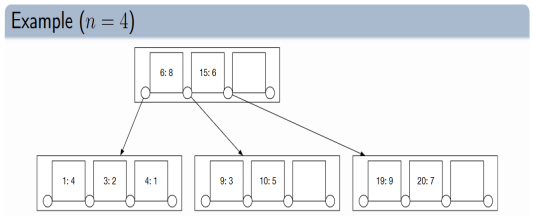
AT23 Data Processing Systems (COMP60029 @ Imperial College London) by ck21

Storage Models

***N*-Ary Storage Model (NSM)**. DBMS stores all attributes for a single tuple contiguously. Ideal for OLTP workloads where transactions tend to operate only on an individual entity and insert-heavy workloads. Tuple-at-a-time Iterator Model. Advantages: Fast inserts, updates and deletes. Good for queries that need the entire tuple. We can also use index-oriented physical storage. Disadvantages: Not good for scanning large portions of the table or subset of attributes.
Decomposed Storage Model (DSM). DBMS stores a single attribute for all tuples contiguously in a block of data. Ideal for OLAP workloads where read-only queries perform large scans over a subset of table attributes. Advantages: Reduces amount of wasted work because DBMS only reads data it needs. Better compression. Disadvantages: Slow for point queries, inserts, updates, and deletes because of tuple splitting/stitching.
Delta/Main (Hybrid). Aims to exploit the nature of DB workloads - analytics on historical data in DSM, transactions on recent data in NSM. However, needs regular migrations which might lock the DB. Also, complicates lookups.
Metadata. Exploit patterns to be more efficient. Type, min/max values, histograms, autocorrelation. Sortedness and Denseness.
Variables-sized Datatypes. Maintain fixed tuple sizes (allows random access to tuples by position). Either overallocate space for varchars or store them out of place. In place storage is good for locality and simple but wastes space (bad for in-memory DB). Out of place storage is space conservative, bad for locality, complicated (tricky GC). We can use Dictionary Compresion. Before every insert into dictionary, check if value already present. If so, slide the insert and use the addr of existing value. Otherwise, insert the value.
Disks. Disks have larger pages (Kb instead of bytes), higher latency, lower throughput (IO bound), and OS gets in the way (limited file size → DBMS needs to map tuple_ids to files and offsets. Dominate costs (complicated IO mgmt strats pay off). Pages are large (each page behaves like a mini-DB for *N*-ary storage.
Buffer Manager. Manages disk-resident data - maps unstructured files to structured tables for RW. Makes sure files have a fixed size and safely writes data to disk when necessary. Buffers write in open pages (open for writing) - make assumption of 1 open page per relation (with slight overflow).
Pages. Unspanned (Small part of each page left empty): Achieves simplicity and random access performance (given a tuple_id, find record with single page lookup). However, this wastes space and we cannot deal with large records. No in-page random access for variable-sized records. `Space Consumption for n tuples c̄eil(data.size() / numberOfTuplesPerPage())`. Spanned (optimise for space efficiency): Minimises space waste and supports large records. Disadvantages: complicated, poor random access performance and no in-page random access for variable-sized records. `dataSizeInBytes t̄upleSizeInButes data.size(); numberOfPagesForTable c̄eil(dataSizeInBytes / pageSizeInBytes)`. Slotted Pages (Random Access for in-place NSM): Store tuples in-place *N*-ary format (spanned/unspanned), store tuple count in page header, store offsets to every tuple (offsets only need to be typed large enough to address page - 1 byte 256 addr 2 bytes 4096). Disk-based DBMS keep a dictionary per page - solves problem of variable sized records and allows duplicate information.

