Ch. 1 Reliable, Scalable and Maintainable Applications

Thinking About Data Systems

Reliability

- general expectations
  - the application performs the function that the user expected
  - it can tolerate the user making mistakes, or using the software in unexpected ways
  - its performance is good enough for the required use case, under expected load and data volume
  - the system prevents any unauthorized access and abuse
- generally handles faults (fault tolerance or resiliance)

Hardware faults

Software errors

Human errors

How important is reliability

Scalability

- ability to cope with increased load

Describing load

- described with numbers called load parameters
- choice for measurement depends on system
- fan-out -> number of requests to other services that need to be made to service one incoming request

Describing performance

- once load parameters have been defined
  - how is system performance affected when a load parameter is increased but system resources stay the same?

    - how to keep system performance constant when increasing a load parameter (i.e. which resources need to be scaled to maintain performance in the face of load)?
- throughput -> number of records/transactions/requests that can be processed per second (or time it takes to run a job on a dataset of a certain size)
- response time -> time from client request to receipt of response by client
- latency and response time
    - not equivalent
    - response time is time observed by client
    - latency is time a request is waiting to be serviced
- measure response time as a distribution
- check for outliers
- consider average/arithmetic mean but prefer percentiles
- service level objectives (SLOs)
- service level agreements (SLAs)

Approaches for coping with load

- scale up -> vertical scaling, i.e. moving to a more powerful machine
- scale out -> horizontal scaling, i.e. distributing load across multiple machines
- elastic -> ability scale according to load
- previous wisdom was to keep database on a single node
- currently conceivable that distributed systems are or will become the default
- distributed architectures are generally very specific to the system

Maintainability

Operability: making life easy for operations

Simplicity: managing complexity

Evolvability: making change easy

Summary

- application has requirements to be useful
- functional requirements
    - what application should do
    - data storage
    - data retrieval
    - data searching, processing, analysis
- non-functional requirements, e.g. properties
    - security

    - reliability
        - correct functionality even during faults
    - compliance
    - scalability
        - handling load gracefully
        - requires quantitative description of load
        - generally requires the ability to add processing capacity to remain reliable under load
    - compatibility
    - maintainability
        - improving quality of life for engineering and operations team working on an application
        - includes
            - understandable abstractions
            - complexity management
        - good operability
            - insight into system health
            - effective system health management

Ch. 2 Data Models and Query Languages

- data models are incredibly important because they affect how developers think about the problem
- layers of abstraction built on each other

Relational Model vs. Document Model

- SQL is the best known relational model
- relations = table in SQL
- unordered(?) collection of tuples = rows in SQL
- started as a theoretical model and became most popular model for data storage systems
- business data processing
    - transaction processing
    - batch processing
- competing approaches
    - network model
    - hierarchical model
    - object databases

The birth of NoSQL

- name started as twitter hashtag

- defining characteristics
  - need for greater scalability than relational databases can easily achieve (large datasets or very high write throughput)
  - preference towards free and open source software over commercial database products
  - specialized query operations that are not well supported by the relational model
  - frustration with the restrictiveness of relational schemas, and a desire for a more dynamic and expressive data model
- polyglot persistence -> combination of data stores with different data models into a connected system

The object-relational mismatch

- impedance mismatch between object-oriented programming and data model storage in relational databases
- ORMs adapt the two but don't achieve complete fluid translation
- some solutions
  - use foreign keys to tables within SQL
  - store multi-valued data in fields in SQL (e.g. array, etc)
    - Oracle
    - DB2
    - SQL Server
    - PostgeSQL
  - store structured data in fields in SQL (JSON, XML, etc)
- many document-oriented databases
  - MongoDB
  - RethinkDB
  - CouchDB
  - Espresso
- JSON is good at translating across the impedance mismatch
  - still has other issues
  - lack of a schema is good at times
  - better locality (one query to retrieve all info for a record)

Many-to-one and many-to-many relationships

- normalization -> removing duplication across tables
  - literature distinguishes many different normal forms in relational model but simple view is removing duplication
- normalization often requires many-to-one relationships
- document model is not good at many-to-one (weak join support)
- relational model uses joins
- may need to emulate joins in application code

- data model evolution in document model often leads to a need for joins

Are document databases repeating history?

- most popular DB in 70's was IBM IMS
    - used a hierarchical model (similar to JSON) as tree of records with nesting
- worked well for one-to-many relationships
- had issues with many-to-many relationships -> use dernomalization or manually resolve references?
- solutions
    - relational model
    - network model

The network model

- CODASYL
- generalizes tree-based hierarchical model to graph-like (in sense of data structure) model to facilitate many-to-many relationships
- used something like pointers to make connections (access path)
- very difficult to make changes to an application's data model

The relational model

- relation (table) as collection of tuples (rows)
- query optimizer handles access path optimizations behind the scenes
- query optimizers are complex but only need to be built once and general purpose solution wins in the long run

Comparison to document databases

- like hierarchical model in that they store nested records (one-to-many)
- many-to-one and many-to-many relationships are not fundamentally different from relational model
    - foreign key in relational model
    - document reference in document model
- use a join or follow-up query to resolve document references

Relational vs. document databases today

- arguments for document data model
    - closer to application data model for some applications
    - schema flexibility
    - performance due to locality

- arguments for relational model
  - better support for joins
  - better support for many-to-one relationships
  - better support for many-to-many relationships

Which data model leads to simpler application code?

- tree structure data application -> use document model
- many-to-many -> use relational
- shredding -> relational technique of splitting a document-like structure into multiple tables
- document model has poor support for references to nested items
- document model is awkward for highly interconnected data

Schema flexibility in the document model

- no enforced schema in document model means arbitrary keys and values can be added to a document and clients have no guarantee of format of data
- sometimes called schemaless (misleading)
- schema-on-read (vs. schema-on-write of relational DBs)
- similar to dynamic (runtime/duck) typing vs static (compile-time) typing
  - each has relative merits
- migration and schema changes in relations DBs can be slow and cause downtime
- schema-on-read is advantageous for heterogeneous data models
  - many different types of objects
  - objects are close but vary in ways that cannot be modeled in relational db
  - structure of data is determined externally and may change over time
- schemas are useful for consistent structures

Data locality for queries

- stored as single string (JSON, XML, BSON, etc)
- storage locality -> all data is close together on disk unlike data split across multiple tables
- only applies if need parts of document at same time
- updates require complete rewrite of document
- recommended to keep documents small
- can be achieved in other models like relational
  - Spanner achieves storage locality by interleaving rows within a parent table's rows
  - Oracle uses multi-table index cluster tables
  - Bigtable model (Cassandra, HBase) uses column-families

Convergence of document and relational databases

- many relational DBs support XML storage

- support for JSON
  - PostgreSQL -> v9.3
  - MySQL -> v5.7
  - DB2 -> v10.5
- RethinkDB supports relational-like joins
- MongoDB drivers automatically resolve database references
- hybrid of relational and document models is a good route for databases to take in the future

Query Languages for Data

- SQL -> declarative
- IMS & CODASYL -> imperative
- many benefits to declarative approach
  - more concise and easier to reason about/work with (at times)
  - lends itself to parallel execution

Declarative Queries on the Web

- CSS and XSL use a declarative approach
- achieving the same modifications in JavaScript by modifying the DOM is often ugly and error prone

MapReduce Querying

- popularized by Google and a limited form is supported by MongoDB and CouchDB
- neither declarative nor imperative (many related traits to functional languages)
- filters, maps across records, and modifies results, sometimes into single result as in map
- must be pure functions
- lower level than SQL in general
- disadvantage
  - often requires writing two coordinate operations rather than one query
- MongoDB added support for aggregation pipeline (a declarative query language) which mixes declarative SQL-like syntax with a JSON like syntactical structure

Graph-Like Data Models

- many-to-many
- vertices (nodes/entities)
- edges (relationship/arcs)
- examples
  - social graphs
  - graph of the web

    - road or rail networks
- many different but related ways of structuring data in graphs
- property graph model
    - Neo4j
    - Titan
    - InfiniteGraph
- triple-store model
    - Datomic
    - AllegroGraph
    - others
- declarative query languages
    - Cypher
    - SPARQL
    - Datalog
- imperative/graph processing frameworks
    - Gremlin
    - Pregel

Property Graphs

- vertex
    - UID
    - outgoing edge set
    - incoming edge set
    - collection of properties (key-value pairs)
- edge
    - UID
    - start vertex (tail vertex)
    - end vertex (head vertex)
    - label describing edge's vertex relationship
    - collection of properties (key-value pairs)
- can think of it as consisting of two relational tables
    - one for vetices
    - one for edges
- important aspects
    - no restrictions on how vertices can be connected, any to any
    - can traverse the graph efficiently given its vertex
    - can store many different kinds of information in a graph by labeling edges and storing key-value pairs
- good for evolvability

The Cypher Query Language

- query language for Neo4j
- see examples here

Graph Queries in SQL

- both are possible and different data will result in different data models in each
- one equivalent query may be easy in one (graph/SQL) and difficult in other

Triple-Stores and SPARQL

- all information is stored in the form of very simple three-part statements
    - (subject, predicate, object)
- subject = vertex
- object
    - a value in a primitive datatype
        - key-value = predicate-object
    - another graph vertex
        - predicate = edge in graph
        - subject is tail vertex
        - object is head vertex

The semantic web

- idea was that websites should publish information as machine-readable data in conjunction with the published human-readable data
- Resource Description Framework (RDF)
- would allow for data from different websites to be combined in a web of data
- never caught on but has many interesting ideas

The RDF data model

- Turtle language
- can also be expressed in XML

The SPARQL query language

- triple-stores using RDF data model

- can be powerful tool for internal use within an application

Graph Databases Compared to the Network Model

- differ from CODASYL's network model in many ways
    - CODASYL used a schema which specified nesting abilities
    - no restrictions on interconnections in graph model DBs
    - CODASYL requires traversal through a path
    - graph DBs can access vertex through ID
    - CODASYL - children of record were ordered set (maintained ordering)
    - no implicit ordering in graph DB
    - CODASYL - imperative and difficult queries broken by schema changes
    - graph DBs high-level declarative query languages as well as some support for imperative queries

The Foundation: Datalog

- older language (80's)
- provided foundations for many later query languages
- used in
    - Datomic
    - Cascalog (used with Hadoop)
- similar to triple-stores
    - predicate(subject, object)
- builds up complex queries from one small piece at a time
- can build rules for predicate-based matching
- requires a different kind of thinking
    - less convenient for one-off queries
    - powerful for dealing with complex data

Summary

- many different models, huge subject
- historically
    - large tree
        - not good for many-to-many
    - relational model
        - some applications don't fit relational model
    - NoSQL/non-relational data stores
        - document databases - data stored in document and relationships between documents are minimal
        - graph databases - any data is potentially related to any other data

- relational, document, and graph are each good for a particular problem domain and awkward when applied to one of the other's respective problem domains
- no one-size-fits-all solution
- document and graph databases rarely enforce a schema
- query languages/frameworks
    - SQL
    - MapReduce
    - MongoDB aggregation pipeline
    - Cypher
    - SPARQL
    - Datalog
- CSS and XSL/XPath
- many other data models
    - sequence similarity searches require specialized genome databases like GenBank
    - massively large datasets often require custom solutions to reduce costs
    - full-text search -> information retrieval is its own problem domain

Ch. 3 Storage and Retrieval

- storage and retrieval are most basic characteristic operations of a database
- generally two main workloads
    - optimized for transactions
    - optimized for analytics
- examine two families of storage engines
    - log-structured
    - page-oriented

