
Glossary



Please note that the definitions in this glossary are short and simple, intended to convey the core idea but not the full subtleties of a term. For more detail, please follow the references into the main text.

asynchronous

Not waiting for something to complete (e.g., sending data over the network to another node), and not making any assumptions about how long it is going to take. See “[Synchronous Versus Asynchronous Replication](#)” on page 153, “[Synchronous Versus Asynchronous Networks](#)” on page 284, and “[System Model and Reality](#)” on page 306.

atomic

1. In the context of concurrent operations: describing an operation that appears to take effect at a single point in time, so another concurrent process can never encounter the operation in a “half-finished” state. See also *isolation*.
2. In the context of transactions: grouping together a set of writes that must either all be committed or all be rolled back, even if faults occur. See “[Atomicity](#)” on page 223 and “[Atomic Commit and Two-Phase Commit \(2PC\)](#)” on page 354.

backpressure

Forcing the sender of some data to slow down because the recipient cannot keep

up with it. Also known as *flow control*. See “[Messaging Systems](#)” on page 441.

batch process

A computation that takes some fixed (and usually large) set of data as input and produces some other data as output, without modifying the input. See [Chapter 10](#).

bounded

Having some known upper limit or size. Used for example in the context of network delay (see “[Timeouts and Unbounded Delays](#)” on page 281) and datasets (see the introduction to [Chapter 11](#)).

Byzantine fault

A node that behaves incorrectly in some arbitrary way, for example by sending contradictory or malicious messages to other nodes. See “[Byzantine Faults](#)” on page 304.

cache

A component that remembers recently used data in order to speed up future reads of the same data. It is generally not complete: thus, if some data is missing from the cache, it has to be fetched from some underlying, slower data storage

CAP theorem

system that has a complete copy of the data.

CAP theorem

A widely misunderstood theoretical result that is not useful in practice. See “[The CAP theorem](#)” on page 336.

causality

The dependency between events that arises when one thing “happens before” another thing in a system. For example, a later event that is in response to an earlier event, or builds upon an earlier event, or should be understood in the light of an earlier event. See “[The “happens-before” relationship and concurrency](#)” on page 186 and “[Ordering and Causality](#)” on page 339.

consensus

A fundamental problem in distributed computing, concerning getting several nodes to agree on something (for example, which node should be the leader for a database cluster). The problem is much harder than it seems at first glance. See “[Fault-Tolerant Consensus](#)” on page 364.

data warehouse

A database in which data from several different OLTP systems has been combined and prepared to be used for analytics purposes. See “[Data Warehousing](#)” on page 91.

declarative

Describing the properties that something should have, but not the exact steps for how to achieve it. In the context of queries, a query optimizer takes a declarative query and decides how it should best be executed. See “[Query Languages for Data](#)” on page 42.

denormalize

To introduce some amount of redundancy or duplication in a *normalized* dataset, typically in the form of a *cache* or *index*, in order to speed up reads. A denormalized value is a kind of precomputed query result, similar to a material-

ized view. See “[Single-Object and Multi-Object Operations](#)” on page 228 and “[Deriving several views from the same event log](#)” on page 461.

derived data

A dataset that is created from some other data through a repeatable process, which you could run again if necessary. Usually, derived data is needed to speed up a particular kind of read access to the data. Indexes, caches, and materialized views are examples of derived data. See the introduction to [Part III](#).

deterministic

Describing a function that always produces the same output if you give it the same input. This means it cannot depend on random numbers, the time of day, network communication, or other unpredictable things.

distributed

Running on several nodes connected by a network. Characterized by *partial failures*: some part of the system may be broken while other parts are still working, and it is often impossible for the software to know what exactly is broken. See “[Faults and Partial Failures](#)” on page 274.

durable

Storing data in a way such that you believe it will not be lost, even if various faults occur. See “[Durability](#)” on page 226.

ETL

Extract–Transform–Load. The process of extracting data from a source database, transforming it into a form that is more suitable for analytic queries, and loading it into a data warehouse or batch processing system. See “[Data Warehousing](#)” on page 91.

failover

In systems that have a single leader, failover is the process of moving the leadership role from one node to another. See “[Handling Node Outages](#)” on page 156.

fault-tolerant

Able to recover automatically if something goes wrong (e.g., if a machine crashes or a network link fails). See “[Reliability](#)” on page 6.

flow control

See *backpressure*.

follower

A replica that does not directly accept any writes from clients, but only processes data changes that it receives from a leader. Also known as a *secondary*, *slave*, *read replica*, or *hot standby*. See “[Leaders and Followers](#)” on page 152.

full-text search

Searching text by arbitrary keywords, often with additional features such as matching similarly spelled words or synonyms. A full-text index is a kind of *secondary index* that supports such queries. See “[Full-text search and fuzzy indexes](#)” on page 88.

graph

A data structure consisting of *vertices* (things that you can refer to, also known as *nodes* or *entities*) and *edges* (connections from one vertex to another, also known as *relationships* or *arcs*). See “[Graph-Like Data Models](#)” on page 49.

hash

A function that turns an input into a random-looking number. The same input always returns the same number as output. Two different inputs are very likely to have two different numbers as output, although it is possible that two different inputs produce the same output (this is called a *collision*). See “[Partitioning by Hash of Key](#)” on page 203.

idempotent

Describing an operation that can be safely retried; if it is executed more than once, it has the same effect as if it was only executed once. See “[Idempotence](#)” on page 478.

