Fundamental Part: Buffer, Page, Replacement

Term	Definition	Scenarios
Buffer	RAM area for temp page storage (Bridges the speed gap between memory and disk, improve I/O efficiencency	All DBMS(Y:Fewer disk reads, faster queries X:RAM limited, eviction policy needed)Key to DBMS speed, enables pipelining
Page	The smallest unit of data transfer between disk and memory in databases.	Reduces I/O overhead by moving data in blocks rather than record-by-record (since Disk I/O is much slower than memory access(buffer critical)
Page Eviction Algorithm	which page to evict from buffer	LRU/MRU/LFU/CLOCK
LRU	evict least recently used	benefit when the most recently used data will be need again soon (generally used)
MRU	evict most recently used	particularly useful for scenarios where older data is more likely to be accessed again, such as when repeatedly scanning a file or in cyclic access patterns
LFU	evict least frequently used	helpful when a particular item has been used infrequently and might not need in future
Slotted Page Architecture	used in the page layout for DBMS storage(for managing records within each page on disk or in the buffer) **contains:** •Page Header: Metadata (number of slots, free space pointer, etc.) •Slot Director y: An array of pointers (offs -ets) to each record. •Records: Actual record data, pla ced wherever there's room (can be compacted later). •Free space: Area available for new/updated records. [Page Header][Slot Dir][----Records stored anywhere---][Free Space]	Default in most DBMS (Postgr -eSQL, Oracle, SQL Server) to store tables, especially those with variable-length records (e.g VARC -HAR, NULLs). •Performance: •Allows efficient inse -rtions/deletions without rewrites or shifting all other records.• Makes updates to variable-length fields (like VARCHAR) fast and simple. •Space Management: •Tolerates internal fragmentation; easy to reclaim space by compaction. •Record References:•Logic al references (page ID, slot number) are stable even if record data is moved. •Supports Advanced Features: •Enables features like multi-version con currency control (MVCC), where old versions of a record can kept alongside new ones.

Row Store vs Column Store

Features	Row Store	Column Store
Definition	Stores full rows together	Stores each attribute separately
Best for	Updates, single-row queries (like search one full row of record	Aggregates, analytics like only need to search one attribute
Bad for	full scan (take more time)	Slow for single-row writes
Example	MySQL, PostgreSQL	HBase, BigQuery

Index

Type	Definition	Use case	Good	Bad	Example
Clustered	Index where data order match index order; only one per table	Range scans, sorting, min/max queries	Efficient for range, minimizes random I/O	Only one per table; slow for random inserts	SQL Server, MySQL
Unclustered	Index where index order ≠ data order; can have many per table	Look up secondary key	Fast lookup, can be built on any attribute	Less efficient for range scans, more random I/O	PostgreSQL, Oracle
Dense	Entry for every record in the table	Small tables, high lookup speed	directlookup, supports fast search	Uses more space, slower for updates	ALL
Sparse	Entry for only some records	Large tables, low update frequency	Saves space, smaller index	may require extra page read, slower lookup	ISAM,B+-tree
Tree-based	Hierarchical structure (B+-tree, ISAM); supports search, insert, range queries	Range and equality queries	Balanced search, good for sorted/ range	Complex to maintain, overhead for updates	All major DBMS
B+-Tree	Balanced tree, all data in leaves; internal nodes route searches	Both range & point queries	Fast search, range, sorted, dynamic update	Overhead on frequent insert/delete	Oracle
ISAM	Indexed sequential Access Method; static, uses overflo w pages for insert	Stable, read-mostly data	Simple to implement, fast for static data	Overflow pages slow down reads, lose balance	Legacy, read-mostly
Hash-Based	Organize records into buckets using hash function on search key	Equality search (=)	Very fast for exact match	Not for range queries, bad with skew/collision	DynamoDB, MongoDB
Static-Hashing	Fixed of buckets, hash table size doesn't change	Stable, known-size datasets	Simple, predictable performance	no growth reorganize need if overflow	old DBMS
Extendibl-Hashing	Directory-based, allows table/bucket to grow as needed	Growing datasets	Scales, avoid global reorg	directory can grow large, indirection cost	ModernNoSql custom
Bitmap	use a bitmap for each possible value of an attribute (low-cardinality)	Gender, status, "flag" columns	Fast for AND/OR queries, small storage	Not for high-cardinalit y, not for updates	Oracle, PostgreSQL
Overflow chain	Linked list of overflow pages (for hash bucket or ISAM)	handling hash collision, ISAM inserts	simple collision resolution	Too many → slow lookup	ISAM, static hash
Bucket	storage location containing records that are share the same hash value	group record for fast access and implemen hash index			

