1 Storage

- N-ary Storage Model (NSM): Good for inserting and finding tuples (data locality) in transactional processing, usually on recent data.
- Decomposed Storage Model (DSM): Good for scan-heavy queries (aggregates/joins), usually on historical data.
- Hybrid Storage: Delta (recent data) + Main (historical data). But needs migrations that lock up database, and complicates lookup.
- Catalog metadata: Sortedness, Denseness, Type, Min/Max Values, Histograms, Auto-correlation
- In place storage: Overallocate space. Good for locality, simple, wastes space.
- Out of place storage: Space conservative, bad for locality, complicated (harder implementation + garbage collection)
- Out of place + Dictionary Compression: Before every insert, if value already present, use existing address.
- Data storage on disk: Pages are larger (each page like mini database for NSM), disks dominate cost by far
- Buffer manager: Manages disk-resident data (maps unstructured files to structured tables, writes to disk when necessary)
- Unspanned Pages: Simple, random access performance. But wastes space, cannot deal with large records, no in page random access for variable sized records.
- Spanned Pages: Minimise space waste, support large records. But complicated, no in place random access for variable sized records.
- Slotted Pages: In place NSM (spanned or unspanned). Store tuple count in page header. Store offset to every tuple at end of page. Pointer size: Byte for 256B, Short for 65,536B, Integer for 4GB.
- In-page Dictionary: Duplicate elimination, solves problem of variable sized records.

2 Algorithms

- Nested Loop Join: Simple, Sequential I/O, Trivial to parallel. $\theta(left \times right)$ or $\theta(\frac{left \times right}{2})$ assuming uniqueness.
- Sort-Merge Join: Sequential I/O in merge phase, Tricky to parallel, Works for inequality joins. $\theta(l \log l + r \log r + l + r)$ if unique.
- Hash Join: Build vs probe. Sequential I/O on inputs, Parallel on probe. But hashing CPU expensive. Slots often allocated in buckets with space for > 1 tuple. Usually over-allocated by factor of > 2. Probed randomly in probe phase so penalty if hash table cannot fit in memory. Used when one relation is much smaller than other. Only good for equi-joins. $o(|build| + |probe|), O(|build| \times |probe|)$.
- Hash function: Must be stateless, known output domain. Ideally contiguous and uniform output domain.
- Conflict handling: Locality but not too much, probe all slots (no holes).
- Linear Probing: Keep trying next slot. Simple, great access locality. But long probe chain for adversarial input.
- Quadratic Probing: Keep trying next slot but doubling the distance. Simple, initially good access locality but increasing worse. But first probes still likely to conflict.
- Rehashing: Simple, conflict probability is constant. But poor access locality. Cyclic groups ensure all slots probed.
- Partitioning: Invest an extra pass to partition. Only join overlapping partitions. Processing of each partition can be parallel.
- Clustered/Primary Index: Stores tuples of a table, max 1 index per table, may use more space than table but no data replication.
- Unclustered/Secondary Index: Store pointers to tuples of table, have as many as desired, some consistency issues.
- Hash-Index: Unclustered (Hashtable of Key to pointers to tuples) or Clustered (Hashtable of Key to Non-key atributes)
- Hash delete: Leave value and mark as deleted or replace value with last value in probe chain (if same hash).
- Bitvectors: Sequence of 1-bit values indicating a boolean condition holding for the elements of a sequence of values.
- Bitmap Indices: Collection of (usually disjoint) bitvectors on a column. Useful if there are few distinct values in a column. Reduce bandwidth needed for scanning a column and predicates can be combined using logical operators on bitvectors.
- Binned Bitmaps: Have n bitvectors, each with predicate covering part of value domain, spanning entire value domain. But cannot distinguish values in a bin, only producing candidates, false positives need to be eliminated.
- Equi-Width: Limited use when indexing non-uniformly distributed data as many false positives in highly populated bins.
- Equi-Height: Resilient against non-uniformly distributed data (false-positive rate is value independent). But construction tricky (sort values and determine quantiles) and distribution changes requiring re-binning.
- Run-Length-Encoding: Replace consecutive equal values with (value x length). Works well on high-locality data but requires sequential scan to find value at specific position. Replace length with length prefix summing to replace scan with binary search.
- B-Trees: A balanced tree, each node has n-1 keys, root has > 1 element, each non-root node $\geq \lfloor \frac{n-1}{2} \rfloor$ key/value pairs. Minimise number of IO operations using a high fanout. Supports range scans and updates. But complicated, causes many node traversals (page faults) and leaf pointers not used (most data in leaf nodes).
- B-Trees insert: Insert into correct leaf node. If node overflows, split node into 2, insert split element into parent. Repeat until parent is root, then introduce new root.
- B-Trees delete: If leaf, delete. If non-leaf, replace with max value from left child. Repeat until leaf. If leaf underflows, obtain a value from neighbour, make it new splitting key and move splitting key into node. On failure (neighbour node less than half full), merge with neighbour node and remove splitting key. If parent underflows, rebalance from parent.
- B^+ Trees: Fast range scan by storing all data in leaves. Inner node contains value copies, every leaf node contains pointer to next.
- Foreign-Key Indices: Store pointer to referenced Primary Key. Directly yields partner tuples in FK/PK joins. Not much downsides.