index

A data structure that lets you efficiently search for all records that have a particular value in a particular field. See “[Data Structures That Power Your Database](#)” on page 70.

isolation

In the context of transactions, describing the degree to which concurrently executing transactions can interfere with each other. *Serializable* isolation provides the strongest guarantees, but weaker isolation levels are also used. See “[Isolation](#)” on page 225.

join

To bring together records that have something in common. Most commonly used in the case where one record has a reference to another (a foreign key, a document reference, an edge in a graph) and a query needs to get the record that the reference points to. See “[Many-to-One and Many-to-Many Relationships](#)” on page 33 and “[Reduce-Side Joins and Grouping](#)” on page 403.

leader

When data or a service is replicated across several nodes, the leader is the designated replica that is allowed to make changes. A leader may be elected through some protocol, or manually chosen by an administrator. Also known as the *primary* or *master*. See “[Leaders and Followers](#)” on page 152.

linearizable

Behaving as if there was only a single copy of data in the system, which is updated by atomic operations. See “[Linearizability](#)” on page 324.

locality

A performance optimization: putting several pieces of data in the same place if they are frequently needed at the same time. See “[Data locality for queries](#)” on page 41.

lock

lock

A mechanism to ensure that only one thread, node, or transaction can access something, and anyone else who wants to access the same thing must wait until the lock is released. See [“Two-Phase Locking \(2PL\)” on page 257](#) and [“The leader and the lock” on page 301](#).

log

An append-only file for storing data. A *write-ahead log* is used to make a storage engine resilient against crashes (see [“Making B-trees reliable” on page 82](#)), a *log-structured* storage engine uses logs as its primary storage format (see [“SSTables and LSM-Trees” on page 76](#)), a *replication log* is used to copy writes from a leader to followers (see [“Leaders and Followers” on page 152](#)), and an *event log* can represent a data stream (see [“Partitioned Logs” on page 446](#)).

materialize

To perform a computation eagerly and write out its result, as opposed to calculating it on demand when requested. See [“Aggregation: Data Cubes and Materialized Views” on page 101](#) and [“Materialization of Intermediate State” on page 419](#).

node

An instance of some software running on a computer, which communicates with other nodes via a network in order to accomplish some task.

normalized

Structured in such a way that there is no redundancy or duplication. In a normalized database, when some piece of data changes, you only need to change it in one place, not many copies in many different places. See [“Many-to-One and Many-to-Many Relationships” on page 33](#).

OLAP

Online analytic processing. Access pattern characterized by aggregating (e.g., count, sum, average) over a large number of

records. See [“Transaction Processing or Analytics?” on page 90](#).

OLTP

Online transaction processing. Access pattern characterized by fast queries that read or write a small number of records, usually indexed by key. See [“Transaction Processing or Analytics?” on page 90](#).

partitioning

Splitting up a large dataset or computation that is too big for a single machine into smaller parts and spreading them across several machines. Also known as *sharding*. See [Chapter 6](#).

percentile

A way of measuring the distribution of values by counting how many values are above or below some threshold. For example, the 95th percentile response time during some period is the time t such that 95% of requests in that period complete in less than t , and 5% take longer than t . See [“Describing Performance” on page 13](#).

primary key

A value (typically a number or a string) that uniquely identifies a record. In many applications, primary keys are generated by the system when a record is created (e.g., sequentially or randomly); they are not usually set by users. See also *secondary index*.

quorum

The minimum number of nodes that need to vote on an operation before it can be considered successful. See [“Quorums for reading and writing” on page 179](#).

rebalance

To move data or services from one node to another in order to spread the load fairly. See [“Rebalancing Partitions” on page 209](#).

replication

Keeping a copy of the same data on several nodes (*replicas*) so that it remains

accessible if a node becomes unreachable. See [Chapter 5](#).

schema

A description of the structure of some data, including its fields and datatypes. Whether some data conforms to a schema can be checked at various points in the data's lifetime (see [“Schema flexibility in the document model”](#) on page 39), and a schema can change over time (see [Chapter 4](#)).

secondary index

An additional data structure that is maintained alongside the primary data storage and which allows you to efficiently search for records that match a certain kind of condition. See [“Other Indexing Structures”](#) on page 85 and [“Partitioning and Secondary Indexes”](#) on page 206.

serializable

A guarantee that if several transactions execute concurrently, they behave the same as if they had executed one at a time, in some serial order. See [“Serializability”](#) on page 251.

shared-nothing

An architecture in which independent nodes—each with their own CPUs, memory, and disks—are connected via a conventional network, in contrast to shared-memory or shared-disk architectures. See the introduction to [Part II](#).

skew

1. Imbalanced load across partitions, such that some partitions have lots of requests or data, and others have much less. Also known as *hot spots*. See [“Skewed Workloads and Relieving Hot Spots”](#) on page 205 and [“Handling skew”](#) on page 407.
2. A timing anomaly that causes events to appear in an unexpected, nonsequential order. See the discussions of *read skew* in [“Snapshot Isolation and Repeatable Read”](#) on page 237, *write skew* in [“Write Skew and Phantoms”](#) on page 246, and *clock*

skew in [“Timestamps for ordering events”](#) on page 291.

split brain

A scenario in which two nodes simultaneously believe themselves to be the leader, and which may cause system guarantees to be violated. See [“Handling Node Outages”](#) on page 156 and [“The Truth Is Defined by the Majority”](#) on page 300.

stored procedure

A way of encoding the logic of a transaction such that it can be entirely executed on a database server, without communicating back and forth with a client during the transaction. See [“Actual Serial Execution”](#) on page 252.