Data Structures that Power Your Database

- simple key-value store database

```
#!/bin/bash

db_set () {
    echo "$1,$2" >> database
}
db_get () {
    grep "^$1," database | sed -e "s/^$1,//" | tail -n 1
}
```

- store has great performance
- lookup has terrible performance because it requires a full table scan

- for efficient find must use an index
    - store metadata in additional structure as a guidepost for finding data
    - adds overheads to writes because index structure must be updated
- many databases require the developer to choose the index manually to optimize benefit for current application

Hash Indexes

- key-value or dictionary type
- indexing with hash map
    - in memory where every key is mapped to a byte offset within the data file
    - writes update hash map also
    - reads query hash map first
- Riak/Bitcask uses this approach
- useful for use case involving many writes
- since only appending to file at this point, run the risk of running out of disk space
- solution
    - break log into size segments
    - perform compaction on these segments
- compaction -> throw away duplicate keys in the log and only keep most recent
- generally also requires merging log segments
- since segments are never modified after written, merge is written to a new segment
- after merging is complete, update the log to read from the new segment and remove the old segment
- each segment then has its own in-memory indexing hash table
- to read a value
    - check most recent index for key
    - continue back through previous segments
- the described process also requires
    - file format -> text vs binary, if text, what structure (JSON, CSV, XML, etc)
    - record deletion -> append tombstone (special record to indicate deletion) to log file, delete on merge
    - crash recovery -> how to recover indexes from shutdown? can use snapshots to disk or some form of journaling
    - partially written records -> can use checksums to discover and remove partially written records on crash recovery
    - concurrency control -> commonly allow only one write and immutable nature of data files facilitates concurrent reads
- append only log is good for several reasons
    - sequential writes are generally much faster than random writes, especially on spinning disks but also generally on SSDs
    - concurrency and crash recovery is simpler

    - merging old segments avoids fragmentation
- limitations of hash table index
    - must fit in memory (can write to disk but causes random access I/O, costly to grow, and hash collisions are difficult to deal with)
    - range queries are not efficient

SSTables and LSM-trees

- modify by requiring that writes to logs are sorted by key, i.e. use a sorted string table or SSTable
- also require each key only appears once within a merged file (handled by merges)
- advantages of sstables over logs with hash indexes
    - merging is simple and efficient (similar to mergesort)
    - no longer require an index of all of the keys in memory but instead can keep an index of some of the keys to provide an offset range and then scan
    - can compress segments to a block size when writing to save disk space and reduce I/O bandwidth of reads

Constructing and maintaining SSTables

- easier to maintain in memory using red-black or AVL trees
    - on write, add to memtable or in-memory balanced tree structure
    - when size of memtable exceeds a limit, write to SSTable file and empty memtable
    - on read, check memtable, then most recent on-disk segment, then next most recent, etc.
    - occasionally run merging and compaction in background to combine segment files and discard overwritten or deleted values
- disadvantage:
    - crash will lose current memtable
    - to avoid, can maintain separate log of all writes on disk, recover memtable on crash recovery, and remove log on memtable write to disk

Making an LSM-tree out of SSTables

- previous algorithm use essentially used in LevelDB and RocksDB
- similar algorithms used in Cassandra and HBase
- SSTables and memtables come from Google's Bigtable paper
- originally described as Log-Structured Merge-Tree (LSM-tree) which builds on log structured file systems
- Lucene (used by Elasticsearch and Solr) stores term dictionary in SSTable like files
    - key = word or term

    - value = posting list of IDs of all the documents that contain the word

Performance optimizations

- LSM-trees are slow for nonexistent key lookups
    - solve by using a Bloom filter
- to determine order and timing of merge and compaction
    - size-tiered
        - newer and smaller SSTables are successively merged into older and larger SSTables
    - leveled
        - key range is split up into smaller SSTables and older data is moved into separate levels (allows compaction to proceed incrementally and uses less disk space)
- leveled
    - LevelDB
    - RocksDB
    - Cassandra
- size-tiered
    - HBase
    - Cassandra
- simple and effective, supports datasets much larger than available memory, supports high write throughput due to sequential disk writes

B-trees

- most widely used indexing structure
- standard in almost all relational and many non-relational DBs
- breaks DB down into fixed-size blocks or pages, usually 4kB, which corresponds to underlying hardware (as opposed to the variable sized segments of log-structured segments)
- each node or page contains k keys and k+1 references to child nodes
- leaf nodes contain values
- traverse to leaf nodes through key range divisions
- on addition, split full pages into two half-full pages and update the parent
- B-trees maintain balance so n keys means a height of O(logn)

Making B-trees reliable

- writes write to pages on disk as opposed to LSM-trees which never overwrite in place
- overwrite on disk
    - magnetic
        - move head of spinning platter and overwrite appropriate sector
        - SSD must erase and rewrite large block of chip at a time

- some operations require writing to many pages which can be dangerous due to potential for failure during write
- common to also maintain a write-ahead log (WAL or redo log)
  - append only log written to before B-tree modification
- must also carefully consider concurrent access
  - commonly done with latches (lightweight locks)
- log-structured approaches are simpler with respect to concurrency

B-tree optimizations

- copy-on-write instead of WAL
- save space by abbreviating key (B+tree)
- pages can be positioned anywhere on disk, but more performance will be increased if leaf pages appear in sequential order on disk (difficult to maintain as size increases)
- add additional pointers to siblings for scan
- B-tree variants (fractal trees) use log-structured ideas to reduce disk seeks

Comparing B-trees and LSM-trees

- generally
  - B-trees faster for reads
  - LSM-trees faster for writes
- benchmarks can be inconclusive

Access patterns for writes

- B-trees must write twice, once to WAL and once to page
- overhead for writing entire page at a time
- LSM-trees write many times for a record on each merge/compaction (write amplification)
- LSM-trees are generally able to sustain higher write throughput
- LSM-trees can be compressed better (smaller on disk)
- B-trees can lead to fragmentation
- LSM-trees have lower storage overhead due to compaction and lower fragmentation
- SSDs firmware converts random writes to sequential writes so impact is less noticeable

Downsides of LSM-trees

- compaction can affect read/write performance, sometimes making the performance of B-trees more predictable
- need to share disk bandwidth for initial writes and compaction writes
- compaction may not be able to keep up with rate of incoming writes
- need explicit monitoring to handle this
- B-trees do not duplicate keys

- best to test empirically for use case to determine between B-tree and LSM-tree

Other indexing structures

- only discussed key-value, like primary key index in a relational model
  - points to
    - row in relational
    - document in document db
    - vertex in graph
- common to have secondary index
- main difference is that keys are not unique for secondary index
  - either make value list of matching row IDs
  - or make each key unique by appending row ID to it

Storing values within the index

- can store actual target object (row, document, vertex) or a reference to an object stored elsewhere
  - heap file -> place to store target object elsewhere (maybe append only)
- can be efficient for updates
- can store indexed row directly within an index (clustered index)
  - can have secondary index refer to primary clustered index
- options
  - clustered index -> store all row data within the index
  - nonclustered index -> store only references to the data within the index
  - covering index (index with included columns) -> stores some of table's columns within the index
- clustering and covering indexes can speed up reads but require additional storage

Multi-column indexes

- may need a key to map to multiple values
- concatenated index -> appends one column to another to retrieve all combinations of certain types (like a join but inherent in storage)
- multi-dimensional indexes -> more general method for querying several columns at once
- PostGIS uses R-trees to implement geospatial indexes
- many common applications for multidimensional indexes

Full-text search and fuzzy indexes

- searching for similar keys (fuzzy querying)
- text search may include
  - synonyms
  - ignore grammatical variations

    - location within a document
    - edit distance
    - other linguistic analysis features
- Lucene uses a structure similar to a trie (FSA over characters of keys)
    - can be transformed into a Levenshtein automaton
- others use document classification and machine learning
    - see information retrieval textbook for more info

Keeping everything in memory

- in-memory databases
    - Memcached (entirely in-memory)
    - VoltDB (relational)
    - MemSQL (relational)
    - Oracle TimesTen (relational)
    - RAMCloud (durable key-value store)
    - Redis (weak durability using async writes)
    - Couchbase (weak durability using async writes)
- some are not concerned with shutdown and loss
- some still write to disk as append only log for backup and serve reads entirely from memory
- performance advantage is from avoiding overhead of having to store data in a structure that can be easily written to disk
- can provide data models (e.g. priority queues and sets, etc) that are difficult to implement with disk-based indexes (Redis)
- anti-caching -> uses LRU eviction to write to disk and reloads on access
- anti-caching can be used to support datasets larger than memory
- anti-caching is similar to OS paging but is more efficient as DBs can work at a more granular level

Transaction Processing or Analytics

- commercial transaction -> previously corresponded to write to database for businesses
- term transaction persisted
    - do not need ACID properties
        - atomicity
        - consistency
        - isolation
        - durability
    - transaction processing vs. batch processing
        - transactions correspond to reads and writes
- online transaction processing (OLTP) - interactive user input (i.e. CRUD in real time)

- data analytics use cases
  - scan huge data set and aggregate statistics
- online analytic processing (OLAP) - workload supporting data analytics over transactions
- OLTP
  - read pattern - small number of record queries fetched by key
  - write pattern - random access, low latency writes from user input
  - main use - end user/customer via (web) application
  - data represents - latest state of data at point in time of access
  - dataset size - Gb to Tb
- OLAP
  - read pattern - aggregate over large set of records
  - write pattern - bulk import (ETL) or event stream
  - main use - internal anlayst for decision support
  - data represents - history of events that occurred over time
  - dataset size - Tb to Pb
- in 80's and 90's trend towards analytics used separate database in what is called a data warehouse

Data warehousing

- business may have many different transaction processing systems that must be highly-available
  - analysts should generally not operate on these
- data warehouse -> separate database(s) used to store relevant data from other systems for use by analysts
- ETL (extract-transform-load)
  - extracted from other systems using periodic dump or streaming
  - cleaned up for analysis
  - loaded into data warehouse DBs
- a large benefit is gained by using transformation to facilitate relevant queries

The divergence between OLTP databases and data warehouses

- data analysis tools
  - drill-down
  - slicing and dicing
- Microsoft SQL Server and SAP HANA support transaction processing and data warehousing within the same product
- commercial vendors
  - Teradata
  - Vertica
  - SAP HANA

    - ParAccel
- open source SQL-on-Hadoop solutions to compete
    - Apache Hive,
    - AMPLab's Shark
    - Cloudera Impala
    - Hortonworks Stinger
    - Facebook Presto
    - Apache Tajo
    - Apache Drill
- some use ideas from Google's Dremel

Stars and snowflakes: schemas for analytics

- many data warehousing techniques are formulaic
    - star schema aka dimensional modeling
    - centered on fact table or the table that reflects an event or a transaction
    - some columns in fact tables are foreign key references to other tables (dimension tables)
    - dimension tables describe events (who, what, when, where, why, how)
- star schema -> fact table is in the middle and references dimension tables as point of the star
- snowflake schema
    - dimensions are further broken down into subdimensions
    - snowflake schemas are more normalized than star schemas but star schemas are easier for analysts to work with
- fact tables may be several hundred columns wide

Column-oriented storage

- large datasets (trillions of rows and petabytes of data) are difficult to store and query
- common queries only access a few columns at a time (out of many hundred)
- this requires loading all of the rows from disk to memory even with multiple keys
- column-oriented storage -> rather than storing all of the values from a row together, store all of the values from a column together which can save work
- can be relational or non-relation
- Parquet -> columnar storage based on Google's Dremel