Algorithms

Joins. Cross products with a selection involving both inputs. Left Join RS returns every row in R even if no rows in S match. No match - columns of S filled with NULL values. Full Outer Join RS returns every row in R even if no rows in S match and also returns every row in S even if no row in R matches. Matching function: need not be equality (if it is, equi-join). May also be inequality join or anti join (! =). Others - Theta joins.
Join Algorithms. Nested Loop - $\theta(|L|x|R|)$. Can be halved if value uniqueness assumed. But simple, sequential IO and trivial to parallelize. Sort Merge - Invariants: Assume that the value on right is greater than value on left. All values succeeding the value on right are greater than value on right (No value beyond value on right can be equal to value on left). The value on the left has no join partners succeeding the value on the right. Cursor on the left can be advanced. $O(sort(L)) + O(sort(R)) + O(merge)$. Sequential IO in merge phase, tricky to parallelize, works for inequality joins.
Hash Join. Distinguish build-side (side buffered in HT) and probe-side (one used to look up tuples in HT). Hash Function - pure and no state. Known output domain (need to know range of generated values). Nice to have - contiguous output domain (do not want holes in output domain and uniform (all values equally likely)). MD5, Modulo Division, MurmurHash. Conflict Handling (slot filled but there's space) - need locality and no holes (all slots probed).
Probing. Linear - when slot is filled, try next one (d1), and next one (d2), and continue until you find a free one. Then, wrap around at end of buffer. Simple and great access locality, but leads to long probe-chains for adversarial input data. Quadratic - when slot is filled, try the next one, double the distance and continue until we find one that is free. Then wrap around at end of buffer. Simple, good access locality for first probes (and increasingly worse after that). First probes are still likely to incur conflicts. Rehashing - distributes probes uniformly (we can use hash function for probing as well). Simple, conflict probability is a constant. Poor access locality - challenge is how to make sure all slots are probed. Overall - Sequential IO on inputs (with Psuedorandom access to HT during build and probe). Parallelizable over values on probe side (though tricky for build). $\theta(|build| + |probe|)$ BC, $O(|build|x|probe|)$ WC.
HJ Practicalities. Hashing is expensive (esp good hashing). Slots are allocated in buckets (slots with space for 1 tuple). Roughly equivalent to rounding each hash value dwon to a multiple of bucket size. HT are arrays (occupy space). Usually overallocated by factor of 2. Probed randomly in probe phase. Assume that if HT does not fit, every access hash constant penalty.
Partitioning. Sequential Access is much cheaper than Random Access. The difference grows with page size (assume random value access cost c - seq value access cost $\frac{c}{pagesize_{os}}$). Assume HT does not fit in buffer page cache/pool (relation larger than half). May pay off to partition → apply partition fn on build and partition fn on probe (modulo 4). Perform HT build on partitioned build table and then do localised random access. We can parallelise processing of each of the smaller joins (disjoint), and the larger relation as well (only join overlapping portions).
Index. When you store or cache the result of first phase (Sort, Build). Secondary Storage - replicating data (opposite of normalization). Clustered/Primary Index - index used to store tuples of a table. No more than one per table. May use more space than a table but don't replicate data (no consistency issues). Unclustered/Secondary Index - an index used to store pointers to tuples of a table. Can have as many as you like per table. Don't replicate data (some consistency issues).
Hash-Indexing. For HJ, we build one-shot HT. No new tuples added during query evaluation. HT discarded after join and no need to worry about updating it. But persistent HT may grow arbitrarily large, so large overallocation needed. If fill-factor grows beyond x percent, rebuild (expensive). Can lead to nasty load spikes. Similar for deletes. Helps reduce number of candidates if not all columns indexed (eq selection).
HT Deletes. Used empty slots as markers for end of probe-chains (we want short probe-chains). On delete, a value has to remain in the slot of the deleted value - either leave the value and mark as deleted, or put another value there (last value in probe chain).
Bitmap Indexing. Bitvector - sequence of 1 bit values indicating boolean condition holding for elements of a sequence of values. E.g. $BV_{==7}([4,7,11,7,7,11,4,7]) = [0,1,0,1,1,0,0,1]$. But CPUs work in words (assume 8bit) - so becomes $128*0+64*1+32*0+16*1+8*1+4*0+2*0+1*1=89$. Bitmap indices are a collection of bitvectors on a column (1 for each distinct value in that col). Useful for few distinct values in col. Usually disjoint - every pos/row, exactly one val set to one. They reduce bandwidth need for scanning a column in order of size of the type of column in bits. Predicates can be combined using logical operators on BVs. Arbitrary conditions can also be indexed (e.g. $7 < n < 12$).
Binned Bitmaps. Have n bitvectors, each with predicate covering different part of value domain. E.g. Bin 1 - 0,7; Bin 2 - 8,20; Bin 3 - 20,255 (Assuming col type is byte). Make sure conditions span entire value domain. Problem - index cannot distinguish values in bit (unless bit contains only one value). Can only produce candidates and false positives need to be eliminated. EquiWidth : Bin Width = $\frac{max(column)-min(column)}{numBins}$. Limited use when indexing non-uniformly distributed data (FP in highly pop bins). Binning - resilient against non-uniformly distributed data but bin construction tricky - sort values and determine quantiles, rebinning (Δ Distr).
RLE. Sequentially traverse vector and replace every run of consecutive equal values with a tuple containing the value (RUN) and number of tuples (LENGTH). Works well on high-locality data and requires seq scan to find value at specific position. Length Prefix Summing - replace scan with binsearch. E.g. $(0,0,0,1,1,1,0,1)$ becomes $[(0,3),(1,3),(0,1),(1,1)]$ becomes $[(0,0),(1,3),(0,6),(1,7)]$. Limited updatability → B-Tree.
B-Tree. DBs are IO bound (on disk) → minimise number of page IO operations. Many equality lookups and updates, sol is to use a tree. DB trees have high fanout to minimise page IO. Definition - balanced tree with out-degree n (every node has $n-1$ keys) and (1) the root has at least one element; (2) each non-root node contains at least $\lfloor \frac{n-1}{2} \rfloor$ KV Pairs.