3 Query Processing Models

- Volcano Design Goals: Flexible, Clean Design, Maintainable, Developer Productivity, Slightly tunable, Extensible (add new operators easily), Good I/O behaviour (tuples consumed as soon as they are produced)

Operator	`void open()`	`optional<Tuple> next()`	`void close()`
Scan		`return i < input.size() ? input[i++] : {};`	
Project	`child->open()`	`return projection(child->next());`	`child->close()`
Select	`child->open()`	`for (auto c = child->next(); c.has_value(); c = child->next()) {if (predicate(c)) return c;} return {};`	`child->close()`
Union	`left->open()` `right->open()`	`auto c = left->next();` `return c.has_value() ? c : right->next();`	`left->close()` `right->close()`

Cross	`left->open()`
	`right->open()`
	`l = left->next()`
		`auto r = right.next(); if (r.has_value()) rbuffer.push_back(r);`
		`if (i==rbuffer.size()) {i = 0; l = left->next()}`
		`return l.concat(rbuffer[i++]);`
			`right->close()`
			`right->close()`

- Pipeline breaker: Operator that produces first output only after all inputs from one of the side is processed.

Difference:

Open	`left->open(); right->open();` `for (auto r = right->next(); r.has_value(); r = right->next()) buffer.push_back(r);`
Next	`for (auto c = left->next(); c.has_value(); c = left->next()) {` `if (find(buffer.begin(); buffer.end(), c) == buffer.end()) return c;} return {};`

Grouped Aggregation:

Open	`for (c = child->next(); c.has_value(); c = child->next()) {auto slot = hash(c[groupOn]);`
	`if (!table[slot].has_value()) { table[slot][0] = c[groupOn]; table[slot].resize(fs.size() + 1);}`
	`for (size_t j = 0; j < fs.size(); j++) table[slot][j+1] = fs[j](table[slot][j+1], c);`

Next	`while (i < table.size()) {auto slot = table[i++]; if (slot.has_value()) return slot.value();}`

- Volcano I/O: Scan = no. of pages. Pipeline breakers open() and next(): if buffer fits in memory then 0 else number of pages for sequential pass or number of tuples for random access. (Hashtables overallocated by factor of 2).
- Volcano Function Calls: Scan = 0. Select/Project = 1 (read input) + 1 (apply predicate) = 2. Cross Product = 1. Join = 1 to read input (inline hash/comparison). Group-by: 1 to read input + [1 to select group] + 1 for each aggregate function. Output = 1.

```
int select(Table& outputBuffer, Table const& input, int predicate, int attributeOffset) {
  for(size_t i = 0; i < input.size(); i++)
    {if(input[i][attributeOffset] == predicate) outputBuffer.push_back(input[i]);}
  return outputBuffer.size(); }
```