Column compression

- data is more easily compressed due to similar data type and reduces disk throughput demands
- number of distinct values in a column is generally also small (due to repetition)

- NOTE:
    - column-oriented storage and column families
        - column families -> comes from Bigtable but used by HBase and Cassandra
        - store all columns from a row together along with a row key
        - do not use compression
        - as a result, mostly row-oriented

Memory bandwidth and pipelined execution

- must consider memory cache performance
- avoiding branch mispredictions and bubbles in CPU instruction processing pipeline
- utilizing vectorized instructions such as SIMD
- column compression allows for more values to fit in cache
- can be iterated and processed in tight loop

Sort order in column storage

- can choose another sort order and use as an indexing mechanism
- doesn't make sense to sort independently because loose the original relationship to entry's row
- can batch sort by rows at a time
- sort order can help with compression because highly repetitive columns as primary sort column can reduce compressed size

Several different sort orders

- C-Store and adopted in Vertica
- can use replicas to store in a separate sort order

Writing to column-oriented storage

- column-oriented storage is good for the OLAP use case
- not good for many writes
- LSM-trees are a good solution
    - store writes in memory
    - merge at merge point with column-oriented files on disk

Aggregation: Data cubes and materialized views

- many data warehouse use row-oriented storage
    - columnar storage is gaining popularity
- materialized aggregates -> storage of many aggregate queries (count, sum, average, min, max, etc)
- materialized view -> a table-like object whose contents are the result of some query
- like a standard (virtual) view

- on data update, must be repopulated due to dernomalized view of data
- don't necessarily make sense in OLTP DBs, make sense in OLAP DBs
- data cube (OLAP cube) -> gird of aggregates grouped by different dimensions
    - each cell contains the aggregate data for a particular combination of the dimensions from a fact table and are repeatedly summarized along each of the dimensions
- makes certain queries very fast
- reduces flexibility from querying raw data
- most data warehouses use raw data for most queries and reserve data cubes for heavily used, performance heavy queries

Summary

- two broad categories for storage and retrieval
    - OLTP
    - OLAP
- log-structured
    - only append to files
    - delete obsolete files
    - use SSTables and LSM-trees
    - Bitcask
    - LevelDB
    - Cassandra
    - HBase
    - Lucene
- update-in place
    - B-trees
    - treats disk as fixed-size pages which can be overwritten
    - most relational DBs and many non-relational DBs
- data warehouse architecture
- column oriented storage

Ch. 4 Encoding and Evolution

- evolvability allows an application to adapt to change
- change will happen
- data schemas may change
- server side code can be upgraded with rolling upgrades
- client-side code will be upgraded by users at their will
- leads to a need for both forward and backward compatibility
- forward compatibility can be tricky

Formats for Encoding Data

- programs generally work with at least 2 different representations of data
    i. in-memory
        - objects
        - structs
        - lists
        - arrays
        - hash tables
        - trees
        - etc
    ii. encoded data for writing to a file or sending over a network which is generally very different from in-memory data
- translation is
    - encoding-decoding
    - serialization-deserialization
    - marshalling-unmarshalling

Language-Specific Formats

- built-in support and third-party libraries
- Java -> java.io.Serializable
- Ruby -> Marshal
- Python -> Pickle
- issues
    - encoding is tied to the programming language which blocks integration with other systems
    - decoding leads to restoring arbitrary object types
        - presents various security issues
    - versioning data is often not built in which prevents forward and backward compatibility
    - they are generally not very efficient
- often best to avoid built-in serialization in enterprise applications

JSON, XML, and Binary Variants

- JSON, XML, and CSV are most common standardized data interchange formats
- issues
    - ambiguity around encoding of numbers (number vs string of digits, int vs float)
    - issues with large numbers
    - JSON and XML support Unicode but not binary strings
    - optional schema support for XML and JSON but it is complicated to learn

    - many JSON based tools do not bother using schemas
    - CSV does not have a schema so application must define meta-info about rows and columns and changes must be handled manually

Binary encoding

- JSON is less verbose than XML, but both still use a lot of space compared to binary formats
- profusion of binary formats for JSON
    - MessagePack
    - BSON
    - BJSON
    - UBJSON
    - BISON
    - Smile
- XML
    - WBXML
    - Fast Infoset
- describes MessagePack binary encoding format here

Thrift and Protocol Buffers (protobuf)

- binary encoding libraries
    - Protocol Buffers was originally developed at Google
    - Thrift was originally developed at Facebook
    - both were made open source in 2007–08
- Thrift defines schema using Thrift IDL
- provide code generation tool which can then be called to encode or decode records based on schema
- Thrift has two binary protocols
    - BinaryProtocol -> format described here
    - CompactProtocol -> format also described here
- CompactProtocol is semantically equivalent but encodes with smaller size
- Protocol Buffers encode the data similarly but in a generally smaller size than CompactProtocol

Field tags and schema evolution

- each format allows for adding and removing schema components
- need to make new fields optional and add a new tag number to maintain both forward and backward compatibility

- can only remove optional fields and can never use same tag number again

Datatypes and schema evolution

- check docs for details
- Protocol Buffers does not have array data type, uses repeated marker
- Thrift has dedicated list datatype

Avro

- binary encoding format started as subproject of Hadoop
- two schema languages
  - Avro IDL -> human readable
  - JSON based -> machine readable
- Avro encoding size is smaller
- reader and writer need to use same schema -> see below

The writer's schema and the reader's schema

- reader schema and writer schema need to be compatible
- spec defines how resolution for changes works

Schema evolution rules

- forward compatibility means that you can have a new version of the schema as writer and an old version of the schema as reader
- backward compatibility means that you can have a new version of the schema as reader and an old version as writer
- uses unions to represent null
- does not support optional and required
- some datatypes can be converted between each other

But what is the writer's schema?

- reader's awareness of writer's schema depends on the context in which Avro is being used
  - large files with lots of records -> i.e. Hadoop, lots of records, same schema, include writer's schema once at the beginning of the file
  - database with individual records -> reader extracts record once, gets version number, finds writer's schema with version number and decodes records

    - sending records over network -> negotiate schema on connection (Avro RPC)
- useful to keep database of schema versions

Dynamically generated schemas

- advantage -> no tag numbers stored in schema
    - friendlier to dynamically generated schemas
    - can easily encode Avro schema from database schema and dump database to Avro records
    - easy to redo on database schema change
    - with Thrift or Protocol Buffers the schema changes and tagging would need to be done manually

Code generation and dynamically typed languages

- code generation for language of choice is especially useful in statically typed languages
- code generation is often frowned upon in dynamic languages like JavaScript, Python, or Ruby
- Avro provides optional code generation
- Pig integrates well with Avro -> can write derived datasets to output files in Avro format without much effort

The Merits of Schemas

- concepts behind Avro, Thrift, and Protocol Buffers are not new - related to ASN.1
- advantages of binary encodings based on schemas
    - much more compact than even binary JSON
    - schema documents much of intent and has to be up to date
    - schema database documents forward and backward compatibility
    - code generation is useful and allows compile time checking
- schema evolution generally provides the same kind of flexibility as schemaless or schema-on-read JSON DBs provide
- provide better guarantees about structure and versioning of data and better tooling

Modes of Dataflow

- sharing data when there is no shared memory requires encoding data as a sequence of bytes
- forward and backward compatibility are essential to evolvability
- dataflow mechanisms
    - databases
    - service calls
    - asynchronous message passing

Dataflow Through Databases

- process for write and read (encode and decode) that may be same process
- need to ensure future changes can still encode/decode current data
- often many processes accessing database and common for processes to run different code versions (written by newer and read by older or vice versa)
- Avro, etc solve issues caused by addition and removal of a field in relation to versioning

Different values written at different times

- data often outlives code
- migrations are possible but expensive
- can use nulls in some DBs to handle potentially missing values
- Espresso uses Avro for storage so it can use the schema evolution rules

Archival storage

- snapshotting for data warehousing is a good use case for Avro's object container files
- also good to consider Parquet or some other column-oriented storage for this use case

Dataflow Through Services: REST and RPC

- client-server architecture is most common (model of the web and web browsers)
- many native apps also use this
- HTTP is used for API access and as transport protocol
- server can also be client to another service
- microservices architecture (aka service oriented architecture or SOA) -> approach often used to decompose a large application into smaller services by area of functionality, such that one service makes a request to another when it requires some functionality or data from that other service
- services are similar to DBs but only expose application-specific API for IO
- key design goal is to make application more adaptable to change

Web services

- use of HTTP = web service
  - client app on user device making HTTP requests for resources
  - middleware within same organization
  - inter-organization API access (OAuth, etc)
- REST and SOAP (SOAP is almost dead)
- REST -> design philosophy outlining data formats, URLs for access to resources, cache control, authentication, and content type negotiation
- gaining much popularity over SOAP and associated with microservices

- SOAP -> XML-based protocol for API requests, commonly associated with web service frameworks that may be monoliths
- uses WSDL to describe web service and enable code generation
- requires tool support and often IDE usage
- RESTful APIs are simpler
  - involve less code generation and automated tooling
- definition format such as OpenAPI, also known as Swagger, can be used to describe RESTful APIs and produce documentation

The problems with remote procedure calls (RPCs)

- web services are latest in API network requests lineage
  - Enterprise JavaBeans
  - Java's Remote Method Invocation (RMI)
  - Distributed Component Object Model (DCOM - Microsoft)
  - Common Object Request Broker Architecture (CORBA)
- all based on RPC
- RPC is fundamentally flawed
  - success or failure of networked function calls is not clear and prone to network issues
  - network function call can throw, return, or timeout and timeouts may be inconclusive
  - retries can lead to violations idempotency
  - network is slow
  - passing references and pointers and other non-primitive objects can be difficult or impossible
  - client and service may be in different languages which can cause any number of impedance mismatches
- best to view RPC as its own thing and accept it as network-based

Current directions for RPC

- Thrift and Avro have RPC support
- gRPC for Protocol Buffers and streams
- Finagle and Rest.li
- some are more explicit about outlining difference between remote request and local function calls (using futures/promises and streams)
- some provide service discovery
- RPC with binary is often faster
- REST advantages
  - experimentation and debugging
  - supported by all mainstream languages and platforms
  - vast ecosystem of tools
- REST = public APIs
- RPC = service requests within the same organization

Data encoding and evolution for RPC

- RPC clients and services must be able to be changed and deployed independently
- reasonable to assume servers are update before clients
    - backward compatibility on requests
    - forward compatibility on responses
- compatibility properties of each listed here
- RPC may be made across organizational boundaries so control over upgrades may be impossible
- no standardization on API versioning
    - REST should use version number in URL or HTTP Accept header
    - can use an API key

Message-Passing Dataflow

- asynchronous message passing systems are somewhere between RPC and databases
    - client request is delivered with low-latency
    - like databases, delivered via intermediary which stores message temporarily
        - message broker
        - message queue
        - message-oriented middleware
- advantages of message broker over RPC
    - improve system reliability by acting as buffer
    - redeliver/retry to prevent loss of messages
    - does not require sender to know IP/port of recipient
    - allows broadcast
    - provides decoupling
- however, message passing is usually one-way

Message brokers

- previously dominated by enterprise software
    - TIBCO
    - IBM WebSphere
    - webMethods
- recent open source
    - RabbitMQ
    - ActiveMQ
    - HornetMQ
    - NATS
    - Kafka
    - Google Cloud Pub/Sub