stream process

A continually running computation that consumes a never-ending stream of events as input, and derives some output from it. See [Chapter 11](#).

synchronous

The opposite of *asynchronous*.

system of record

A system that holds the primary, authoritative version of some data, also known as the *source of truth*. Changes are first written here, and other datasets may be derived from the system of record. See the introduction to [Part III](#).

timeout

One of the simplest ways of detecting a fault, namely by observing the lack of a response within some amount of time. However, it is impossible to know whether a timeout is due to a problem with the remote node, or an issue in the network. See [“Timeouts and Unbounded Delays”](#) on page 281.

total order

A way of comparing things (e.g., timestamps) that allows you to always say which one of two things is greater and which one is lesser. An ordering in which

transaction

some things are incomparable (you cannot say which is greater or smaller) is called a *partial order*. See “[The causal order is not a total order](#)” on page 341.

transaction

Grouping together several reads and writes into a logical unit, in order to simplify error handling and concurrency issues. See [Chapter 7](#).

two-phase commit (2PC)

An algorithm to ensure that several database nodes either all commit or all abort a

transaction. See “[Atomic Commit and Two-Phase Commit \(2PC\)](#)” on page 354.

two-phase locking (2PL)

An algorithm for achieving serializable isolation that works by a transaction acquiring a lock on all data it reads or writes, and holding the lock until the end of the transaction. See “[Two-Phase Locking \(2PL\)](#)” on page 257.

unbounded

Not having any known upper limit or size. The opposite of *bounded*.

A

- aborts (transactions), [222](#), [224](#)
 - in two-phase commit, [356](#)
 - performance of optimistic concurrency control, [266](#)
 - retrying aborted transactions, [231](#)
- abstraction, [21](#), [27](#), [222](#), [266](#), [321](#)
- access path (in network model), [37](#), [60](#)
- accidental complexity, removing, [21](#)
- accountability, [535](#)
- ACID properties (transactions), [90](#), [223](#)
 - atomicity, [223](#), [228](#)
 - consistency, [224](#), [529](#)
 - durability, [226](#)
 - isolation, [225](#), [228](#)
- acknowledgements (messaging), [445](#)
- active/active replication (see multi-leader replication)
- active/passive replication (see leader-based replication)
- ActiveMQ (messaging), [137](#), [444](#)
 - distributed transaction support, [361](#)
- ActiveRecord (object-relational mapper), [30](#), [232](#)
- actor model, [138](#)
 - (see also message-passing)
 - comparison to Pregel model, [425](#)
 - comparison to stream processing, [468](#)
- Advanced Message Queuing Protocol (see AMQP)
- aerospace systems, [6](#), [10](#), [305](#), [372](#)
- aggregation
 - in stream processes, [466](#)
- aggregation pipeline query language, [48](#)
- Agile, [22](#)
 - minimizing irreversibility, [414](#), [497](#)
 - moving faster with confidence, [532](#)
 - Unix philosophy, [394](#)
- agreement, [365](#)
 - (see also consensus)
- Airflow (workflow scheduler), [402](#)
- Ajax, [131](#)
- Akka (actor framework), [139](#)
- algorithms
 - algorithm correctness, [308](#)
 - B-trees, [79-83](#)
 - for distributed systems, [306](#)
 - hash indexes, [72-75](#)
 - mergesort, [76](#), [402](#), [405](#)
 - red-black trees, [78](#)
 - SSTables and LSM-trees, [76-79](#)
- all-to-all replication topologies, [175](#)
- AllegroGraph (database), [50](#)
- ALTER TABLE statement (SQL), [40](#), [111](#)
- Amazon
 - Dynamo (database), [177](#)
- Amazon Web Services (AWS), [8](#)
 - Kinesis Streams (messaging), [448](#)
 - network reliability, [279](#)
 - postmortems, [9](#)
 - RedShift (database), [93](#)
 - S3 (object storage), [398](#)
 - checking data integrity, [530](#)
- amplification
 - of bias, [534](#)
 - of failures, [364](#), [495](#)