- Bulk Processing: Buffer tuples instead of processing imeediately, then pass buffer to next operator. Every operator is pipeline breaker.
- Bulk Processing I/O: Read/Write = no. of pages. Read/Write Buffer = same as Volcano (but buffer is bulk processing's buffer).

```
int select(vector<int>& outputBuffer, vector<int> const& candidatePositions, int predicate,
    int attributeOffset, vector<Tuple> const& underlyingRelation) {
  for (size_t i = 0; i < candiatePositions->size(); i++) {
    if(underlyingRelation[(*candidatePositions)[i]][attributeOffset] == predicate)
      outputBuffer[outputCursor++] = (*candidatePositions)[i]; }
  return outputCursor; }
```

- By-Reference Bulk Processing: Produce IDs instead of tuples. When processing a tuple use ID to look up actual value.
- By-Reference Bulk Processing I/O: $p = 1 - (1 - selectivity)^{\text{tuples per page}}$.
- By-Reference DSM Bulk Processing: Every operator processes eactly 1 column, saving even more bandwidth.

4 Query Planning & Optimisation

- Logical (Algorithm Agnostic) vs Physical (Algorithm Aware). Rule-Based (Data Agnostic) vs Cost-Based (Data Aware)
- General: Cost grows with cardinality - Joins increase cost, selections/aggregations reduce cost. Data access more expensive than function evaluation.
- Rule-Based Optimisation: Unprincipled, Brittle, Infinite cycles from contradicting rules, Often wrong as data not taken into account.
- Rule-Based Logical Optimisation: Selection Pushdown - Selection pushed through joins if only rely on attributes from one side of join. Selection ordering - `s(==)<s(<)<s(<>)`
- Peephole Optimisation: Simple, fast, verifiable, composability of rules. But potential for inifites loops and local optima.
- Cost-Based Logical Optimisation: Assume uniform distribution or use (multidimensional) histograms
- Rule-Based Physical Optimisation: Focused on hardware (cache-conscious partitioning). Indexing: B-Tree/Hashes for equality, B+-Trees/Bitmap for ranges, FK indices for joins.
- Cost-Based Physical Optimisation: Taking hardware + data + algorithm into account (Query Processing Models)

5 Transaction Processing & Concurrency Control

- Serialisability (Isolation): State after all transactions must be such that it could have been the product of a serial execution.
- Recoverability (Durability): Committed transaction must not depend on effect of an uncommitted transaction.
- Dirty Read (read after write): T1 select a from R where b = 1; T2 update R set a = 5 where b = 1; T1 select a from R where b = 1.
- Phantom Read (read deleted or newly inserted rows): T1 select * from R; T2 delete from R; T1 select * from R;
  or - T1 select count(*) from R where x > 19; T2 insert into R values (20,1); T1 select sum(y) from R where $x > 10$;
- Dirty Write (write after uncommitted write): T2 update R set a = 5 where b = 1; T1 update R set a = 9 where b = 1; T1 update R set a = 9 where b = 3; T2 update R set a = 5 where b = 3;
- Write Skew (update based on stale data): T1 update R set a = c where b = 1; T2 update R set c = a where b = 1;
- Inconsistent Analysis (Data modified in the MIDDLE of an aggregate)
- Lost Update: T1 with old as (select a where b = 1); T2 update R set a = 17 where b = 1; T1 update R set a = (select a + 4 from old) where b = 1;
- Read Uncommitted: Readers just read, writers wait for one another, full parallelism for read-only transactions, all anomalies possible.
- Read Committed: Readers wait for writers to finish. All writes of a transaction visible at the same time.
- Repeatable Read: Readers guaranteed same value for each read.
- Serialisable: Readers guaranteed full isolation, no anomalies possible.