    - Amazon Kinesis
    - Simple Queueing Service
    - (ZeroMQ allows broker, no broker, distributed broker, directory service, distributed directory service)
- delivery semantics varies by implementation
    - send to queue or topic
    - delivered to consumer or subscriber (potentially many to on topic)
- topic is one-way
- consumers can publish to topics consumed by orginal sender allowing two-way dataflow
- usually do not enforce a data model

Distributed actor frameworks

- actor model -> programming model for concurrency in a single process where actors maintain state and communicate with other actors (asynchronously) for the sake of abstracting over error prone manual threaded programming
- distributed actor frameworks -> actor model across multiple with communication transparently performed over network
- location transparency is better in actor model than RPC (local and remote communication generally looks the same)
- puts message broker and actor into single framework
- still requires forward and backward compatibility
- popular
    - Akka -> works with Java built-in serialization but can add Protocol Buffers, etc for versioning
    - Orleans -> custom data encoding but no support for rolling upgrades (plug-ins for other serializers)
    - Erlang OTP -> difficult to change schemas
    - (CAF)

Summary

- explored methods of encoding/decoding
- need for rolling upgrades and evolvability
- data encoding formats
    - programming-language specific
    - JSON, XML, CSV
    - Thrift, Protocol Buffers, Avro
- modes of dataflow
    - DBs with process(es) for read and write
    - RPC and REST APIs (client-server)
    - async message passing with message brokers or actors

PART II Distributed Data

- reasons for use of multiple machines
    - scalability
        - spread load for reads or writes across multiple machines if it is greater than what can be handled by a single machine
    - fault tolerance/high availability
        - use multiple machines for redundancy to handle faults or failures by allowing another to take over
    - latency
        - latency imposed by bad networks or geographic distance can be mediated by having a machine that is part of the application in a location that is closer

Scaling to Higher Load

- if processing power/load increase is all that is needed, can just use a more powerful computer (vertical scaling or scaling up)
- shared-memory architecture
    - combining multiple CPUs, RAM chips, and disks together under one operating system with a fast interconnect to treat as a single machine
- has a linear cost but due to bottlenecks may not be able to handle load in a linear proportion to scaling
- offers limited fault tolerance
- shared-disk architecture
    - CPU and RAM form a single machine but storage is on an array of disks that is shared between machines
    - increases fault tolerance and storage but is generally not scalable beyond that

Shared-Nothing Architectures

- aka
    - horizontal scaling
    - scaling out
- each machine is a node with independent hardware and coordination is performed at a software level
- can choose machines by best price/performance ratio because no special hardware is needed
- can use VMs for cloud deployments

- a lot to be aware of in shared-nothing architecture for application developers

Replication vs. Partitioning

- replication
    - storage of a copy of the same data on several different nodes for redundancy and performance
- partitioning
    - splitting data in a data store into smaller subsets of the data (partitions) so they can be assigned to different nodes (sharding)

Ch. 5 Replication

- keeping multiple copies of same data
    - geographic location (reduce latency)
    - fault tolerance and availability
    - increase throughput by increasing number of machines that can serve queries (scale out)
- assumes each dataset can be stored on a single machine (and discuss partitioning later)
- replication is easy for immutable data sets
- difficulty lies in handling changes
- three popular algorithms
    - single-leader
    - multi-leader
    - leaderless
- considerations
    - synchronous vs. asynchronous replication
    - handling of failed replicas
- studied since 70's
- many misunderstandings for application developers

Leaders and Followers

- replica -> node with a copy of data
- main question -> how to ensure all data updates get propagated to all replicas?
- leader-based replication (aka active/passive or master-slave)
    - replica designated as leader (aka master or primary)
    - write requests sent through leader
    - leader sends updates to followers (aka read replicas, slaves, secondaries, hot standbys) as a replication log or change stream
    - follower udpates by running all writes in order of replication log
    - reads can be sent to leader or any follower used by

    - relational
        - PostgresQL (since 9.0)
        - MySQL
        - Oracle Data Guard
        - SQL Server's AlwaysOn Availability Groups
    - non-relational
        - MongoDB
        - RethinkDB
        - Espresso
    - distributed message brokers
        - Kafka
        - RabbitMQ
    - networked file systems/replicate block devices
        - DRBD

Synchronous vs. asynchronous replication

- important and often configurable
- sync
    - leader waits for a successful ack from first follower before reporting success
    - advantage
        - follower is always guaranteed to have up-to-date copy
    - disadvantage
        - write cannot be processed in case of replica or network failure and system will become blocked
    - only makes sense for one of the followers to be synchronous
- async
    - leader fires and forgets change stream
    - advantage
        - leader can process writes even in case of replica or network failure between leader and replica
    - disadvantage
        - if leader fails, writes are not recoverable so not durable

Setting up new followers

- increase number of replicas or replace failed nodes
- cannot simply copy
- could lock before copy but this remove HA requirement
- conceptual process to avoid downtime
    - snapshot leader's DB at some pint in time (without locking)
    - copy snapshot to follower

    - follower comes online and requests replication log from point of snapshot (log sequence number or binlog coordinates)
    - after processing backlog, replica continues as any other follower would

Handling node outages

- reboots and crashes occur
- achieving HA with leader-based replication

Follower failure: catch-up recovery

- follower logs data changes on disk
- to recover, follower supplies leader with last log sequence point and leader provides an update

Leader failure: failover

- promote follower to leader position
- manual or automatic
- process
    i. determine that leader has failed -> can use a heartbeat/timeout
    ii. choose a new leader -> leader election algorithm (consensus problem) or chosen by previously determined controller node (generally chooses most up-to-date replica)
    iii. reconfigure system to new leader -> set clients and followers to send requests to new leader, handle old leader coming back online by assigning as a follower and updating data
- common failover issues
    - for async replication, new leader may not be up-to-date (may discard old leaders writes but there are other solutions)
    - discarding writes can be dangerous
    - can result in two nodes believing they are leaders (split brain) and may not be able to resolve conflicts between both leaders accepting writes
    - proper timeout before electing a new leader -> failover too often or not often enough can degrade performance in situations where the system is likely already struggling to handle performance
- all point to fundamental distributed systems problems
    - replica consistency
    - durability
    - availability
    - latency

Implementation of replication logs

- presents common leader-based replication methods

Statement-based replication

- leader logs each insert, update, or delete request and forwards to followers, where followers parse each as a client request
- disadvantages:
    - non-deterministic functions (like reading time) will not be the same
    - counters necessitate updates in exact same order which may be difficult to maintain for multiple concurrently executing transactions
    - side-effects may not result in the same state on each replica unless forced to result in deterministic side-effects
- there are workarounds, but other replication methods are preferred
- used in MySQL prior to 5.1
- used by VoltDB but forces transactions to be deterministic

Write-ahead log (WAL) shipping

- exist as part of log-structured storage engine
- written to before page writes in B-tree based storage engine
- append only sequence of bytes indicating all writes to DB
- used in
    - PostgreSQL
    - Oracle
    - others

Logical log replication

- use different log formats for replication than those used by storage engine
- logical vs physical log
- describes writes at granularity of a row
    - insert -> all values for row
    - delete -> ID for row to remove
    - update -> ID for row and values of all columns or only the updated values
- easier to maintain backwards compatibility, allowing for different software versions
- easier for other software to parse (for data warehousing, etc)

Trigger-based replication

- may need more flexibility to manage what data is replicated
- makes sense to move replication to application layer
- trigger (stored procedure) -> register custom application code to DB so it is triggered on some DB event (i.e. a write) (can be used to transfer data to OLAP DB)
    - Databus for Oracle
    - Bucardo for PostgreSQL

- adds overhead to replication

Problems With Replication Lag

- for mostly read workloads (common pattern on web)
  - create many followers and distribute reads across these followers
- results in read-scaling or load balancing
- only realistically works with async replication
  - synchronous replication would hang with a node failure or network outage and large systems are almost guaranteed to have some sort of failure
- reads from async followers may have outdated info (inconsistencies)
- eventual consistency -> consistency property in async replication systems where at a single instant in time, replicas may not be consistent with each other, but stopping writes for some (unspecified) amount of time will cause the system to return to a consistent state
- not only NoSQL
  - followers in an asynchronously replicated relational database have the same characteristics

Reading your own writes

- with async replication, if user writes (through leader) and reads immediately after from follower, may not see their own write
- read-after-write consistency (read-your-writes consistency) -> guarantees a user will always see their own writes in an async replication system
- options for implementing in a leader-based async replication system
  - when reading a value the user may have modified, read from leader
    - requires knowing what the user may have modified without querying
    - can resort to always reading user's own data (profile, etc) from leader
  - for applications where a user can modify most things, can instead track last update (to system overall) and default to reading from leader if last update occurred within a preset time duration
  - can also track last write by user and check last update to follower's replica, resorting to leader if follower is not up-to-date (requires distributed timestamping system using logical timestamps)
- geography (i.e. routing through datacenters) adds complexity
- additional issues to consider for cross-device access by a user
  - need to centralize last write timestamp information for a user

    - connections from different devices may not be routed through same datacenter an thus interaction with the same follower

Monotonic reads

- leader-based with async replication can result in user seeing things going backwards in time when making several reads from different replicas with varying degrees of lag
- monotonic reads -> guarantee that several reads in sequence will not result in later reads reporting older data than previous reads (weaker guarantee than strong consistency and stronger guarantee than eventual consistency)
- one method - always read from same replica

Consistent prefix reads

- in cases where there is a causal relationship between data written to a database over time (i.e. chat exchange with sequential responses) requires another guarantee
- consistent prefix reads -> if a sequence of writes happens in a certain (causal) order then anyone reading those writes will see them appear in the same order
- common in partitioned/sharded databases
- in general, requires a distributed transaction with a guarantee such as snapshot isolation

Solutions for replication lag

- need to consider increasingly long lags
- applications can provide stronger guarantees than the underlying database
- many distributed databases have abandoned transactions in favor of eventual consistency
- claim is that the need to rely on eventual consistency within a scalable system is inevitable
- this is not true (see Ch 7)

Multi-leader replication

- leader-based replication results in a single point of failure
- multi-leader configuration (aka master-master replication or active/active)
    - allow more than one node to accept writes
    - each node that accepts writes must forward data to all other nodes
    - each leader simultaneously acts as a follower to other leaders

Use cases for multi-leader replication

- generally not useful in a single datacenter

Multi-datacenter operation

- can have a leader in each datacenter

- within each datacenter, regular leader-follower replication is used
- comparison
    - performance
        - inter-datacenter delay for inter-leader replication is hidden from user so perceived performance may be better
    - datacenter outage tolerance
        - single leader -> failover of single leader in outage can cause a performance hit
        - mult-leader -> each datacenter can continue operating independently of others
    - tolerance of network problems
        - inter-datacenter traffic is over the public net and is slower and less reliable
        - single leader is sensitive to this
        - multi-leader with async replication handles much better
    - can be implemented with external tools
        - Tungsten Replicator for MySQL
        - BDR for PostgreSQL
        - GoldenGate for Oracle
- often considered dangerous and should be avoided if possible

Clients with offline operation

- many desktop/mobile apps like calendar need offline operation
- local database acts as leader
- may have very long replication lag
- essentially the same as multi-leader replication
- tools designed for this
    - CouchDB
    - (PouchDB)

Collaborative editing

- Etherpad
- Google docs
- share many of the same distributed database problems
- for complete conflict freedom guarantee a lock must be used
    - equivalent to single-leader replication with transactions on the leader
- can make unit of change very small to avoid locks
    - brings challenges of multi-leader replication and conflict resolution