- of tail latency, [16](#), [207](#)
- write amplification, [84](#)
- AMQP (Advanced Message Queuing Protocol), [444](#)
 - (see also messaging systems)
 - comparison to log-based messaging, [448](#), [451](#)
 - message ordering, [446](#)
- analytics, [90](#)
 - comparison to transaction processing, [91](#)
 - data warehousing (see data warehousing)
 - parallel query execution in MPP databases, [415](#)
 - predictive (see predictive analytics)
 - relation to batch processing, [411](#)
 - schemas for, [93-95](#)
 - snapshot isolation for queries, [238](#)
 - stream analytics, [466](#)
 - using MapReduce, analysis of user activity events (example), [404](#)
- anti-caching (in-memory databases), [89](#)
- anti-entropy, [178](#)
- Apache ActiveMQ (see ActiveMQ)
- Apache Avro (see Avro)
- Apache Beam (see Beam)
- Apache BookKeeper (see BookKeeper)
- Apache Cassandra (see Cassandra)
- Apache CouchDB (see CouchDB)
- Apache Curator (see Curator)
- Apache Drill (see Drill)
- Apache Flink (see Flink)
- Apache Giraph (see Giraph)
- Apache Hadoop (see Hadoop)
- Apache HAWQ (see HAWQ)
- Apache HBase (see HBase)
- Apache Helix (see Helix)
- Apache Hive (see Hive)
- Apache Impala (see Impala)
- Apache Jena (see Jena)
- Apache Kafka (see Kafka)
- Apache Lucene (see Lucene)
- Apache MADlib (see MADlib)
- Apache Mahout (see Mahout)
- Apache Oozie (see Oozie)
- Apache Parquet (see Parquet)
- Apache Qpid (see Qpid)
- Apache Samza (see Samza)
- Apache Solr (see Solr)
- Apache Spark (see Spark)
- Apache Storm (see Storm)
- Apache Tajo (see Tajo)
- Apache Tez (see Tez)
- Apache Thrift (see Thrift)
- Apache ZooKeeper (see ZooKeeper)
- Apama (stream analytics), [466](#)
- append-only B-trees, [82](#), [242](#)
- append-only files (see logs)
- Application Programming Interfaces (APIs), [5](#), [27](#)
 - for batch processing, [403](#)
 - for change streams, [456](#)
 - for distributed transactions, [361](#)
 - for graph processing, [425](#)
 - for services, [131-136](#)
 - (see also services)
 - evolvability, [136](#)
 - RESTful, [133](#)
 - SOAP, [133](#)
- application state (see state)
- approximate search (see similarity search)
- archival storage, data from databases, [131](#)
- arcs (see edges)
- arithmetic mean, [14](#)
- ASCII text, [119](#), [395](#)
- ASN.1 (schema language), [127](#)
- asynchronous networks, [278](#), [553](#)
 - comparison to synchronous networks, [284](#)
 - formal model, [307](#)
- asynchronous replication, [154](#), [553](#)
 - conflict detection, [172](#)
 - data loss on failover, [157](#)
 - reads from asynchronous follower, [162](#)
- Asynchronous Transfer Mode (ATM), [285](#)
- atomic broadcast (see total order broadcast)
- atomic clocks (caesium clocks), [294](#), [295](#)
 - (see also clocks)
- atomicity (concurrency), [553](#)
 - atomic increment-and-get, [351](#)
 - compare-and-set, [245](#), [327](#)
 - (see also compare-and-set operations)
 - replicated operations, [246](#)
 - write operations, [243](#)
- atomicity (transactions), [223](#), [228](#), [553](#)
 - atomic commit, [353](#)
 - avoiding, [523](#), [528](#)
 - blocking and nonblocking, [359](#)
 - in stream processing, [360](#), [477](#)
 - maintaining derived data, [453](#)

- for multi-object transactions, 229
 - for single-object writes, 230
 - auditability, 528-533
 - designing for, 531
 - self-auditing systems, 530
 - through immutability, 460
 - tools for auditable data systems, 532
 - availability, 8
 - (see also fault tolerance)
 - in CAP theorem, 337
 - in service level agreements (SLAs), 15
 - Avro (data format), 122-127
 - code generation, 127
 - dynamically generated schemas, 126
 - object container files, 125, 131, 414
 - reader determining writer's schema, 125
 - schema evolution, 123
 - use in Hadoop, 414
 - awk (Unix tool), 391
 - AWS (see Amazon Web Services)
 - Azure (see Microsoft)
- ## B
- B-trees (indexes), 79-83
 - append-only/copy-on-write variants, 82, 242
 - branching factor, 81
 - comparison to LSM-trees, 83-85
 - crash recovery, 82
 - growing by splitting a page, 81
 - optimizations, 82
 - similarity to dynamic partitioning, 212
 - backpressure, 441, 553
 - in TCP, 282
 - backups
 - database snapshot for replication, 156
 - integrity of, 530
 - snapshot isolation for, 238
 - use for ETL processes, 405
 - backward compatibility, 112
 - BASE, contrast to ACID, 223
 - bash shell (Unix), 70, 395, 503
 - batch processing, 28, 389-431, 553
 - combining with stream processing
 - lambda architecture, 497
 - unifying technologies, 498
 - comparison to MPP databases, 414-418
 - comparison to stream processing, 464
 - comparison to Unix, 413-414
 - dataflow engines, 421-423
 - fault tolerance, 406, 414, 422, 442
 - for data integration, 494-498
 - graphs and iterative processing, 424-426
 - high-level APIs and languages, 403, 426-429
 - log-based messaging and, 451
 - maintaining derived state, 495
 - MapReduce and distributed filesystems, 397-413
 - (see also MapReduce)
 - measuring performance, 13, 390
 - outputs, 411-413
 - key-value stores, 412
 - search indexes, 411
 - using Unix tools (example), 391-394
 - Bayou (database), 522
 - Beam (dataflow library), 498
 - bias, 534
 - big ball of mud, 20
 - Bigtable data model, 41, 99
 - binary data encodings, 115-128
 - Avro, 122-127
 - MessagePack, 116-117
 - Thrift and Protocol Buffers, 117-121
 - binary encoding
 - based on schemas, 127
 - by network drivers, 128
 - binary strings, lack of support in JSON and XML, 114
 - BinaryProtocol encoding (Thrift), 118
 - Bitcask (storage engine), 72
 - crash recovery, 74
 - Bitcoin (cryptocurrency), 532
 - Byzantine fault tolerance, 305
 - concurrency bugs in exchanges, 233
 - bitmap indexes, 97
 - blockchains, 532
 - Byzantine fault tolerance, 305
 - blocking atomic commit, 359
 - Bloom (programming language), 504
 - Bloom filter (algorithm), 79, 466
 - BookKeeper (replicated log), 372
 - Bottled Water (change data capture), 455
 - bounded datasets, 430, 439, 553
 - (see also batch processing)
 - bounded delays, 553
 - in networks, 285
 - process pauses, 298
 - broadcast hash joins, 409

