tuple, and if You're just selecting most of everything, that overhead can become expensive.

Multi-column blocks keeps various columns together, so again with the "not going all the way but staying in the middle" theme of storing rows v.s. storing columns.

It's like PAX but You don't store all the attributes on a block.

Multi-column blocks are good if You expect two attributes to be queried together often.

Used in IBM blink lol.

#### 9.5.10 Inserts/updates/deletes

Use buffers.

Downside is that You need to combine them, upside is You can maybe avoid the buffers talking to each other until the end, a.k.a. delayed materialization.

Jive Join - if the join results in disordered

#### 9.5.11 Indexing

So, scans are very efficient in column DBs, but that doesn't mean improvements can't be made.

Better to keep sorted arrays than to do a tree over memory, apparently.

Indexing - in C-Store called projections, columns ordered by different criteria. Doing this isn't as expensive as You might think, because the columns can be well compressed.

"Zonemaps" are also used, which is just metadata for the page like min/max, stuff that can be used to skip over the block if applicable.

#### 9.5.12 Database Cracking

Can't create all the indexes You want due to computational and storage constraints, so which ones should be made?

Turns out things are becoming more and more unpredictable w.r.t. use cases, which makes this worse.

So the idea is that the queries continuously create partial indexes, or optimize the current indexes for the current use, with the hope that it eventually converges.

### 9.5.13 Summary and Design Principles Taxonomy

Good table on page 264 here.

## 9.6 Comparing MonetDB/VectorWise/C-store

C-store tries to do column at a time processing lol, VectorWise and C-store do blocks (though containing columns)

# 10 A hybrid page layout integrating PAX and NSM

Stores things column-wise at the top of the page with pointers to variable-length stuff at the bottom of the page/

## 11 Sort vs. Hash Revisited

Optimizations: You've got fan-in, which is the maximum number of like data collections that can be joined to create other runs (probably dictated by something like available blocks to store things). So then of course the larger the blocks You're dicking around with, the fewer of them You'll be able to afford, the smaller the fan-in. Or You could think of it in terms of: larger blocks, fewer of them needed to hold entire data. Turns out bigger blocks seem to work better, even if it results in smaller fan-in and consquently maybe more layers of fan-ins.

Another one is to keep in mind what You actually have left to do. Merge smallest blocks available first, I suppose, is the deal. The example they give is neat.

Early aggregation good.

Choice between hashing and sorting depends on relative sizes of the tables. More equal size tables = sorting I think.

Second, stuff depends on key distribution.

## 12 Query Optimization

Two stages: rewriting and optimization.

Rewriting changes the query text, ish. Replace view with base tables for example, flatten out base queries, write out possible permutations for predicates or whatever.

Planner orders the operations.

Algebraic space - space of possible operator orderings.

Method-structure space - available implementations for an operation e.g. hash join v.s. sort and merge.

Each implementation has a cost model associated with it.

There's also a size-distribution estimator, basically trying to eyeball the size of query results via histogram.

# 13 The State of the Art in Distributed Query Processing

Challenges to distributes processing: heterogenous hardware, huge systems, rapid changes in workload, legacy systems.

Advantages: cost and scalability.

Parallel systems - just parallelizing execution I guess.

## 13.1 Architecture of a Query Processor

Parser - take string of query, translate it into code objects/queries.

Query Rewrite - Redundant predicates, views, flattening queries, whatever.

Optimizer - decides operation order, decides how much memory is required apparently.

Code gen - generate assembly that executes query tree.

Query execution engine - the bit that runs the assembly. Accepts calls to next() I suppose.

Catalog - all the database shit - segments, extents, page locations, views, whatever. Logical information v.s. physical information

## 13.2 Query Optimization

Dynamic programming tries to find optimal solutions.

Iterative to find iteratively better solutions.

Construct plans bottom up, keeping track of possibilities.

In centralized systems cost = CPU + IO.

IO: seek + latency + transfer.

Distributed systems have to take messaging costs, transfer costs (different IO I suppose), CPU costs to unpack/decompress maybe.

Planner ought to maybe consider intra-query parallelism e.g. cache shit or reusing whatever.

Cost does not account for response time.

## 13.3 Query Execution Techniques

Row blocking - send blocks okay lol.

Optimization of multicasts - just optimize paths it sounds like, don't route the data through paris.

Semijoin - start by shipping columns required for joining comparisons, then only send back stuff that is relevant.

Pointer-based joins: foreign keys are pointers, so a lot of random access.

## 13.4 Client-Server database systems

Architectures:

Peer to peer - everyone can act as a server and as a client.

Strict - there are servers, and clients. Commercial I suppose.

Middleware - hierarchical strict. A layer will act as a server for layers below and a client for layers above.

Strict makes things cleaner - privileges, types of machines etc.

## 13.5 Exploiting Client Resources

Query shipping - send query to data, send back answer.