Handling write conflicts

- multi-leader replication allows write conflicts

Synchronous vs. asynchronous conflict detection

- single leader -> second write blocks and waits or aborts their own transaction and retires
- multi-leader -> both writes succeed and conflict is detected at some later point
  - can make conflict detection synchronous but loses main advantage of multi-leader replication (allowing each leader to accept writes independently)

Conflict avoidance

- easiest to avoid (all writes for an application go through a single leader)
- if there is a need to change the leader for any reason (failure etc) must deal with the possibility of concurrent writes on different leaders

Converging towards a consistent state

- single-leader -> writes a committed in a sequential order
- multi-leader -> no defined ordering to writes, so database can end up in inconsistent state
- method for convergent conflict resolution
  - ID writes (timestamp/UUID) and prefer highest ID (last write wins or LWW)
    - prone to data loss
  - give each replica an ID and prefer writes at a higher numbered replica
    - prone to data loss
  - merge written data in some way
  - record conflict in a structure that preserves all information and add handling to resolve conflict at some later time

Custom conflic resolution logic

- many multi-leader replication tools expose conflict resolution customization hooks
  - on write
    - handler cannot prompt user, is generally run in background, and must run quickly
  - on read
    - application stores conflicts and may prompt user to resolve on read or resolve automatically and write back to DB
    - CouchDB

Automatic conflict resolution

- conflict-free replicated data types (CRDTs)
  - family of data structures (sets, maps, ordered lists, counters, etc) which can be modified concurrently and automatically resolve conflicts (see Riak 2.0)
- mergeable persistent data structures
  - track history (like git) and use a three-way merge function (CRDTs use two-way merge) (see Merkl trees)

- operational transformation
    - algorithm behind Google docs
    - manages concurrent editing of an ordered list of items, e.g. characters composing a text document
- all are relatively young algorithms but likely they will mature and grow as they are used in more applications

What is a conflict?

- subtle conflicts
    - single usage conflicts (no overlapping usage of a resource) occurring at same time

Multi-leader replication topologies

- replication topology -> describes path along which writes are propagated from one node to another
- examples
    - circular
    - star
    - all-to-all
- writes may pass through several nodes before reaching all replicas
- nodes are given UUID
- circular and star topologies can be interrupted by node failure
- all-to-all topologies demonstrate better fault tolerance
- all-to-all topologies can have messages overtake others due to differences in network latency between nodes
    - results in problem of causality similar to consistent prefix reads
- ordering of events properly can be solved with a technique called version vectors
- read system docs and be aware of issues that may arise with multi-leader replication

Leaderless replication

- leader determines order of writes and followers replicate in same order

- leaderless allows any replica node to accept writes

    - Dynamo
    - Riak
    - Cassandra
    - Voldemort

- Dynamo-style databases

- some allow clients to send writes to multiple replicas at once

- some use a coordinator (controller node) to forward writes
    - coordinator does not enforce a particular ordering
- NOTE: Dynamo is different from DynamoDB -> Dynamo is internal and DynamoDB is external and has a different architecture

Writing to the database when a node is down

- leaderless configurations do not have a concept of failover
- client sends write to all replicas in parallel
- uses quorum writes and quorum reads to manage leaderless reads and writes to handle the possibility of a failure
- requires read repairs after outages

Read repair and anti-entropy

- two mechanisms used in Dynamo-style datastores to restore nodes to a consistent state after an outage
    - read repair -> client can detect stale responses from reads from several replicas and writes newer value back to replica
        - works well for frequently read values
    - anti-entropy -> background process that constantly looks for differences in data between replicas (not ordered like log process, may be significant delay)
- some systems use only one

Quorums for reading and writing

- generally
    - for n replicas, need w writes and r reads from replicas
    - as long as $w + r > n$ can expect to get an up-to-date value when reading
        - at least one of the r nodes reading from must be up-to-date
    - reads and writes that obey these r and w values are called quorum reads and writes
    - view r and w as the minimum number of votes required for the read or write to be valid
- typically configurable in Dynamo style databases and various settings can result in different read vs. write throughput
- $w + r > n$ facilitates fault tolerance because
    - if $w < n$ writes can still be processed on node failure
    - if $r < n$ reads can still be processed on node failure
    - various settings of odd numbers of n and 1/2 or greater than 1/2 of r/w allows 1 to 2 nodes to go down and still allow reads/writes
    - normally both reads and writes are sent to all n replicas in parallel quorum is only used ack back
- error on fewer than w or r nodes available

Limitations of quorum consistency

- edge cases exist with above quorum formula where stale values may be returned for reads
    - sloppy quorum -> writes may end up on different nodes than reads so value overlap is not guaranteed
    - two concurrent writes must be merged because LWW can result in dropping writes due to clock skew
    - write and read concurrently, write may only go to some replicas and returned read value is unclear
    - write succeed on some and fail on others below quorum (i.e failed write) the result might not get rolled back and read may not return write value
    - if a node that is part of quorum fails before update across cluster the quorum condition may fail
    - more edge cases even in healthy functioning
- for w + r <= n, stale values are more likely but increases availability and lowers latency
- Dynamo-style databases are optimized for use cases that tolerate eventual consistency
- stronger guarantees require transactions or consensus

Monitoring staleness

- need to monitor for health
- easier to monitor in leader-based systems
- in leaderless
    - no fixed order for writes
    - if only read repair is used, no limit to how old a value might be
- makes monitoring more difficult
- very important to be able to quantify eventuality in eventually consistent
- research in area

Sloppy quorums and hinted handoff

- quorums can tolerate network delay and node failure (to a certain extent)
- how to handle failed quorum (when data has a location to which it is designated)?
    - error
    - write to quorum subset
- sloppy quorum -> still require a quorum as above but the successful responses can be from nodes for which the data is not destined
- on repair, the data must be sent to designated nodes aka hinted handoff
- sloppy quorum is not a quorum but more an assurance of durability
- sloppy quorums are optional in all common Dynamo implementations
    - Riak - enabled by default
    - Cassandra and Voldemort - disabled by default

Multi-datacenter operation

- leaderless replication is well-suited for multi-datacenter replication
- Cassandra and Voldemort use the normal leaderless model
    - clients usually wait for quorum ack within its own datacenter
    - higher latency writes to other datacenters happen asynchronously in background
- Riak performs cross-datacenter replication in a style similar to multi-leader replication

Detecting concurrent writes

- Dynamo-style allows multiple concurrent writes to same key, resulting in conflict
    - also in read-repair and hinted handoff
- most DBs handle the convergence towards eventually consistency poorly

Last write wins (discarding concurrent writes)

- only store recent and overwrite older values in each replica
    - requires unambiguous mechanism for determining most recent
- not correct to discuss concept of most recent in context of concurrency because order is undefined
- instead, force arbitrary order using timestamp and LWW
- achieves eventual consistency at the cost of durability
    - multiple successful concurrent writes will only result in a single write
- if data loss is unacceptable, LWW is not a good choice
- best solution (with Cassandra, etc) is to ensure a key is only written once (thus is immutable) possibly by using a UUID as the key so each write operation receives a unique key

The "happens-before" relationship and concurrency

- to determine concurrency (by intuition)
    - think of causally dependent relationships between writes
        - have an implicit ordering
    - concurrent writes cannot have a causally dependent relationship because neither client knows of the other's action
- three possibilities for events A and B
    - A -> B
    - B -> A
    - A and B are concurrent

Concurrency, time, relativity

- difficult problem due to clocks in distributed systems
- simplify to classifying two operations as concurrent if they are unaware of each other

Tracking happens-before relationships

- see text for description of problem and solution algorithm

Merging concurrently written values

- merging sibling values is similar to conflict resolution in multi-leader replication
- can use LWW but this is often not good enough
- can use set unions for siblings but does is complicated by removal because if only one client removes the other will see a value previously removed
- system must maintain a deletion marker for a key as a tombstone
- CRDTs are applicable here as well

Version vectors

- using a single version number is not acceptable when multiple replicas are accepting writes concurrently
- version vector -> a per replica, per key version number instead
- most interesting is Riak's dotted version vector
- facilitates
  - distinguishing overwrites
  - distinguishing concurrent writes
  - sibling merge
  - read and write from separate replicas
- version vectors are similar to vector clocks
  - version vector -> client server systems
  - vector clock -> peer-to-peer systems

Summary

- purposes of replication
  - HA
  - disconnected operation
  - managing latency
  - scalability
- approaches
  - single-leader
  - multi-leader
  - leaderless
- sync and async replication
- consistency models for handling replication lag
  - read-after-write

    - monotonic reads
    - consistent prefix reads
- concurrency issues

Ch. 6 Partitions

- aka
    - shard
    - region
    - tablet
    - vBucket
- the result of splitting data replicas to handle large datasets or handle high query throughput
- generally each piece of data only belongs to a single partition
- done to facilitate scalability
- shared-nothing architecture -> horizontal scaling or scaling out, each machine or VM is a distinct node, with independent CPU, RAM, disks, etc, and coordination is done at a software level (i.e. through messaging) rather than sharing resources
- not related to network partitions
- pioneered in 80's with Teradata and Tandem NonStop SQL

Partitioning and replication

- commonly combined
- node may store more than one partition (various configurations)

Partitioning of key-value data

- spread query load
- data may be skewed where one node receives more queries than others (hot spot)
- random distribution of keys is simplest resolution but causes a need to query all nodes for data since do not know where the data is

Partitioning by key range

- can determine which node contains a key the range it is responsible is known
- can manually set partitions or automatically
- partition boundaries adapt to data
- used by
    - BigTable
    - HBase
    - RethinkDB
    - MongoDB (before 2.4)

- can keep keys in sorted order within partitions
- key range partitioning can lead to access patterns that cause hot spots
- timestamps are susceptible to problems in this scheme

Partitioning by hash of key

- good hash function uniformly distributes keys across partitions
- MD5
  - Cassandra
  - MongoDB
- Fowler-Noll-Vo
  - Voldemort
- (also Murmur 3 and City Hash)
- can assign range of hashes to a partition
- ranges can be evenly spaced or chosen pseudo-randomly
- lose ability to do efficient range queries
- in MongoDB, sends range queries to all partitions
- not supported in Riak, Couchbase, and Voldemort
- Cassandra compromises
  - table can be declared with a compound primary key consisting of several columns
  - only first part of key is hashed to determine partition
  - other columns are used to concatenate index for sorting the data in SSTables
  - cannot scan range on first part, but if first part is set, can scan range on remainder
  - facilitates one-to-many relationships

Consistent Hashing

- method for evenly distributing load across internet-wide system of caches such as CDN
- chooses random partition boundaries to avoid need for central control or distributed consensus
- use of word consistent here is different from other uses in book
- doesn't work well for databases

Skewed workloads and relieving hot spots

- hashing does not completely avoid hot spots
- one technique is to add random number to beginning or end of hot key to split across many keys

- affects reads because reads then have to do more work

Partitioning and secondary indexes

- very useful for data modeling, common in relational databases, some document DBs, and has been added to Riak
- Solr and Elasticsearch function solely for secondary indexes
- don't map neatly to partitions
- solutions
  - document-based partitioning
  - term-based partitioning

Partitioning secondary indexes by document

- each partitions maintains the secondary index for the data within that partition and is not concerned with secondary indexes in other partitions
- known as local index
- reading a document-partitioned secondary index does not guarantee to return a full set of data based on the secondary index
- scatter/gather -> send secondary index to all partitions and combine returned results
- scatter/gather is a solution but is expensive
- used in
  - MongoDB
  - Riak
  - Cassandra
  - Elasticsearch
  - SolrCloud
  - VoltDB
- suggest to structure partitioning scheme so secondary index queries can be served from a single partition