- brokerless messaging, 442
- Brubeck (metrics aggregator), 442
- BTM (transaction coordinator), 356
- bulk synchronous parallel (BSP) model, 425
- bursty network traffic patterns, 285
- business data processing, 28, 90, 390
- byte sequence, encoding data in, 112
- Byzantine faults, 304-306, 307, 553
 - Byzantine fault-tolerant systems, 305, 532
 - Byzantine Generals Problem, 304
 - consensus algorithms and, 366

C

- caches, 89, 553
 - and materialized views, 101
 - as derived data, 386, 499-504
 - database as cache of transaction log, 460
 - in CPUs, 99, 338, 428
 - invalidation and maintenance, 452, 467
 - linearizability, 324
- CAP theorem, 336-338, 554
- Cascading (batch processing), 419, 427
 - hash joins, 409
 - workflows, 403
- cascading failures, 9, 214, 281
- Cascalog (batch processing), 60
- Cassandra (database)
 - column-family data model, 41, 99
 - compaction strategy, 79
 - compound primary key, 204
 - gossip protocol, 216
 - hash partitioning, 203-205
 - last-write-wins conflict resolution, 186, 292
 - leaderless replication, 177
 - linearizability, lack of, 335
 - log-structured storage, 78
 - multi-datacenter support, 184
 - partitioning scheme, 213
 - secondary indexes, 207
 - sloppy quorums, 184
- cat (Unix tool), 391
- causal context, 191
 - (see also causal dependencies)
- causal dependencies, 186-191
 - capturing, 191, 342, 494, 514
 - by total ordering, 493
 - causal ordering, 339
 - in transactions, 262
 - sending message to friends (example), 494
- causality, 554
 - causal ordering, 339-343
 - linearizability and, 342
 - total order consistent with, 344, 345
 - consistency with, 344-347
 - consistent snapshots, 340
 - happens-before relationship, 186
 - in serializable transactions, 262-265
 - mismatch with clocks, 292
 - ordering events to capture, 493
 - violations of, 165, 176, 292, 340
 - with synchronized clocks, 294
- CEP (see complex event processing)
- certificate transparency, 532
- chain replication, 155
 - linearizable reads, 351
- change data capture, 160, 454
 - API support for change streams, 456
 - comparison to event sourcing, 457
 - implementing, 454
 - initial snapshot, 455
 - log compaction, 456
- changelogs, 460
 - change data capture, 454
 - for operator state, 479
 - generating with triggers, 455
 - in stream joins, 474
 - log compaction, 456
 - maintaining derived state, 452
- Chaos Monkey, 7, 280
- checkpointing
 - in batch processors, 422, 426
 - in high-performance computing, 275
 - in stream processors, 477, 523
- chronicle data model, 458
- circuit-switched networks, 284
- circular buffers, 450
- circular replication topologies, 175
- clickstream data, analysis of, 404
- clients
 - calling services, 131
 - pushing state changes to, 512
 - request routing, 214
 - stateful and offline-capable, 170, 511
- clocks, 287-299
 - atomic (caesium) clocks, 294, 295
 - confidence interval, 293-295
 - for global snapshots, 294
 - logical (see logical clocks)

- skew, 291-294, 334
- slewing, 289
- synchronization and accuracy, 289-291
- synchronization using GPS, 287, 290, 294, 295
- time-of-day versus monotonic clocks, 288
- timestamping events, 471
- cloud computing, 146, 275
 - need for service discovery, 372
 - network glitches, 279
 - shared resources, 284
 - single-machine reliability, 8
- Cloudera Impala (see Impala)
- clustered indexes, 86
- CODASYL model, 36
 - (see also network model)
- code generation
 - with Avro, 127
 - with Thrift and Protocol Buffers, 118
 - with WSDL, 133
- collaborative editing
 - multi-leader replication and, 170
- column families (Bigtable), 41, 99
- column-oriented storage, 95-101
 - column compression, 97
 - distinction between column families and, 99
 - in batch processors, 428
 - Parquet, 96, 131, 414
 - sort order in, 99-100
 - vectorized processing, 99, 428
 - writing to, 101
- comma-separated values (see CSV)
- command query responsibility segregation (CQRS), 462
- commands (event sourcing), 459
- commits (transactions), 222
 - atomic commit, 354-355
 - (see also atomicity; transactions)
 - read committed isolation, 234
 - three-phase commit (3PC), 359
 - two-phase commit (2PC), 355-359
- commutative operations, 246
- compaction
 - of changelogs, 456
 - (see also log compaction)
 - for stream operator state, 479
 - of log-structured storage, 73
 - issues with, 84
 - size-tiered and leveled approaches, 79
- CompactProtocol encoding (Thrift), 119
- compare-and-set operations, 245, 327
 - implementing locks, 370
 - implementing uniqueness constraints, 331
 - implementing with total order broadcast, 350
 - relation to consensus, 335, 350, 352, 374
 - relation to transactions, 230
- compatibility, 112, 128
 - calling services, 136
 - properties of encoding formats, 139
 - using databases, 129-131
 - using message-passing, 138
- compensating transactions, 355, 461, 526
- complex event processing (CEP), 465
- complexity
 - distilling in theoretical models, 310
 - hiding using abstraction, 27
 - of software systems, managing, 20
- composing data systems (see unbundling data-bases)
- compute-intensive applications, 3, 275
- concatenated indexes, 87
 - in Cassandra, 204
- Concord (stream processor), 466
- concurrency
 - actor programming model, 138, 468
 - (see also message-passing)
 - bugs from weak transaction isolation, 233
 - conflict resolution, 171, 174
 - detecting concurrent writes, 184-191
 - dual writes, problems with, 453
 - happens-before relationship, 186
 - in replicated systems, 161-191, 324-338
 - lost updates, 243
 - multi-version concurrency control (MVCC), 239
 - optimistic concurrency control, 261
 - ordering of operations, 326, 341
 - reducing, through event logs, 351, 462, 507
 - time and relativity, 187
 - transaction isolation, 225
 - write skew (transaction isolation), 246-251
- conflict-free replicated datatypes (CRDTs), 174
- conflicts
 - conflict detection, 172
 - causal dependencies, 186, 342
 - in consensus algorithms, 368
 - in leaderless replication, 184