Index examples

Clustered	CREATE CLUSTERED INDEX idx_city_id ON City(id); (Best for queries like WHERE id BETWEEN 10 AND 100) (id ordered)
Uncluster	CREATE INDEX idx_city_name ON City(name); (Best for WHERE name='XX') (name is not ordered)
composite(multi-column)	CREATE INDEX idx_city_code_pop ON City(countrycode, population); (Best for queries using both columns in WHERE/join)
Bitmap	CREATE BITMAP INDEX idx_status ON Employee(status);

Index Choosing

Type	Best Index	Why
Equality search (single col)	Hash or dense B+-tree	Fastest for exact match
Range query (<, >, between)	Clustered B+-tree	Supports range, sequential I/O
Frequent updates	Unclustered/Hash	Less disruption to table order
Low-cardinality attribute	Bitmap	Small, fast logical operations
Lookup by secondary attr	Unclustered dense index	Multiple per table allowed

CAP Theorem Table (also known as Brewer's Theorem):

CAP Theorem	Can only have two of: Consistency, Availability, Partition Tolerance		Forces design trade-offs	Cannot achieve all three	Drivesdesign ofNoSQL/cloud DBs
System	Consistency	Availiability	Partition Tolerance	Sacrifice	Example
DynamoDB	No	Yes	Yes	Consist	Amazon
HBase	Yes	No	Yes	Availiabil	apacheHbase
sql RDBMS	Yes	Yes	No	Partition	Orcale

Distributed Data Management (DDM): Comparison Table

Concept	Definition	When Best	Advantage	Disadvantage	Effect/Notes
Shared Memory	all processors access same physical memory	SMP servers, multi-core CPUs	sim progra, fast comms	Scalability limited by memory bus	Good for small clusters; hard to scale
Shared Disk	processors have private memory, share disk subsystem	Clustered databases (Oracle RAC)	Fault tolerance, easier failover	disk can be bottleneck sync overhead	Useful for moderate scaling, expensive HW
Shared Nothing	Each node has own cpu, memory,disk; no shared resources	Google Spanner, NoSQL clusters	Maximum scalability, fault isolation	Complex partitioning, network overhead	Most modern cloud DBs, best for big data
Horizontal Partitioning (Sharding)	Split data by rows (each partition=subset of rows)	Users by region/customer ID	Distributes load, enables parallelism	Can cause "hot spots" if skewed	localize query, but|co-partitioning need for joins
Vertical Partitioning	split by col (each partition = subset of attributes)	Sensitive data separation	Reduces row size, improved IO	more complx to manage, many joins	Used for wide tables, some time in OLAP
Replication	Store copies of data on multiple node	Master-slave, multi-master	fault toleren, high availabil	data consisten overhead, more storage	enable failover backup,scaling
Synchronous Replication	Update apply to all replica before comit	Financial data, banking	strong consistency, no data loss	high latency, slow writes	All nodes always in sync
Asynchronous Replication	Primary updates, replicas catch up later	Social media, analytics	Fast, low-latency writes	Possible data loss on failure	Eventual consistency, common in NoSQL
Cascading Replication	Replicas propagate changes to further replicas	Multi-tier geo-distributed DBs	Reduces master load, scalable	Replication lag can propagate	Failure/lags may "cascade" down
Log Shipping (Delayed)	Ship logs, apply after a set delay	Disaster recovery, undo mistake	Time window for recovery	Replica always lags, not up to date	Useful for accidental delete/attack undo
Partitioning Strategy	How data is split across nodes	Hash/range/round-robin	Performance, data locality	Data skew or uneven load	Affects join strategy,failover
Co-Partitioning	Partition related tables by same key	Fact/dim tables in warehouse	Fast local joins, less data shuffle	Needs design discipline, coordination	Essential for efficient-distributed joins
Distributed Join - Broadcast	Copy small table to all nodes	Small-dimension, big-fact	Simple, fast for small tables	Network overhead for large tables	Best for 1 big, 1 small table
Distributed Join - Shuffle	Partition both tables by join	Large tables, equi-join	Scalable, balanced load	high network cost for large	Main method in Spark/Flink

	key			tables	/MapReduce
Distributed-Join-Fragment-and-Replicate	Partition and copy as needed	Complex queries, non-equi joins	General purpose, handles any join	Most expensive, lots of data movement	Rare in practice, fallback strategy
Consistency Model	Guarantees about data visibility across nodes	CP, AP, CA (CAP theorem)	Predictable results, easier reasoning	Harder to scale, or availability trade-off	Each model sacrifices something

Query Processing & Execution: **Parsing → (Rewriting) → Logical Plan → (Logical Opt) → Physical Plan → (Cost-Based Opt) → Scheduling → Execution → Result**