Partitioning secondary indexes by term

- can construct global index that covers data in all partitions, but it must also be partitioned
- can be partitioned differently
- term-based partitioning
- can make reads more efficient by avoiding scatter/gather
- can add write overhead
- updates to global secondary indexes are often asynchronous
- used in
  - DynamoDB
  - Riak
  - Oracle data warehouse

Rebalancing partitions

- database change
  - query throughput increases, so add more CPUs to handle load
  - dataset size increases, so more disks and RAM for storage
  - machines fail and other machines handle new responsibilities
- moving data for these reasons is called rebalancing
  - updates load to balance load
  - database should continue to handle reads and writes during rebalance
  - rebalance should minimize data transfer

Strategies for rebalancing

How not to do it: hash mod N

- number of nodes changing modifies N so most data needs to be moved making rebalancing very expensive

Fixed number of partitions

- create more partitions than nodes and store many partitions on each node
- allows for stealing partitions on node addition
- only entire partitions are transferred
- can also assign more partitions to more powerful machines to balance load
- used in
  - Riak
  - Cassandra since 1.2
  - Elasticsearch
  - Couchbase
  - Voldemort
- can split and modify partitions, but fixed number of partition databases usually avoid this so best to choose a number of partitions high enough to handle future growth

Dynamic partitioning

- key-range partitioning facilitates dynamic partitioning
- when partition grows beyond a configured size, partition is split and vice versa
- similar to how B-trees work
- used by
  - HBase (uses underlying HDFS)
  - RethinkDB
- adapts to total data volume
- empty DB starts with one partition which defeats the purpose

- HBase and MongoDB allow an initial pre-split configuration to set the number of minimum partitions
- can work equally well with hash-partitioned data

Other rebalancing strategies

- before 1.2, Cassandra used consistent hashing with pseudo-random partition boundaries
- suffered from poor load distribution and made it difficult to add nodes
- replaced with fixed partitioning after
- most widely used approaches
  - hashing with a fixed number of partitions
  - dynamic partitioning by key range

Request routing

- which node to connect to for request by client?
- relates to service discovery problem (not limited to databases)
  - any piece of software running on a network has this problem
  - especially highly available software
- many in-house solutions
- approaches
  - clients contact node (via round robin load-balancer) either handles or forwards appropriately
  - send all requests to routing tier which forwards, acting as partition-aware load balancer
  - client receives node-partition mapping and handles requests
- difficult problem (related to consensus)
- ZooKeeper often used to keep mapping of partitions to nodes
- other actors can subscribe to read mapping
- uses ZooKeeper
  - Espresso -> Helix
  - HBase
  - SolrCloud
  - Kafka
- MongoDB uses config server and mongos daemons for routing
- uses gossip protocol
  - Cassandra
  - Riak
- gossip allows clients to connect to any node, removing reliance on external coordination service, but adds more complexity to database node responsibility
- Couchbase does not rebalance automatically
- still requires IP discovery for machine connections, but can use DNS for this purpose

Parallel query execution

- most NoSQL distributed data stores are limited to supporting simple queries
- massively parallel processing (MPP) relational databases can support complex queries and are often used for analytics
- business analytics makes data warehouse queries an important topic

Summary

- spread data and query load evenly across multiple machines, avoiding hot spots
- main approaches
  - key range partitioning
  - hash partitioning
  - (hybrid approaches)
- secondary index partitioning
  - document-partitioned index
  - term-partitioned index (global index)
- query routing for partition-aware load balancing or sophisticated parallel query execution
- partitions generally operate independently but operating on many partitions requires difficult reasoning

Ch. 7 Transactions

- many things can go wrong in a data system
  - software or hardware
  - crash
  - network interruptions
  - multiple conflicting concurrent writes
  - corrupted data
  - race conditions
- implementing fault tolerance requires units on which one can reason about complex data systems
- transactions -> means to group several reads and writes into one logical unit or operation
  - commit -> transaction success
  - abort (rollback) -> transaction failure
- transactions simplify the programming model
- facilitate conceptual and theoretical safety guarantees
- investigation of transactions will include look at concurrency control and isolation levels
  - read committed
  - snapshot isolation
  - serializability

The slippery concept of a transaction

- most relational and many non-relational follow concept introduced in 1975 by IBM System R, first SQL database
  - MySQL
  - PosgreSQL
  - Oracle
  - SQL Server
- many NoSQL databases abandoned the conceptual model of transactions or redefined them
  - some view transactions as the source of what makes a system unscalable
  - some view transactions (and related guarantees) as a prerequisite for serious applications handling valuable data
- both are overstatements and transactions have advantages and disadvantages

The meaning of ACID

- ACID
  - atomicity
  - consistency
  - isolation
  - durability
- safety guarantees provided by transactions
- in practice, each definition is malleable and "ACID compliant" is a label that is not necessarily clear
- BASE (systems used to refer to systems that do not meet ACID criteria)
  - basically available
  - soft state
  - eventually consistent

Atomicity

- atomic -> something that cannot be broken down into smaller parts
- in threading, if one thread performs an atomic action means there is no way another thread can see an intermediate state
- in ACID, not about concurrency
- in ACID, refers to guarantees about state after a transaction
  - if a transaction is composed of many writes and the system fails in the middle then the system guarantees the state of the writes will not be persisted

- centered on notion of transaction and guarantees about state if a conceptually atomic transaction is failed or aborted

Consistency

- overloaded word in the contexts discussed in this book
    - replica consistency and eventual consistency
    - consistency = linearizability
    - for ACID -> application-specific notion of a database being in a functionally sound state
- idea is that data has invariants and after any transactions there is still a guarantee these invariants will always be satisfied
- dependent on the application's notions of invariants and guarantees
    - e.g. how to handle bad data from user? is any resulting inconsistency beyond the scope of these guarantees
- atomicity, isolation and durability are properties of the database
- consistency (in the ACID sense) is a property of the application (so may not fit with acronym)

Isolation

- concurrently executing transactions are isolated from each other and will not affect the resulting state of the database
- isolation = serializability in classical theoretical sense
- in practice, serializable isolation causes performance issues and is rarely used

Durability

- guarantees that, once a transaction has been successfully been committed, it will never be lost or removed even if there is a hardware fault or database crash
- usually equivalent to commit to disk but also includes logging and crash recovery
- can include replication
- no such thing as perfect durability

Replication and durability

- write to disk vs. replication as the definition of durable
    - write to disk and machine death = unavailable, replication would stay available in face of a machine crash
    - correlated fault (power outage or bug) can result in all replicas being unavailable, writing to disk is still a necessary guarantee
    - SSDs have been shown to lose data on a sudden power outage
    - data storage engine interactions with the filesystem has been shown to result in subtle bugs outside of data storage engine that corrupt data (esp after crash)
    - disk data can become corrupted which can propagate through replicas
    - 30-80% of SSDs develop at least one bad block, magnetic drives have a higher rate of failure but are less susceptible to corruption

    - SSDs can lose data when not powered indefinitely
- take theoretical guarantees with a grain of salt

Single-object and multi-object operations

- atomicity and isolation describe proper behavior for multi-write transactions
    - atomicity -> error partway through should be rolled back (all-or-nothing)
    - isolation -> concurrent transactions should not affect each other
- assumes several objects may be modified at once
- multi-object transactions require the ability to identify source and interval/duration
- many non-relational databases do not provide a mechanism for grouping operations

Single-object writes

- large amounts of data (that take time to modify) still require same guarantees
    - network connection
    - power failure
    - concurrent reads/writes
- some databases provide atomic (in the sense of multithreaded programming) operations
- these atomic operations are not transactions in the usual sense of the word
    - sometimes referred to as lightweight transactions or ACID
- to compare, a transaction is a mechanism for grouping multiple operations on multiple objects into one unit of execution

The need for multi-object transactions

- many distributed datastores have abandoned multi-object transactions
- required uses
    - in relational model, foreign key references require modification in a single transaction
    - graph databases require modifications on vertices connected by edges
    - in document database, a document generally only requires one operation for updates, but joined documents require denormalization and updates to denormalized information require updating multiple documents in one transaction
    - secondary indexes require update
- without transactions, error handling becomes much more complex

Handling errors and aborts

- transactions allow for abortion and retries
- others (esp leaderless replication) follow the model of doing as much as possible but not undoing when an error is encountered
- popular ORMs (Django, Rails) do not retry aborted transactions
- retries are susceptible to
    - failed acks (network etc) which may result in client retrying manually

    - errors from overload are worsened by retries and can result in feedback cycles
    - retrying permanent errors is useless
    - side-effects outside of DB may persist even in failure
    - data will be lost if process fails while retrying

Weak isolation levels

- race conditions -> concurrent read and write, concurrent write and write
- databases try to hide concurrency issues from application developers with transaction isolation
    - theoretically provides serializability
- serializable isolation has a performance cost
    - many databases don’t want to pay that price
- common to use weaker isolation
- concurrency issues have caused
    - money loss
    - financial audit
    - data corruption
- some ACID databases provide weak isolation
- need strong understanding of concurrency issues to adequately understand and avoid them

Read committed

- most basic with two guarantees
    i. only see data that has been committed for reads (no dirty reads)
    ii. on write, will only overwrite committed data (no dirty writes)
- (also read uncommitted -> no dirty writes but does allow dirty reads)

No dirty reads

- read during incomplete transaction -> should be prevented
    - incomplete multi-object updates may cause reader to see inconsistent state
    - aborted multi-object updates may result in reading modifications that are later rolled back

No dirty writes

- avoids concurrency problems
    - incomplete multi-object updates may cause inconsistent states

- does not prevent counter updates

Implementing read committed

- default setting
  - Oracle 11g
  - PosgreSQL
  - SQL Server 2012
  - MemSQL
  - others
- writes most commonly implemented using row-level (unique) write locks
- can use read locks to avoid double reads but affects performance
- most databases prevent dirty reads by retaining old values during long running writes and displaying the new value to users only after the write transaction has completed

Snapshot isolation and repeatable

- read skew can result in reads that show an inconsistent view (example is transfer from one table to another and reading between completion of first write and before completion of second write)
- skew is an overloaded word -> means timing anomaly here
- non-repeatable read or read skew is acceptable under read committed isolation
- it is unacceptable during
  - backups
  - analytic queries and integrity checks
- snapshot isolation is the most common solution
- each transaction must read from a consistent snapshot of the database
- snapshot isolation aids reasoning about isolation
- supported in
  - PostgreSQL
  - MySQL
  - Oracle
  - SQL Server
  - more

Implementing snapshot isolation

- uses write locks to prevent dirty writes also but for reads does not use locks
  - readers never block writers
  - writers never block readers
- must maintain many versions of the data at any given time to handle dirty reads
- known as multi-version concurrency control (MVCC)
- sufficient to only keep two versions for read committed isolation

- implementation in PostgreSQL (similar to others)
    - when a transaction is started, it is given an increasing UUID
    - whene a transaction writes anything to the database, the data it writes is tagged with the transaction ID of the writer
    - each row in a table has a created by field, containing the ID of the transaction that inserted this row into the table
    - each row has a deleted by field, which is initially empty
    - if a transaction deletes a row, the row isn’t actually deleted from the database
        - it is marked for deletion by setting the deleted by field to the ID of the transaction that requested the deletion
    - when no transaction can access the deleted data, it is garbage collected