- in log-based systems, 351, 521
 - in nonlinearizable systems, 343
 - in serializable snapshot isolation (SSI), 264
 - in two-phase commit, 357, 364
 - conflict resolution
 - automatic conflict resolution, 174
 - by aborting transactions, 261
 - by apologizing, 527
 - convergence, 172-174
 - in leaderless systems, 190
 - last write wins (LWW), 186, 292
 - using atomic operations, 246
 - using custom logic, 173
 - determining what is a conflict, 174, 522
 - in multi-leader replication, 171-175
 - avoiding conflicts, 172
 - lost updates, 242-246
 - materializing, 251
 - relation to operation ordering, 339
 - write skew (transaction isolation), 246-251
- congestion (networks)
- avoidance, 282
 - limiting accuracy of clocks, 293
 - queueing delays, 282
- consensus, 321, 364-375, 554
- algorithms, 366-368
 - preventing split brain, 367
 - safety and liveness properties, 365
 - using linearizable operations, 351
 - cost of, 369
 - distributed transactions, 352-375
 - in practice, 360-364
 - two-phase commit, 354-359
 - XA transactions, 361-364
 - impossibility of, 353
 - membership and coordination services, 370-373
 - relation to compare-and-set, 335, 350, 352, 374
 - relation to replication, 155, 349
 - relation to uniqueness constraints, 521
- consistency, 224, 524
- across different databases, 157, 452, 462, 492
 - causal, 339-348, 493
 - consistent prefix reads, 165-167
 - consistent snapshots, 156, 237-242, 294, 455, 500
 - (see also snapshots)
- crash recovery, 82
- enforcing constraints (see constraints)
- eventual, 162, 322
 - (see also eventual consistency)
- in ACID transactions, 224, 529
- in CAP theorem, 337
- linearizability, 324-338
- meanings of, 224
- monotonic reads, 164-165
- of secondary indexes, 231, 241, 354, 491, 500
- ordering guarantees, 339-352
- read-after-write, 162-164
- sequential, 351
- strong (see linearizability)
- timeliness and integrity, 524
- using quorums, 181, 334
- consistent hashing, 204
- consistent prefix reads, 165
- constraints (databases), 225, 248
 - asynchronously checked, 526
 - coordination avoidance, 527
 - ensuring idempotence, 519
 - in log-based systems, 521-524
 - across multiple partitions, 522
 - in two-phase commit, 355, 357
 - relation to consensus, 374, 521
 - relation to event ordering, 347
 - requiring linearizability, 330
- Consul (service discovery), 372
- consumers (message streams), 137, 440
 - backpressure, 441
 - consumer offsets in logs, 449
 - failures, 445, 449
 - fan-out, 11, 445, 448
 - load balancing, 444, 448
 - not keeping up with producers, 441, 450, 502
- context switches, 14, 297
- convergence (conflict resolution), 172-174, 322
- coordination
 - avoidance, 527
 - cross-datacenter, 168, 493
 - cross-partition ordering, 256, 294, 348, 523
 - services, 330, 370-373
- coordinator (in 2PC), 356
 - failure, 358
 - in XA transactions, 361-364
 - recovery, 363