Balancing. Find right Leaf Node to insert (walk the tree) and insert val. If overflow, split into two halves. Insert new split element in parent (one in mid of split node). If overflow, repeat procedure. If parent is root, introduce new root.
Delete. Find value to delete. If in leaf node, delete. If internal node, replace with max leaf-node val from left child (remove val from leaf-node). If affected LN underflows, rebalance tree bottom up. **Range Scans**. Complicated. Many node traversals. Node sizes codesigned with page sizes. Translates into page faults.

B^+-Trees. Keep data only in leafs (no up/down). Link one leaf to next, inner-node split values are replicas of leaf-node values. Only have single kind of node layout.
FK-Indices. FK Constraints specify that for every value that occurs in an attribute of a table, exactly one value in PK column of another table. DBMS needs to ensure that constraint holds. On Insert/Update, DBMS looks up PK value. Instead of storing value, DBMS could store Ptr to referenced PK or tuple.

Query Processing Models

Volcano Operators. Scan - read table and return contained tuples one by one (operators are stateful). Projection - transform tuple into another tuple using projection function. Selection - return all tuples that satisfy a Boolean predicate. Difference (*Pipeline Breaker*) - forces us to read all inputs from one side before working on the other. Op that produces first correct output tuple only after all input tuples from one side processed. Grouped Aggregation - group all tuples that are equal (given proj fn) and calculate one or more per-group aggregates.
Estimating Buffer IO Ops. Scans read all pages of a relation. If buffer and all other buffers fit in memory - no IO. Otherwise, buffer accessed seq - num occupied pages (per pass), buffer accessed randomly/OoO - one page access per tuple. If all buffers in the fragment (combined) fit in memory - No IO. If not, hash probes - one page access per accessed tuple. Otherwise - num occupied pages per pass.
How to know if buffers fit in mem. Nested loop buffers and sorted relations - exactly their input ($N_{tuples} * TupleSize$). Assume perfect knowledge on IO cardinalities - $InputBuffSz = InputCardinality * TupleSize$. Also, `GroupingHashTableSize=Ceil[2 × GroupingCardinality × NumberOfAttributesInGroupingTable×4 Bytes]`. Less than Buffer Pool Size so ignore (no IO cost).
CPU Efficiency. Function Pointers cause pipeline bubbles (contro hzd). JMP sets ip to arb addr → CPU needs to read next instr from this address → next instr can only be read once JMP complete. We need to estimate Func Calls. Per Tuple - Scan (None, input read straight from Buffer), Sel/Proj (One for input read, one to apply predicate), CrProd Inner (One), CrProd Outer (One), Join (One (inline intermediate step)), GrpBy (Two), Output (One).
Bottleneck. So CPU is bottleneck not IO. Can we process queries without FuncCalls? Turn ctrl dep to data dep. Instead of processing tuples right away, buffer them. Fill buffer with lots of tuples. Pass buffer to next operator. But every operator is pipeline breaker → similar rules to volcano IO. If buffer fits into mem no IO. If buffer accessed sequentially - calculate number of accessed pages per pass. If accessed randomly or OoO, one page access per accessed tuple.
By-Reference Bulk Processing. Save bandwidth - we are copying a lot of data around. Instead of producing tuples, produce IDs (32b positions in buffer). When processing tuple, use ID to look up actual value.
Page Access Probability. Selectivity s is % of tuples touched, n is no tuples in page. Assume uniform dist → $p(s,n) = 1-(1-s)^n$. Now in ByRef BP of DSM → every oper proc exactly one col of tuple. In N-Ary, values of tuple coloc on page. Useless values also occupy space in buffer pool. DSM fixes this.
of data repeatedly vs going down whole cols or rows).