Step	Name	What Happens	Key Operator/Concept	Why Important / Effect	Distributed/Big Data Note
1	Query Parsing	SQL is checked for syntax, translated to an internal form	-	Errors caught, query decomposed into sub-parts	Same, but may involve parsing at client/gateway
2	Query Rewriting	Applies heuristics, rewrites algebra for efficiency	σ, π, pushdown, combine, etc.	Reduces result size early, enables further optimization	Ensures filters/projects "pushed" to each node
3	Logical Plan Creation	Translates to relational algebra (σ, π, ⋈, ×, γ, etc.)	Relational algebra	Makes query machine-readable, enables logical optimization	Platform-agnostic, used in federated/parallel plans
4	Logical Optimization	Further applies algebraic rewrites for efficiency	Selection/projection pushdown	Further shrinks work needed, combines redundant ops	May "ship" filters to data nodes for early pruning
5	Physical Plan generation	Chooses physical operators: scan, join, sort, hash, etc.	Table/index scan, join algo	Determines how query will actually run, introduces costs	Considers data location, data flow, node assignment
6	Cost-based optimization	Estimates cost (I/O, CPU, network), picks cheapest plan	Cost model, statistics	Key to high performance; wrong choice = slow query	Network, partition stats added in distributed DB
7	Operator Scheduling	Assigns operators to buffers, RAM, CPU, nodes	-	Enables parallelism, controls memory/disk usage	Schedules tasks to cluster nodes/stages
8	Execution	Executes chosen plan, operators process data	Materialization/pipelining	Results are produced, temp/intermediate storage managed	Task/stage execution on each worker, DAG for Spark
9	Result Format/Delivery	Formats result for client/user (e.g., table, chart)	-	Results are returned in the form the user expects	May involve collation/gather at coordinator

Distributed Data Processing & Platforms

platfo+method	Definition	When Best	Advantage	Limitation	Effect/Notes
MapReduce	Batch model: map, shuffle, reduce	Large, parallel batch jobs	Scalable, fault tolerant, simple	Always materializes, high latency	Good for ETL, slow for interactive
Spark	In-memory, DAG-based batch and streaming	Iterative, analytical, Machine Learning	Fast (in-memory), supports DAG, streaming	Needs RAM, not all ops can pipeline	RDDs (resilient distributed datasets)
Flink	True stream & batch, low-latency	Real-time analytics	Handles event time, out-of-order	Needs tuning, newer than Spark	Operator chaining for pipelining
Batch Processing	All data valid, processed in one shot	Reports, ETL	Simpler, deterministic	High latency, not interactive	All MapReduce is batch
Stream Processing	Process data as it arrives	Real-time monitoring, IoT	Low latency, up-to-date analytics	Approximate results, memory use	DSMS, Flink, Spark Streaming
DSMS (vs. DBMS)	Data Stream Mgmt:continuous,unbounded, low-latency	IoT, fraud detection	Real-time queries, windowing support	Limited query power, approximate	No tables, supports time-based logic

NoSQL Data Models

Type	Definition	When Best	Advantage	Disadvantage	CAP part
Key-Value	Get/set by key only	Shopping carts, cache	Simple, scalable, high throughput	No query by value, just key	AP(DynamoDB, Redis)
Column	Sparse columns, wide rows	Analytics, IoT	Efficient for aggregates, scalable	complex query hard,denormalized	CP (HBase, Cassandra)
Document	JSON-like documents	Web apps, catalogs	Flexible schema, semi-structured	Joins/ACID hard, big docs slow	AP/CP (MongoDB, CouchDB)
Graph	Nodes & edges	Social network, recommendations	Powerful for relationships	Not as scalable, niche use	CP (Neo4j)
Consistency Models	Strong, eventual, etc.	Varies by system	Configurable for need	Weaker consistency if available	NoSQL trades CAP, usually AP/CP

Extra Concepts:

SSD vs. Disk	SSD: fast random I/O, low latency; Disk: cheap, slow	SSD: random I/O fast, less seek	SSD:write-wear, cost; Disk: slow	I/O optimizations (sequential read) matter less on SSD; batch/write minimization always matters
DSMS vs. DBMS	DSMS:continuous, window/aggregation;DBMS:static/tables	DSMS: real-time; DBMS: complex queries, joins	DSMS: less power, less durable	Use DSMS for IoT, monitoring; DBMS for OLTP/OLAP
Window (Streaming)	Subset of stream, e.g., 5-min or count of 1000	Enable-aggregation, joins over infinite data	Memory cost, window management	Essential for stream analytics
Join in Distributed	Broadcast, shuffle, fragment-replicate	Broadcast: fast if small; Shuffle: scalable	Broadcast: high net cost for big; Shuffle: latency	Pick based on table size & filter

Relational Algebra:

SQL Feature	SELECT cols	WHERE cond	JOIN	GROUP/AGG	UNION	INTERSECT	EXCEPT
Rel. Algebra	π	σ	⋈	γ	∪	∩	−
Example	π_name(Employee)	σ_age>30(Employee)	R ⋈_{A=B} S	γ_dept,avg(sal)(Employee)	R ∪ S	R ∩ S	R − S

Example: SQL: SELECT name FROM E WHERE dept='Sales'= π_name(σ_dept='Sales'(E))