	Dirty Write	Dirty Read	Write Skew	Inconsistent Analysis	Lost Update	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
Serialisable	No	No	No	No	No	No

- Locks: Shared/Read Locks vs Exclusive/Write Locks. Entire database vs table-level vs tuple-level vs key-range-level (hardest)
- Two-Phase Locking: Growing Phase & Shrinking Phase. Guarantees serialisability but does not prevent deadlocks.
- Deadlock detection: Acquire locks in lock-set in global order, Timeout, Cycle detection (abort youngest transaction)
- Timestamp ordering: Each tuple has a last write and read timestamp. Every transaction has a timestamp at the start. When reading, if written since start of transaction, abort. When writing, if read since start of transaction, abort.

```
std::unordered_map<string, std::map<Key, std::pair<Range, LockType>>> locks;
void LockManager::acquireLock(string table, Key key, Range range, LockType type) {
mutex.lock(); auto pc = prev(locks[table].upper_bound(key));
while(pc > locks[table].begin() && pc->first + pc->second.first > key &&
        pc-> second.second == LockType::write && type = LockType::write) {
    mutex.unlock(); sleep(); mutex.lock(); pc = prev(locks[table].upper_bound(key));}
    locks[table][key] = {range, type};   }
void LockManager::releaseLock(string table, Key key) {mutex.lock(); locks[table].erase(key); mutex.unlock(); }
void Table::insert(Tuple tuple, Transaction&& transaction) {
  std::scoped_lock<std::mutex> lock(mutex); storeage.emplace_back(transaction.start, transaction.start, tuple); }
bool Table::update(Key key, Tuple tuple, Transaction&& transaction) {
  std::scoped_lock<std::mutex> lock(mutex); if(transaction.start < storage[key].read) return false;
  storeage[key].write = transaction.start; storage[key].tuple = tuple; return true;
}
```

- Optimistic Concurrency Control: Run transactions without locks but buffer reads/inserts/updates instead of applying. Upon commit, check if database has been modified after transaction.start.
- Multi-Version Concurrency Control: Generalises OCC by keeping multiple timestamped versions around the same time.
- TPL if serialisability is required. OCC if expected conflicts is low. MVCC if expected conflicts is high.

```
void Table::insert(Tuple tuple, Transaction&& transaction) {
  std::scoped_lock<std::mutex> lock(mutex); storage.push_back({{transaction.start, tuple}}); }
void Table::update(Key key, Tuple tuple, Transaction&& transaction) {
  std::scoped_lock<std::mutex> lock(mutex); auto location = storage[key].begin();
  while(location->first > transaction.start) location++;
  storage[key].insert(location, {transaction.start, tuple}); }
Tuple Table::get(Key key, Transaction& transaction) {
  std::scoped_lock<std::mutex> lock(mutex);
  for(auto& [timestamp, tuple] : storage[key] {if(timestamp < transaction.start) return tuple;}) }
```

6 Stream Processing

```
class Select : PushOperator { PushOperator& plan; function<bool(Tuple)> predicate;
  float process(Tuple t) override { auto before = std::chrono::system_clock::now();
    if (predicate(t)) plan.process(t); return std::chrono::system_clock::now() - before; } }
```