- copy-on-write (B-trees), 82, 242
 - CORBA (Common Object Request Broker Architecture), 134
 - correctness, 6
 - auditability, 528-533
 - Byzantine fault tolerance, 305, 532
 - dealing with partial failures, 274
 - in log-based systems, 521-524
 - of algorithm within system model, 308
 - of compensating transactions, 355
 - of consensus, 368
 - of derived data, 497, 531
 - of immutable data, 461
 - of personal data, 535, 540
 - of time, 176, 289-295
 - of transactions, 225, 515, 529
 - timeliness and integrity, 524-528
 - corruption of data
 - detecting, 519, 530-533
 - due to pathological memory access, 529
 - due to radiation, 305
 - due to split brain, 158, 302
 - due to weak transaction isolation, 233
 - formalization in consensus, 366
 - integrity as absence of, 524
 - network packets, 306
 - on disks, 227
 - preventing using write-ahead logs, 82
 - recovering from, 414, 460
 - Couchbase (database)
 - durability, 89
 - hash partitioning, 203-204, 211
 - rebalancing, 213
 - request routing, 216
 - CouchDB (database)
 - B-tree storage, 242
 - change feed, 456
 - document data model, 31
 - join support, 34
 - MapReduce support, 46, 400
 - replication, 170, 173
 - covering indexes, 86
 - CPUs
 - cache coherence and memory barriers, 338
 - caching and pipelining, 99, 428
 - increasing parallelism, 43
 - CRDTs (see conflict-free replicated datatypes)
 - CREATE INDEX statement (SQL), 85, 500
 - credit rating agencies, 535
 - Crunch (batch processing), 419, 427
 - hash joins, 409
 - sharded joins, 408
 - workflows, 403
 - cryptography
 - defense against attackers, 306
 - end-to-end encryption and authentication, 519, 543
 - proving integrity of data, 532
 - CSS (Cascading Style Sheets), 44
 - CSV (comma-separated values), 70, 114, 396
 - Curator (ZooKeeper recipes), 330, 371
 - curl (Unix tool), 135, 397
 - cursor stability, 243
 - Cypher (query language), 52
 - comparison to SPARQL, 59
- ## D
- data corruption (see corruption of data)
 - data cubes, 102
 - data formats (see encoding)
 - data integration, 490-498, 543
 - batch and stream processing, 494-498
 - lambda architecture, 497
 - maintaining derived state, 495
 - reprocessing data, 496
 - unifying, 498
 - by unbundling databases, 499-515
 - comparison to federated databases, 501
 - combining tools by deriving data, 490-494
 - derived data versus distributed transactions, 492
 - limits of total ordering, 493
 - ordering events to capture causality, 493
 - reasoning about dataflows, 491
 - need for, 385
 - data lakes, 415
 - data locality (see locality)
 - data models, 27-64
 - graph-like models, 49-63
 - Datalog language, 60-63
 - property graphs, 50
 - RDF and triple-stores, 55-59
 - query languages, 42-48
 - relational model versus document model, 28-42
 - data protection regulations, 542
 - data systems, 3
 - about, 4

- concerns when designing, 5
- future of, 489-544
 - correctness, constraints, and integrity, 515-533
 - data integration, 490-498
 - unbundling databases, 499-515
- heterogeneous, keeping in sync, 452
- maintainability, 18-22
- possible faults in, 221
- reliability, 6-10
 - hardware faults, 7
 - human errors, 9
 - importance of, 10
 - software errors, 8
- scalability, 10-18
- unreliable clocks, 287-299
- data warehousing, 91-95, 554
 - comparison to data lakes, 415
 - ETL (extract-transform-load), 92, 416, 452
 - keeping data systems in sync, 452
 - schema design, 93
 - slowly changing dimension (SCD), 476
- data-intensive applications, 3
- database triggers (see triggers)
- database-internal distributed transactions, 360, 364, 477
- databases
 - archival storage, 131
 - comparison of message brokers to, 443
 - dataflow through, 129
 - end-to-end argument for, 519-520
 - checking integrity, 531
 - inside-out, 504
 - (see also unbundling databases)
 - output from batch workflows, 412
 - relation to event streams, 451-464
 - (see also changelogs)
 - API support for change streams, 456, 506
 - change data capture, 454-457
 - event sourcing, 457-459
 - keeping systems in sync, 452-453
 - philosophy of immutable events, 459-464
 - unbundling, 499-515
 - composing data storage technologies, 499-504
 - designing applications around dataflow, 504-509
 - observing derived state, 509-515
- datacenters
 - geographically distributed, 145, 164, 278, 493
 - multi-tenancy and shared resources, 284
 - network architecture, 276
 - network faults, 279
 - replication across multiple, 169
 - leaderless replication, 184
 - multi-leader replication, 168, 335
- dataflow, 128-139, 504-509
 - correctness of dataflow systems, 525
 - differential, 504
 - message-passing, 136-139
 - reasoning about, 491
 - through databases, 129
 - through services, 131-136
- dataflow engines, 421-423
 - comparison to stream processing, 464
 - directed acyclic graphs (DAG), 424
 - partitioning, approach to, 429
 - support for declarative queries, 427
- Datalog (query language), 60-63
- datatypes
 - binary strings in XML and JSON, 114
 - conflict-free, 174
 - in Avro encodings, 122
 - in Thrift and Protocol Buffers, 121
 - numbers in XML and JSON, 114
- Datomic (database)
 - B-tree storage, 242
 - data model, 50, 57
 - Datalog query language, 60
 - excision (deleting data), 463
 - languages for transactions, 255
 - serial execution of transactions, 253
- deadlocks
 - detection, in two-phase commit (2PC), 364
 - in two-phase locking (2PL), 258
- Debezium (change data capture), 455
- declarative languages, 42, 554
 - Bloom, 504
 - CSS and XSL, 44
 - Cypher, 52
 - Datalog, 60
 - for batch processing, 427
 - recursive SQL queries, 53
 - relational algebra and SQL, 42
 - SPARQL, 59