Visibility rules for observing a consistent snapshot

- database can ensure consistent snapshots by carefully defining visibility rules for transactions based on transaction IDs
    i. on transaction start, tracks each currently running transaction
    ii. writes made by aborted transactions are ignored
    iii. writes with transaction ID after current transaction are ignored even if later write has already been committed
    iv. all other writes are visible to the application's queries
- apply to both creation and deletion
- object is visible if both
    - at the time when the reader’s transaction started, the transaction which created the object had already committed
    - the object is not marked for deletion — or if it is, the transaction which requested deletion had not yet committed at the time when the reader’s transaction started

Indexes and snapshot isolation

- many details to multi-version concurrency control in practice
- PostgreSQL has optimizations for avoiding index updates if different versions of the same object can fit on the same page
- append-only/copy-on-write for B-trees
    - CouchDB
    - Datomic
    - LMDB

Repeatable read and naming confusion

- snapshot isolation is useful especially for red-only transaction
- known by different names
    - serializable -> PostgreSQL
    - repeatable read -> MySQL

- historically the SQL standard has confused these naming conventions and usages

Preventing lost updates

- many concurrent write issues
- lost update problem
  - two concurrent read-modify-write cycles can lead to second clobbering first
- occurs when
  - incrementing a counter or a balance
  - modifying a complex value with some numeric operation (i.e. value in list in JSON doc)
  - concurrent page edits (to a web based document, etc)

Atomic write operations

- atomic updates avoid implementing read-modify-write cycles
- SQL UPDATE is safe in most databases
- MongoDB provides atomic document updates
- atomic operations are usually implemented using an exclusive lock
- aka cursor stability
- can also force all atomic operations to be performed on a single thread
- ORMs expose the ability to write read-modify-write cycles that do not take advantage of DBs atomic operations

Explicit locking

- can be used for updates to objects
- difficult to get right and can result in race conditions

Automatically detecting lost updates

- can allow read-modify-write cycles to execute in parallel and a transaction manager can detect aborted transactions and force retries
- used in
  - PostgreSQL -> repeatable read
  - Oracle -> serializable
  - SQL Server -> snapshot
- not used in
  - MySQL/InnoDB
- less error prone than locking

Compare-and-set

- sometimes provide atomic compare-and-set in place of transactions
- check safety on a per database basis before relying on it

Conflict resolution and replication

- replicated databases require additional steps to avoid lost updates
- multi-leader and leaderless databases can have multiple concurrent writes to different nodes and cannot rely on locks or compare-and-set
- one technique is allowing writes to create several versions (siblings) and resolve conflicts after the fact
- can use atomic operations in a replicated context (esp commutative atomic operations)
- LWW is prone to lost updates

Preventing write skew and phantoms

- two race condition types
  - dirty writes
  - lost updates
- many more subtle bugs

Characterizing write skew

- occurs if two transactions read the same objects and then update some of those objects (different transactions can update different objects)
- generalization of lost update
- solutions are more restricted
  - atomic single-object operations don't work because many objects are involved
  - automatic lost update detection in some snapshot isolation doesn't help because requires true serializable isolation
  - some DBs allow configurable constraints but not always easy to solve with constraints
  - best option may end up being explicit locks

More examples of write skew

- related to topological sorting constraints
- booking systems, multiplayer games and physics constraints, concurrent resource claims

Phantoms causing write skew

- common events in write skew (maybe different order)
  i. a SELECT query
  ii. continue or abort based on result of 1.
  iii. on continue, make a write (INSERT, UPDATE, or DELETE) and commit transaction
  iv. start over and receive different result from SELECT
- could lock on 1. but when 1. does not return a result there is nothing to lock
- phantoms -> one transaction changes the result of a search query in another transaction

Materializing conflicts

- takes a phantom and turns it into a lock conflict on a concrete set of rows that exist in the database
- difficult and error prone - should be seen as a last resort

Serializability

- situation to this point
    - isolation levels are hard to understand and inconsistently implemented
    - large application code can be difficult to reason about and results of current isolation level may not be apparent
    - static analysis can help find race conditions but many concurrency problems are non-deterministic
- research answers that solution is serializable isolation (strongest isolation level)
- guarantees that concurrent execution can be mapped to a particular serial execution
    - in other words database prevents all possible race conditions
- options explored
    i. actual serial execution
    ii. two-phase locking
    iii. optimistic concurrency control such as serializable snapshot isolation
- each has problems
- consider from single node and then examine generalizations

Actual serial execution

- simplest solution is to remove concurrency
- only became feasible due to
    - reduced RAM cost to keep active dataset in memory
    - OLTP transactions are usually short and make a small number of reads and writes
- used in
    - VoltDB/H-Store
    - Redis
    - Datomic
- (also consider the approach of Node.js)
- can compete in performance due to avoidance of locking overhead but has a limited throughput

Encapsulating transactions in stored procedures

- rather than waiting for a user, databases encapsulate transactions and generally commit on a single HTTP request

- systems with single-threaded serial transaction processing don't allow interactive multi-statement transactions
- the application submits the entire transaction code to the database ahead of time as a stored procedure
  - instead of querying and writing, the whole transaction is a chunk of code with the logic for handling

Pros and cons of stored procedures

- existed for some time
- each has own language
  - Oracle -> PL/SQL
  - SQL Server -> T-SQL
  - PostgreSQL -> PL/pgSQL
- languages are generally pretty cumbersome and archaic
- harder to debug, version control, test, and integrate with metrics
- databases are very performance sensitive and bad stored procedures can cause serious bottlenecks
- many modern databases use general-purpose programming languages for stored procedures
  - VoltDB -> Java or Groovy
  - Datomic -> Java or Clojure
  - Redis -> Lua
- can achieve good throughput on a single trhead
- VoltDB allows stored procedures to define the replication policy

Partitioning

- single thread execution limitations can be scaled to run on separate partitions
- steps that require multiple partitions need to coordinate across partitions which adds overhead
- the ability to do this at all can depend on the structure of the data being stored

Summary of serial execution

- transactions must be small and fast
- limited to use cases where active dataset can fit in memory (can also use LRU-style caching to swap some data to disk to augment requirement)
- write throughput must be acceptably low or data needs to be able to be partitioned to require little to no cross-partition coordination

- cross-partition transactions are possible, but there is a hard limit to the extent which they can be used

Two-phase locking (2PL)

- previously the main algorithm used for database serializability
    - completely different from two-phase commit
- several transactions can read the same object as long as it is not modified
- writes require exclusive access
    - uses mutual exclusion
    - also requires that writers block readers but readers never block writers
    - provides serializability (not provided by snapshot isolation)

Implementation of two-phase locking

- used int
    - MySQL (InnoDB)
    - SQL Server
    - DB2
- lock has modes
    - shared mode
    - exclusive mode
- readers acquire lock in shared mode and wait for only exclusive mode locks to release
- writers acquire lock in exclusive mode and wait on any shared or exclusive mode locks to release
- readers can convert shared mode locks to exclusive mode locks
- all locks are held to end of transaction
- deadlock detection is automated and triggers one thread holding a lock to release and retry

Performance of two-phase locking

- waiting and lock contention reduce performance
- long running transactions add to potential bottlenecks caused by locks
- can have unstable throughput, latencies, and be unstable at high percentiles
- deadlocks are more common with 2PL serializability

Predicate locks

- serializable isolation requires the prevention of phantoms
- can be used by many objects that satisfy some condition, even those that do not yet exist
- restricts access as follows
    - transaction A reads objects matching a condition by acquiring a shared-mode predicate lock and must wait for any existing exclusive locks

    - to modify any objects, A must check for any existing matching predicate locks and wait for any to be released

Index-range locks

- aka next-key locking
- simplifies predicate locks by locking on a larger range of objects
- any modifications to overlapping ranges requires waiting
- can fall back to shared locks for cases where a query cannot be satisfied by a particular range

Serializable snapshot isolation (SSI)

- promising solution to performance hits caused by other serializable solutions
- 2008 Michale Cahill PhD thesis
- used in
    - PostgreSQL 9.1 ->
    - other distributed databases

Pessimistic vs. optimistic concurrency control

- pessimistic
    - like mutual exclusion -> if anything might go wrong, wait until situation is safe again
- serial execution is pessimistic to an extreme
- optimistic
    - don't block -> continue as normal
    - check if isolation was violated in retrospect and retry if necessary
    - only allow transactions that executed serializably
- old idea
- performs poorly under high contention
- performs much better than pessimistic approaches under low contention
- uses snapshot isolation
- adds algorithm for detecting serialization conflicts and aborting accordingly

Decisions based on an outdated premise

- write skew in snapshot isolation requires reading and deciding action based on result of read
- transaction takes action based on a potentially outdated premise
- SSI must detect situations where a transaction takes action based on an outdated premise and abort the transaction
    i. detect stale MVCC reads (uncommitted write occurred before the read)

    ii. detect writes that affect prior reads (the write occurs after the read)

Detecting stale MVCC reads

- reads from consistent MVCC snapshot ignore uncommitted writes
- if the premise on which a transaction commits is invalid by the time it commits SSI must abort the transaction
- must also support long running reads, which is what complicates the situation and keeps SSI from aborting as soon as the stale read is detected

Detecting writes that affect prior reads

- writes must look for currently executing reads and notify the transactions that the data they are currently reading may be out of date

Performance of serializable snapshot isolation

- granular tracking may reduce performance but less granular tracking may lead to more aborted writes
- sometimes stale reads are OK
    - used by PostgreSQL to reduce aborts
- SSI removes need for locking in 2PL in cases
    - read-only queries can run without locks
    - improves variability in performance
    - good for read-heavy workloads
- can distribute cost of serialization across many CPUS (unlike serial execution)
    - done by FoundationDB
- rate of aborts affects performance of SSI

Summary

- abstraction layer that reduces many considerations down to aborting the transaction
- abandoned in many NoSQL DBs
    - can result in inconsistent data on error
- isolation levels
    - read committed
    - snapshot aka repeatable read
    - serializable
- race conditions
    - dirty reads
    - dirty writes
    - read skew (non-repeatable reads)
    - lost updates

    - write skew
    - phantom reads
- only serializable isolation protects against all of the above issues
    i. actual serial execution
    ii. two-phase locking
    iii. optimistic concurrency control such as serializable snapshot isolation

Ch. 8 The Trouble with Distributed Systems

- distributed systems engineering is fundamentally different than systems engineering on a single machine

Faults and Partial Failures

- software running on a single, properly functioning computer should be deterministic so issues are bugs with the software and not issues with the system
- distributed systems operate within a model that has to assume much worse than ideal conditions (the network and physical world do not operate as idealized theoretically)
- partial failure -> one portion of system is broken in some unpredictable way even though the rest of the system may be working as expected
- partial failures are non-deterministic
- these two things (partial failures and non-determinism) are what make distributed systems difficult to work with

Cloud computing and supercomputing

- high-performance computing (HPC) -> used for computationally intensive (often scientific) computing tasks
- cloud computing -> (not well-defined) related to multitenant datacenters, commodity hardware connected via IP network, elastic/on-demand resource allocation, and metered billing for scalable resources access
- traditional datacenters lie in between
- supercomputers are more like a single computer than a distributed system and are often built from specialized hardware
- nodes are generally reliable and communicate through shared memory and remote direct memory access (RDMA)
- use specialized network topologies such as multidimensional meshes and toruses
- book focuses on systems for implementing internet services
    - online -> service users at any/all times
    - commodity hardware
    - utilize large datacenter networks which are often based on IP and Ethernet in Clos topologies to provide high bisection bandwidth