Query Optimisation

. **Introduction**. Start by generating correct plan. Query optimisers generate equivalent plans (semantic equivalence hard to prove).
Guiding Assumptions. Cost grow with cardinalities. Joins tend to improve. Selections tend to reduce. Aggregations tend to reduce. Data access more expensive than function evaluation.
Selection Pushdown. Selections can be pushed through joins if they only refer to attributes from one side of the join. Usually good opt - selections are pipelineable - depends on join costs (eg in presence of join index). We can also do selection ordering (select A first instead of B).
Peephole Optimisation. If plans distinguished by order of selections → will run into loop. Also branching. Potential for infinite optimiser loops and missed opportunities (local minima). Rule-based opt very brittle (missing rule → devastating consequences, inf cycles).
Cost-based Opt. Sum num tuples produced by operator (assumption). Simple approach → histogram of tuple count of unique value in col. To estimate, evaluate query on hist first and sum result. However, this assumes attribute indep, so we could be off by a lot. One way is to do multidimensional hist. Then for 1st select, sum over values in respective row; 2nd select, sum over values in respective cell over sum of values in the row induced by first select. But combinatorial growth.

Concurrency Control

Serialization. Illusion of exclusive access. Transactions that break are said to be conflicting. Must be executed serially (any order). System does not need to commit to serialization apriori. Atomicity guaranteed. Aborts must cascade to dependent transactions. Our system must ensure serializability (isolation): State of DB after all transactions must be executed such that it could have been the product of serial exec. Also, recoverability: committed transaction must not depend on effect of uncommitted transaction.
Anomalies. Dirty Read (Read after Write). Phantom Read (select > delete > select). Write Skew (update > update). Inconsistent Analysis (update > select > update > select). Lost Update (Txn1 read; Txn2 read and write then commit; Txn1 write and commit so Txn2 lost). Other challenge - limited concurrency. Serial exec would easily fix problems but not an option - underutilises cores and makes users wait. Approach - limit concurrency where necessary; go for max conc everywhere else; accept some anomalies and make acceptable anomalies configurable.
Isolation Levels. SQL prescribes four. Read Uncommitted (weakest level, anything goes). Readers just read, writes happen immediately and inplace (still wait for each other). Full parallelism for Read Only Txn. All anomalies are possible. Read Committed (stronger than read uncommitted, readers wait for writers to finish, all writes of a txn are visible at the same time (atomically)). Repeatable Read (readers guaranteed to see same value for same read). Serializable (strongest).
Locks. Locks - Shared R locks R[o1] , Exclusive W locks W[o2]. Multiple granularities (Key Range Level - hardest). Can be formalised in RW ops. 2Phase Locking - guarantees serializability but does not prevent deadlocks.
Deadlocks. Sol - acquire locks in global order. But, set of req locks not known a-priori. Possible Solution - run txn in 'dry mode' to determine lock-set, acquire locks, run txn for real. Timeouts: lock will be safe for predefined time; after timeout, acquisition of lock by another txn aborts one holding the lock. Cycle detection is a bit like garbage collection. In regular intervals, walk graph of transactions (incoming edges are locks, that are waited for; outgoing edges are held locks). If cycle detected, break by aborting transaction (usually the youngest).
Timestamp ordering. Attach two timestamps to every tuple: time of last W and time of last R. Also attach timestamp to every txn at its start: when reading a tuple, check if it has been written since start of txn. If so, abort txn (value is inconsistent with txn timestamp). When writing tuple, check if it has been read since start of txn. If so, abort txn (other txns may write based off value).
Optimistic Concurrency Control. Run txn without locks but buffer reads, inserts and updates instead of applying. Upon commit - check if DB is unmodified. If so, apply updates (with locks), if not, abort txn.
MVCC. Generalises OCC by keeping multiple timestamped versions around the same time.
Conclusion. Use TPL if serializability req'd. OCC if num conflicts low. MVCC if num conflicts high.