- delays
 - bounded network delays, 285
 - bounded process pauses, 298
 - unbounded network delays, 282
 - unbounded process pauses, 296
- deleting data, 463
- denormalization (data representation), 34, 554
 - costs, 39
 - in derived data systems, 386
 - materialized views, 101
 - updating derived data, 228, 231, 490
 - versus normalization, 462
- derived data, 386, 439, 554
 - from change data capture, 454
 - in event sourcing, 458-458
 - maintaining derived state through logs, 452-457, 459-463
 - observing, by subscribing to streams, 512
 - outputs of batch and stream processing, 495
 - through application code, 505
 - versus distributed transactions, 492
- deterministic operations, 255, 274, 554
 - accidental nondeterminism, 423
 - and fault tolerance, 423, 426
 - and idempotence, 478, 492
 - computing derived data, 495, 526, 531
 - in state machine replication, 349, 452, 458
 - joins, 476
- DevOps, 394
- differential dataflow, 504
- dimension tables, 94
- dimensional modeling (see star schemas)
- directed acyclic graphs (DAGs), 424
- dirty reads (transaction isolation), 234
- dirty writes (transaction isolation), 235
- discrimination, 534
- disks (see hard disks)
- distributed actor frameworks, 138
- distributed filesystems, 398-399
 - decoupling from query engines, 417
 - indiscriminately dumping data into, 415
 - use by MapReduce, 402
- distributed systems, 273-312, 554
 - Byzantine faults, 304-306
 - cloud versus supercomputing, 275
 - detecting network faults, 280
 - faults and partial failures, 274-277
 - formalization of consensus, 365
 - impossibility results, 338, 353
 - issues with failover, 157
 - limitations of distributed transactions, 363
 - multi-datacenter, 169, 335
 - network problems, 277-286
 - quorums, relying on, 301
 - reasons for using, 145, 151
 - synchronized clocks, relying on, 291-295
 - system models, 306-310
 - use of clocks and time, 287
- distributed transactions (see transactions)
- Django (web framework), 232
- DNS (Domain Name System), 216, 372
- Docker (container manager), 506
- document data model, 30-42
 - comparison to relational model, 38-42
 - document references, 38, 403
 - document-oriented databases, 31
 - many-to-many relationships and joins, 36
 - multi-object transactions, need for, 231
 - versus relational model
 - convergence of models, 41
 - data locality, 41
- document-partitioned indexes, 206, 217, 411
- domain-driven design (DDD), 457
- DRBD (Distributed Replicated Block Device), 153
- drift (clocks), 289
- Drill (query engine), 93
- Druid (database), 461
- Dryad (dataflow engine), 421
- dual writes, problems with, 452, 507
- duplicates, suppression of, 517
 - (see also idempotence)
 - using a unique ID, 518, 522
- durability (transactions), 226, 554
- duration (time), 287
 - measurement with monotonic clocks, 288
- dynamic partitioning, 212
- dynamically typed languages
 - analogy to schema-on-read, 40
 - code generation and, 127
- Dynamo-style databases (see leaderless replication)

E

- edges (in graphs), 49, 403
 - property graph model, 50
- edit distance (full-text search), 88
- effectively-once semantics, 476, 516

- (see also exactly-once semantics)
- preservation of integrity, 525
- elastic systems, 17
- Elasticsearch (search server)
 - document-partitioned indexes, 207
 - partition rebalancing, 211
 - percolator (stream search), 467
 - usage example, 4
 - use of Lucene, 79
- ElephantDB (database), 413
- Elm (programming language), 504, 512
- encodings (data formats), 111-128
 - Avro, 122-127
 - binary variants of JSON and XML, 115
 - compatibility, 112
 - calling services, 136
 - using databases, 129-131
 - using message-passing, 138
 - defined, 113
 - JSON, XML, and CSV, 114
 - language-specific formats, 113
 - merits of schemas, 127
 - representations of data, 112
 - Thrift and Protocol Buffers, 117-121
- end-to-end argument, 277, 519-520
 - checking integrity, 531
 - publish/subscribe streams, 512
- enrichment (stream), 473
- Enterprise JavaBeans (EJB), 134
- entities (see vertices)
- epoch (consensus algorithms), 368
- epoch (Unix timestamps), 288
- equi-joins, 403
- erasure coding (error correction), 398
- Erlang OTP (actor framework), 139
- error handling
 - for network faults, 280
 - in transactions, 231
- error-correcting codes, 277, 398
- Esper (CEP engine), 466
- etcd (coordination service), 370-373
 - linearizable operations, 333
 - locks and leader election, 330
 - quorum reads, 351
 - service discovery, 372
 - use of Raft algorithm, 349, 353
- Ethereum (blockchain), 532
- Ethernet (networks), 276, 278, 285
 - packet checksums, 306, 519
- Etherpad (collaborative editor), 170
- ethics, 533-543
 - code of ethics and professional practice, 533
 - legislation and self-regulation, 542
 - predictive analytics, 533-536
 - amplifying bias, 534
 - feedback loops, 536
 - privacy and tracking, 536-543
 - consent and freedom of choice, 538
 - data as assets and power, 540
 - meaning of privacy, 539
 - surveillance, 537
 - respect, dignity, and agency, 543, 544
 - unintended consequences, 533, 536
- ETL (extract-transform-load), 92, 405, 452, 554
 - use of Hadoop for, 416
- event sourcing, 457-459
 - commands and events, 459
 - comparison to change data capture, 457
 - comparison to lambda architecture, 497
 - deriving current state from event log, 458
 - immutability and auditability, 459, 531
 - large, reliable data systems, 519, 526
- Event Store (database), 458
- event streams (see streams)
- events, 440
 - deciding on total order of, 493
 - deriving views from event log, 461
 - difference to commands, 459
 - event time versus processing time, 469, 477, 498
 - immutable, advantages of, 460, 531
 - ordering to capture causality, 493
 - reads as, 513
 - stragglers, 470, 498
 - timestamp of, in stream processing, 471
- EventSource (browser API), 512
- eventual consistency, 152, 162, 308, 322
 - (see also conflicts)
 - and perpetual inconsistency, 525
- evolvability, 21, 111
 - calling services, 136
 - graph-structured data, 52
 - of databases, 40, 129-131, 461, 497
 - of message-passing, 138
 - reprocessing data, 496, 498
 - schema evolution in Avro, 123
 - schema evolution in Thrift and Protocol Buffers, 120