Data shipping lol

Hybrid shipping - allow to process queries and store data at both ends e.g. client cache

## 13.6 Query Optimization

1. Select where what will happen - query v.s. data shipping.

Two-step optimization: at compile time generate operator tree. Then update just before running w/ site selection.

## 13.7 Heterogeneous Database Systems

Mediator figures what goes where, wrappers account for heterogeneity.

Plan the same but change costs depending on hardware

Calibration - estimate costs by running sample queries.

Individual wrappers - just give them individual costs.

Learning curve - learn at runtime.

## 13.8 DYNAMIC DATA PLACEMENT

### 13.8.1 Replication vs. Caching

Replication: explicit, benefits many, replicates a whole lot, stored on disk, removal explicit, updates instead of removal, is done by people.

Caching: fine, on memory, removal implicit, replicates small amounts, is done by CPU lol.

### 13.8.2 Dynamic Replication Algorithms

Goals: reduce communications, spread hot blocks.

ADR: replicate objects in replicant data must pass through venue.

Expansion test: If more reads originate in neighbours and pass through node than writes, just add it to replication.

Similarly: drop node if more writes and reads pass through

### 13.8.3 Cache investment

Once caching a.k.a. data shipping would have saved more data, make a copy.

# 14 Practice

## 14.1 Storage

Physical independence - database hides how data is actually stored on disk - pages v.s. idk like the cloud or something.

Logical independence - the same data can be viewed in different ways - the "logical" structure of the underlying table and the table presented to the user by the database are independent.

When is shared nothing good?

If You can partition the data cross nodes and most updates don't involve all data and queries can be parallelized i.e. queries don't require merging data across nodes.

Shared memory - a sort of lower-level shared nothing where the nodes get to pretend the memory between nodes is the same, though actually it's being shuffled around.

Shared disk - what it says on the tin. Lots of data shipping. Snowflake is in here somewhere.

Tablespace - abstraction to keep track of objects like tables, indexes, whatever. Memory is allocated to tablespaces, which is interesting, and stuff in a tablespace is broadly speaking supposed to be spatially local.

Segments are a collection of extents - they "look" contiguous, but the extents are allocated as needed so no dice.

Partitioning a table will result in numerous segments.

Size of extent allocated can change, ofc.

Extents function as a sort of in between of individually managing the memory for each tuple and just allocating one huge file for the table.

Slotted pages - header contains info about table and block, as well as points to tuples (depending on storage format).

PCFREE - Percentage Free - percentage of block that is to be kept free for updates.

PCUSED - if the percentage of space used in block falls below PCUSED, new tuples are allowed on the block.

I think they sort of flip a bit someplace - hit the ceiling and take the block off the free list, hit PC used and

Free lists live on segments.

Shadow paging - create duplicate tuples, modify those, on commit switch pointer. Good for recovery, concurrency. Bad for locality, book keeping.

Delta file basically the same, except the changes are made in place.

Buffer cache - area of memory reserved for caching blocks in memory.

Latches for blocks in memory since things can go wrong if multiple queries or transactions are looking for an item, e.g. updating an item, removing an item from cache etc.

Hash on block number, and a latch covers a number of hash buckets.

Linked list contains buffer headers, so You go until You find Your block and then use the buffer header to get the block.

## 14.2 Access Methods

How is a tuple stored, anyway.

It's got a header, which contains whatever,

Then You can store it attribute by attribute or store pointers at the start. Store long variable stuff at the end.

If attributes v. large, the store a pointer to those and put the large objects someplace else (BLOBs a.k.a. Binary Large Objects).

Decomposition storage model - split tables in columns (so each columns gets id + column) and store these new columns as before, thus getting columnar storage.

Column stores tend to use the cache better since You're not loading in the irrelevant attributes.

Extensible hashing: use first n bits for buckets, then split when a bucket overfills.

Linear hashing: just practice it before the exam.

Composite index: just index on multiple attributes and define an order like in any other case - composite index on A, B means the tree will have something like A0 B10 < A1 B0 or whatever.

Range lookup in trees finds the starting leaf node and traverses the leaves - no need to do a Looping Hierarchical access pattern.

There are a bunch of reasonable ways to deal with insertion and deletion. You can lock a child and parent and that way You know You'll be able to split a parent, but in theory You'll have to split all the way up to the root, so maybe try the first approach, if fail then lock entire tree.

Bulk inserting into a B-Tree: don't just create an empty one and go to town. Sort entire data, then create nodes bottom up. Also may want to consider PCFREE when creating those blocks.

B-trees suffer from hot blocks same as dictionaries. In order to avoid problems with say sequential inserts, reverse the sequence number to create a semi-random pattern.

Then come smaller optimizations like store just the minimal amount of information needed to decide which node to take.

Materialized aggregates - keep track of stats lol, like average, max etc.

Denormalized tables - just tables with a join pre-applied.

Clustered indexes - stuff in index is in order in memory.

Clustered tables - stored different tables on the same segment I think? Data used together to be kept together.