	Dirty Writes	Dirty Reads	Write Skew	Inconsistent Analysis	Lost Updates	Phantom Read
Read Uncommitted	No	Yes	Yes	Yes	Yes	Yes
Read Committed	No	No	Yes	Yes	Yes	Yes
Repeatable Read	No	No	No	No	No	Yes
Serializable	No	No	No	No	No	No

Stream Processing

Infinite Input. `SELECT * FROM BallPosition, PlayerPosition WHER distance(BallPosition.x, BallPosition.y, PlayerPosition.x, PlayerPosition.y) < .3 AND BallPosition.timestamp = PlayerPosition.timestamp`. Timestamps are monotonic - slightly off? Limited Mem + ∞ input → appx results. In volcano, tuples pulled from underlying operators. In SP, data sources push data into operators. Back Pressure → slow operators slow down plan. Make source buffers grow indefinitely (particularly bad in MT environments). Operators need way to communicate this up the stream.
Timestamp Order. Hard to guarantee in-order in DS (no global clock, easy bottlenecks). Soft order. 3 options - treat stream like seq of txn (input all tuples into persistent DB - but memory is finite, streams are not); make assumptions with lateness bound; make users provide guarantees (low-watermarks/punctuations).
Windows. LB is impl detail - Window queries make this part of semantics. `SELECT avg(x) OVER (ORDER BY timestamp ROWS BETWEEN 5 PRECEEDING AND 5 FOLLOWING) AS smoothedX`. Types of windows - size N elements in window instance, and slide N elements between start of two window instances. Sliding Windows (slide < window), Tumbling Wiondows (slide = window), Stream Sampling (slide > window). Session Window - group events that arrive at similar times, filtering out time without data. Timeout, Max Duration, Partitioning Key (opt).
Aggregation Fns. Sum, Count, AVG (Invertible), Min, Max, Percentiles (Non-Invertible). AVG is algebraic, PCTL is holistic. Non-invertible - easy to implement using ordered indices (trees) but highly inefficient. Use dedicated algo (2stacks).
Joins. Handshake (Similar to nested loop. Optimised for parallel window joins and only works for window q). Symmetric Hash Join (pipelineable version of HJ. No natural window-version and requires infinite mem). Filters (Bloom). To tune - config size $m \approx -1.44 * n * log_2(\epsilon)$; config num hash fn $k \approx \frac{m}{n} * log_e(2)$; Error rate $\epsilon = (1 - e^{-\frac{k*n}{m}})^k$ where m is the Number of bits for filter; n is the number of distinct elements, k is the number of hash functions and ϵ is the FP rate.

Advanced Topics

Adaptive Indexing. Dynamic Adaptation of Indexing Structures for fast lookups. But might mean one operator undos the work of another. Evaluate Range Predicates (Scan, Sort, Crack). Predicated Cracking → Two Cursors, Compare and Move Cursor - L if 0 R if 1. Sum and swap active and backup values.
Composable DMS. Due to increasing heterogeneity. Suboptimal for vertically integrated construction.