- Database Analytics (ad-hoc queries over static data) vs Database Transactions (ad-hoc modifications) vs Stream Processing System (static queries over incoming data)
- Push-driven scheduling: Data sources push data into operators
- Back Pressure: Operators need a way to communicate buffers growing (by measuring time/buffer size) due to slow operators.
- Time: Processing Time - Acquired by every operator, not consistent, unpredictable, low-overhead.
  Ingestion Time - Acquired by first operator, passed as attribute, consistent, unpredictable, medium-overhead.
  Event Time - Provided externally, passed as attribute, consistent, predictable, highest-overhead. (Only correct approach)
- Out of Order Processing: Transactions - Treat stream like transactions, insert all tuples into persistent database.
  Lateness Bound - On stream arrival, delete all data before lateness bound. (Implementation detail for window-queries semantics)
  Watermarks/Punctuation: Whenever user inserts watermark, delete all data before max timestamp in other table.
- Windows: 2 parameters. Size (no. of elements in a window instance) & Slide (no. of elements between start of two window instances)
- Sliding Windows (slide < size) vs Tumbling Windows (slide = windows) vs Stream Sampling (slide > window)
- Session Windows: Windows that open and close through events eg users logging in / ball entering the field
- Aggregate Functions: Distributive (Min, Max, Sum, Count) Algebraic (Average) Holistic - Recalculate everything (Percentiles) Invertible (Sum, Count, Average) Non-invertible (Min, Max, Percentiles)
- Non-invertible aggregation functions: Ordered indices (trees) highly inefficient.
  Two Stacks Algorithm (Each element also contains min of that item and the min below it in the stack; Push - front stack; Pop - back stack, whenever back stack is empty reverse the front stack)
- Handshake Join: Similar to nested loop join, optimised for parallel window joins, only works for window queries.
- Symmetric Hash Join: On every tuple arrival, build one hash table and probe the other hash table. Pipelinable, no natural window version, requires infinite memory.
- Bloom Filter: False positives, finite memory, cpu cost for hashing.
  m = number of bits for filter, n = expected number of distinct elements, k = number of hash functions, ϵ = false positive rate.

$$m = -1.44 * n * log_2(\epsilon), \quad k = \frac{m}{n} * log_e(2), \quad \epsilon = (1 - e^{-\frac{k*n}{m}})^k$$

7 Advanced Topics

- Heterogeneity: Data (Relational, Document, Graph, Neural Network, Key-Value), Hardware (CPU, FPGA, GPU, TPU), Workload (Data Analytics, Transaction Processing, Mixed Workload, Model Training, Data Integration, Data Clearning, Inference)
- Classic Database Architecture: DBMS[SQL, Logical Plan (RA), Physical Plan (Tablescan, Hashjoin), DB Kernel (C code)], [OS]
- Query Compiler Architecture: DMS[SQL, Logical Plan], Compiler[Executable], [OS]
- Voodoo: Intermediate algebra between logical plan and executable with Data and Hardware Aware Optimiser
- Adaptive Indexing: Scanning vs Sorting vs Cracking (Partition as needed, lazy sort)

```
for(i=0;i<size;i++) {output[outl] = input[i]; outl += (input[i] < pivot);}
```

- Predication: turns control into data dependencies, no branch misprediction, causes unconditional I/O, only for out of place algos
- Predicated Cracking: Fastest Hoare Partition, no branching, no misprediction. Active and backup. Compare active to pivot, conditionally move 1 pointer, read next element, swap backup and active.
- Generalised Aggregation Graph: Prefix scan + suffix scan
- Composable Data Processing Systems: How to make it fast, scalable, efficient and correct.
- Challenge of increasing heterogeneity: Increased system complexity, Components have overlapping functionality (even redundant functionaltiy) - suboptimal for vertically integrated construction, Creates placement/scheduling problems.
- Opportunities of composable architecture: Strong seperation of concerns (components only interact through standardised interfaces), Users can make systems without overhead, Components can serve multiple purposes (optimisation, indexing, processing)
- Challenges of composable architecture: Defining interfaces is hard, Debugging components with overlapping functionality is harder, Overhead between components, Defining component concern/functionality is difficult.
- Problem of removing physical plan layer: Physical optimisation no longer possible, Mapping of logical operations to code may be unclear, Abstraction-gap violates good software engineering (huge codebase for component)
- Opportunities of adaptive indexing: Reduce cost of DBA, Adapt to workload shifts, Supports data exploration very well, Encourages users to "stay with a system" (soft vendor lock-in)
- Challenges of adaptive indexing: Increases load on system without guaranteed pay-off, Makes read-only operators read-write, May make one operator undo the work of another
- Adaptivity: Performing work in response to user behaviour & data (memoising results, building histograms when scanning, opportunistically compressing data)
- Challenges of integrating a predicated selection: Overlapping functionality is generally tricky, Highly hardware specific requirements (might be useful now but detrimental later when hardware or workload changes), Might only be beneficial for part of the data (not when data is sorted)