ADBS tx 3

Distributed Database: appears to user like centralized DBMS
nodes over network, possible no homogenity
+: no single point of failure, isolate faults, replace nodes, load balancing
shared Memory: common memory —: memory contention
shared Disk: common disk, SAN —: congestion
shared Nothing: linear speed up, linear scale up
Fragmentation: relation → smaller fragments Allocation: distrib. frag, replication
Replication: several copies, repl. factor: # of copies
Distribution Transparency: location, fragmentation, migration, concurrency, failure tr.
Failures in DBMS: server failure, Link f., Message f., Network partition-difficult
Availability: Probability System is operational Reliability: Sys. is failure free
Fault tolerance: Failures happen, detect failures, no inconsistent data, cont. operation
Goals of Fragmentation/Replication: Data locality, ↓ communication cost
small sets → improved efficiency, load balancing: — find compromise
Fragmentation: Horizontal/Sharding, $R = R_1 \cup R_2 \cup R_3$ / Vertical/Colums $R = R_1 \bowtie R_2 \bowtie R_3$
Non Redundancy/Disjointness $R_1 \cap R_2 = \emptyset$ / $PK(R) = attr(R_1) \cap attr(R_2)$
Granularity of Frag.: not too coarse/fine, workload aware, local frag. criteria
+Replication: availability, read performance —: consistency, concurrency
Primary-Secondary Repl.: write to primary + no concurrency — bottleneck
Multi Primary: synchronize servers + write availability — consistency, concurrency
Atomicity Consistency Isolation Durability
Distributed Concurrency Control deadlock detection, Distr. Recovery after fail.
Distinguished Copy Lock, Voting majority/timeout —: more traffic, complex
→ at Primary Site for all copies, Primary copy on different servers
Dist. Deadlock Detection: Centralized: send waits-for-graph to central site
Hierarchical: send wfg to parent Time-Out: abort after timeout

Two Phase Commit: coordinator —prepare→ subordinate yes/no force write log —commit/abort→ coord. force write log —ack→ subord. write end log
subordinate yes/no write log —abort/commit→ subord. ack

Join: $\Pi_B(R) \to S$, $S' = \Pi_B(R) \bowtie S \to R$, $R \bowtie S' = R \bowtie S \to$ to site 3

+ of RDBMS: formal foundation (logic, rel. algebra), standard Query language
concurrency, multi user, integration Database
- of RDBMS: schema known in advance, structured data/column type, split into Tables, ...
Want: nested Structure/JSON, graphs, schema evolution, sparse data/many nulls
→ application DBS → REST API — flexible, scale out
NoSQL: distributed, schemaless, non-relational, no joins, horizontally scaleable
Basically Available Soft State Eventual Consistency — no transactions
Aggregates: key value opaque, Document Store JSON, Column Store, Graph Relations, Nodes
Arbitrary Content Structure combination of C. combination
Consistency — like R/W on single copy Availability Partition Tolerance — continue operation
AP or CP, CA is impractical, PACELC Latency Consistency, PA/EC or PC/EL
ROWA read one-write all: $W = N$, $R = 1$, Majority: $W \geq \lfloor N/2 \rfloor + 1$, $R \geq \lceil N/2 \rceil$
Lamport Clock $C_j = \max(C_j, C_i) + 1$ 0 — 1 — 5
Weak Clock: $e_1 \to e_2$ then $C(e_1) < C(e_2)$ 0 — 2 — 3 — 4 — 5
Strong Clock: $e_1 \to e_2 \leftrightarrow C(e_1) < C(e_2)$ 0 — 1 — 6
Vector Clock: $VC_j[k] = \max(VC_j[k], VC_i[k])$, then own counter $VC_j[j] = VC_j[j] + 1$
strong

Key Value: value is array of bytes — meaning only known to app, scaleable, available
Session: user → session data, Logs: Date → Logfile, Ad: Campaign ID → Ad content
Riak: Erlang, masterless, available/fault tolerant, tag+Link values in Metadata
when Bucket/Namespace is created: set N #of Replicas, R-ead Quorum, W-rite Quorum
Document Store: understands value XML, JSON, BSON no joins → denormalize
MongoDB for log data, product catalogue, CMS data, collection ~ table, document ~ row
1:n rel: one-millions → foreign key one-thousands → array of references one-few → embed
n:m rel: three collections like SQL, or 2 with reference array embedded
Denormalization: — updates not atomic, + avoid join + when high read to write ratio

MongoDB primary-secondary writes on primary → propagate to secondary, error → vote new primary
write concern - when is write successful, majority: ⌊N/2⌋+1, read concern e.g. majority
Sharding by Shard Key - Hashed key or Ranged key, access through Router

Graph DB Edges directed, optimized for Graph Traversal, Nodes + Edges have props
Use Case: Customer - products, companies - ownership, people - places, social network
+ edges stored as pointers - no slow down if big - no index, no schema, easy to adapt/change
Graph in RDBMS: Node * Table, Edge Table, Node property T., edge prop. T.
or one Table <id, property key, value> eg <4, Name, 'Bob'>, <5, since, 1990>
Transform ERD to Graph: Table → NodeLabel, Row → Node, colum → N. prop., FK → Relations.
Neo4J: ACID compliant, several primaries, secondaries replicate asynchronous
Caching, eventual consistency, sharding difficult - different secondaries cache in RAM

Cypher: declarative, pattern matching, () node, (p) match var p against node
(:Person) node Label 'Person', (p:Person {name: 'Tom Hanks'})
-->/<-- directed rel. -- undirected rel, -[:ACTS_IN]-> rel. type ACTS_IN
Match (a)--() where (a)-[works_in]->(:City) Return a.name
-[:ACTED_IN*1..3]- 1 to 3 hops Match p=((foo)--(bar)) return nodes(p)/rels(p)
p = ShortestPath ((x)--(y)) Return nodes(p), length(p), relationships(p)
Creates erstellt Merge updates/creates
Match ... set p.age=23 Match .... Delete p
Aggregate: avg, sum, max, min, count(*), collect(p.name) - List of values, order by
Constraints: uniqueness, existence, property key

Colums stores: vertical fragments / no rows, virtual id or explicit id SIMD
+ aggregation, only needed columns - less IO, CPU heavy - less disk, vectorch
Explicit ID: larger Data → more IO Virtual ID: Lookup with offset: start + n·width
Compression Run length encoding: <value, #repetitions> can be worse than no comp.
Bit Vector: n Vectors for n distinct values, 1 if = value, else 0
Dictionary Encoding: Dictionary with values, column has dic keys
Null Suppression for sparse data, Vector has 0 for Null, separate dictionary
++ if I can operate on compressed data, e.g. RLE sum = value * #repetitions
Distinct values still good: RLE 20, Bit Vec. 30, Dictionary 60-100, LZ 100-∞
Date Column → RLE, Month → Bit Vector, Adress → Dictionary
Late Materialization: operate on individual columns, join late
Updates: expensive on compressed columns - decompress/update/compress from both
→ write to write store, flush periodically to read store, read/merge
Wide Column: HBase, Cassandra Google Bigtable: scale, continuous updates, lots of IO
Distributed, multi-level, sparse map: Row key - column - timestamp
ordered by key, grouped into Tablets, columns in column family

| | column contents | c.family anchor: cnnsi.com | qualifier | timestamp |
|---|---|---|---|---|
| row com.cnn.www | <CNN> .. <p> .. | cnn.com | | |

Lookup: return most recent values / return timestamp range