    - increase in size of system increases likelihood of a broken component
    - tolerance to failed nodes is important (e.g. rolling upgrades, node replacements)
    - geographically distributed systems often requires public internet
- must accept possibility of partial failure and build fault tolerant systems
- beneficial to test for the widest range of possible faults explicitly (chaos testing/engineering)

Building a reliable system from unreliable components

- many computing systems have error-correcting codes
- TCP can reassemble packets
- despite the possibility for error correction, there is always a bound on reliability that can be achieved when considering reliability of components

Unreliable Networks

- focus on shared-nothing systems
- asynchronous packet networks -> internet and most internal networks, a node can send a packet but network provides no time or correctness guarantees
- possible errors
    i. dropped request
    ii. stalled in a queue for delivery
    iii. receiver may have failed/crashed
    iv. receiver may have temporarily stopped responding
    v. dropped response
    vi. delayed response
- generally handled with timeouts but still do not know how to handle delayed case

Network faults in practice

- network faults are common even in controlled environments
- network partitions -> occurs when one part of the network is cut off from the rest due to a network fault (aka netsplit or network fault)
- software must account for possible network faults but does not have to tolerate them

Detecting faults

- systems need to detect faulty nodes
    - load balancer should remove a faulty request from retry rotation
    - single-leader replication leader failure should result in follower promotion
- common situations
    - can reach target machine but port is not open or service is not running the OS should respond with TCP RST or FIN packet
    - node process crashed or was killed by admin should send alert to trigger some form of failover

    - link failure at hardware level of network switches (can be queried for private network but not public internet)
    - router can reply with ICMP Destination Unreachable packets but router does not know of many of the same failures as above
- may never receive error responses

Timeouts and unbounded delays

- no simple answer to optimal timeout
- high can lead to poor user experience/application performance
- short timeout an lead to false negatives and issues where an ongoing action is superseded by a failover or other transfer of responsibility based on a false negative
- transferring responsibility incorrectly can lead to cascading failures
- asynchronous networks have unbounded delays and no guarantee of service time

Network congestion and queueing

- variability of packet delays in networks is most often due to queueing
    - two concurrent packets to a destination will get queued in a network link and delivered sequentially
    - network congestion may increase delay at a link and may lead to packet drop
    - a packet may be queued at a machine if all CPUs are busy which may take an arbitrary amount of time
    - scheduling of VMs to CPUs may cause more queued delays
    - TCP and applications perform flow control (aka congestion avoidance or backpressure) to rate limit packet sends
    - TCP invalidates packets over a time limit which may lead to retries and greater delays
- systems that are close to max capacity will suffer from these queueing delays to increasingly extreme degrees
- public clouds and data centers share resources among many tenants and batch workloads can lead to network saturation
- often need to choose application timeouts experimentally
- systems can continually measure response times and variability/jitter and adjust times based on observed response time distribution
- can be done with Phi Accrual failure detector
- used in
    - Cassandra
    - Akka
- TCP retransmission timeouts work similarly

TCP Versus UDP

- latency sensitive applications prefer UDP over TCP due to increased speed at the cost of decreased reliability

- UDP
  - no flow control
  - no packet retransmission
  - still susceptible to switch queues and other packet delays
- good choice when delayed data is useless

Synchronous Versus Asynchronous Networks

- reliable packet delivery would simplify distributed systems
- fixed-line telephone network is extremely reliable with constantly low end-to-end latency
- fixed bandwidth, synchronous network with no queueing, bounded delay

Can we not simply make network delays predictable?

- circuit-switched networks would be possible to establish a guaranteed maximum round-trip
- Ethernet and IP are packet-switched networks
- datacenters are optimized for bursty traffic
- circuit-switched networks are good for transferring a predictably constant number of bits-per-second
- packet-switched networks allow for servicing connections with variable amounts of bandwidth
- trying to predict bandwidth for normal datacenter connections could result in unpredictable misallocations of resources
- have been attempts to build hybrid netwrosk
  - Asynchronous Transfer Mode (ATM)
  - InfiniBand
- can use QoS (prioritization and scheduling of packets) and admission control (rate-limiting senders) to emulate circuit switching on packet networks
- QoS like this is not enabled in multi-tenant datacenters and public clouds

Latency and Resource Utilization

- latency -> consequence of dynamic resource partitioning
- circuit-switching -> static resource partitioning
- networks -> dynamic resource sharing
  - uses queueing which leads to potential downsides above
  - optimizes resource utilization
- similar to CPU usage within a computer
- latency guarantees are possible if resources are statically partitioned but comes at the cost less than optimal resource utilization

- variable delays in networks are result of cost/benefit trade-off

Unreliable Clocks

- clocks and understanding of time are important
  - time outs
  - response times
  - load handling (queries per second handled)
  - timing of user interactions
  - event timestamps
  - times to trigger events
  - caching and cache expiration
  - error tracking
- 1-4 -> durations
- 5-8 -> points in time
- distributed time is tricky partially due to issues above and effect on communication
- complicates ordering of events for record keeping
- each machine has its own notion of time and quartz clocks are not perfectly precise
- most common synchronization mechanism is NTP

Monotonic Versus Time-of-Day Clocks

- two types of clocks in computers

Time-of-day clocks

- current data and time according to some calendar
- usually synchronized with NTP
- oddities
  - if time-of-day clocks get too far ahead of NTP, clock may jump
  - ignore leap second
  - generally coarse grained resolution
- unsuitable for measuring elapsed time

Monotonic clocks

- suitable for measuring elapsed time due to monotically increasing guarantees
- CPUs may have own monotonic clock, OS attempts to present synchronized view
- guarantee of monotonicity is not an absolute guarantee
- NTP can slew a monotonic clock forward or back
- usually distributed systems can use monotonic clocks to measure elapsed time

Clock Synchronization and Accuracy

- monotonic -> don't need synchronization
- time-of-day -> must be synchronized via NTP
- hardware clocks and NTP both have issues
    - quartz clocks have drift (Google assumes drift of 200 ppm for servers -> 6 ms drift for server that is resynchronized every 30 seconds)
    - drift puts bound on accuracy
    - a delta that is too large from NTP can prevent resynchronization
    - accidental firewalls from NTP may go unnoticed
    - limit to accuracy of NTP due to network delay (min of 35ms)
    - some NTP servers are wrong or misconfigured
    - leap seconds occur and can crash servers
        - often best to distribute leap second adjustment over time
    - VMs virtualize hardware clock
    - if software is on a device that is not under the operator's complete control it cannot be trusted
- hardware and protocols exist to achieve much more reliable clock accuracy
    - MiFID II
    - GPS receivers
    - Precision Time Protocol

Relying on Synchronized Clocks

- software must be designed under assumption of network issues and clock issues
- clocks often seem to work fine when they are faulty
- must implement monitoring of clocks to handle faults

Timestamps for ordering events

- dangerous to rely on clocks for ordering of events across multiple nodes
- LWW
    - Cassandra
    - Riak
- LWW will cause issues when clocks are faulty
    - writes can disappear -> laggy clock will never be able to overwrite competing writes from a faster clock
    - cannot distinguish sequential writes in quick succession (requires version vectors to prevent violations of causality)
    - tiebreakers can lead to violations of causality
- logical clocks are a safer solution which are incrementing counters

Clock readings have a confidence interval

- better to think of single clock reading as representing a range of times based on confidence interval
- hardware clocks (and related crystal) have a measured spec
- often not exposed at software level
- exposed in TrueTime API in Spanner

Synchronized clocks for global snapshots

- snapshot isolation often requires a monotonically increasing snapshot ID
- difficult to generate in distributed environment due to coordination requirement
- can use the fact that two confidence intervals are only questionable if they overlap
  - Spanner waits for length of confidence interval before making a commit
  - ensures any reads are at a later time
- Google uses a GPS receiver to improve clock accuracy in a datacenter

Process Pauses

- leaders must stay leader or release leadership in spite of dangerous clocks and network issues
- lease -> similar to distributed lock with timeout between nodes
- to remain leader, node periodically renews lease
- lease renewing in a loop is not reliable if clock is not reliable
- monotonic clocks may not work in the face of processor pauses and more
  - garbage collection
  - VM suspension
  - VM live migration
  - end-user devices may be paused arbitrarily
  - OS context switching
  - slow disk I/O
  - paging and page faults (and thrashing)
  - Unix signals (accidental)
  - thread preemption

Response time guarantees

- some systems have response time deadlines
- hard real-time vs. soft real-time
- requires an RTOS, guaranteed performance within libraries, memory management, and much testing and measurement
- range of tools, hardware, libraries and programming languages becomes limited

- real-time != high-performance and may have lower throughput

Limiting the impact of garbage collection

- emerging idea is to treat GC pauses like planned node outages
    - used in some latency-sensitive financial trading systems
- can use GC only for short-lived objects and periodically restart processes to remove long-lived objects

Knowledge, Truth, and Lies

- distributed vs. single computer
    - no shared memory
    - message passing via unreliable network
    - partial failures
    - unreliable clocks
    - processing pauses
- a node cannot know anything with certainty -> it must make guesses based on messages (received and not received) via the network
- to address many semi-philosophical questions related to this, a system can state its system model -> assumptions about behavior/design of system
- algorithms can be proven correct within a model
- difficult to make software behave correctly within an unreliable system model

The Truth is Defined by the Majority

- a distributed system cannot rely on the judgment of a single node to reliably determine its state in relation to the rest of the system
- many distributed algorithms rely on quorum to address this issue
- commonly quorum = absolute majority

The leader and the lock

- often a system allows only one of some thing
    - leader for a partition (to prevent split brain)
    - one lock owner for a resource (to prevent corruption)
    - one user per username (to ensure uniqueness)
- shows actual data corruption bug HBase previously had

Fencing tokens

- fencing -> technique used to ensure owner of a protected resource does not disrupt rest of system

- lock returns a fencing token or monotonically increasing number, and all write requests must be issued with fencing token
- storage server disallows writes with lower numbers than max number for resource
- ZooKeeper achieves this with zxid or cversion
- resource must also protect against bad writes

Byzantine Faults

- fencing tokens can protect against nodes mistakenly acting in error but cannot protect against forgeries
- Byzantine fault -> messaging in a shared nothing system that is intentionally falsified
- Byzantine Generals problem -> problem of reaching consensus in an environment that can have Byzantine faults (Lamport named the problem)
- Byzantine fault tolerant -> a system that continues operation even if some of the nodes are malfunctioning and not obeying the protocol, or if malicious attackers are interfering with the network
- protocols for making systems Byzantine fault-tolerant are quite complicated, and fault-tolerant embedded systems rely on support from the hardware level
- many public web facing applications require assumption that malicious behavior will occur, but this only requires making server handle and deny as many malicious situations as possible
- most Byzantine fault tolerant software requires a supermajority so a bug deployed to all nodes cannot be Byzantine fault tolerant
- in most systems, if an attacker can compromise one node, it can compromise all of them

Weak forms of lying

- invalid messages due to hardware failure
- software bugs
- misconfiguration
- network packet corruption
  - use checksums at an application level
- user input
  - application must sanitize, limit string size, and perform other sanity checks
- misconfigured NTP server
  - usually handled by protocol (requires a majority among many servers) but should prefer configuration with multiple servers over one server

System Model and Reality

- distributed algorithms need to handle distributed faults and be generally hardware independent

- system models for timing
    - synchronous -> assumes bounded network delay, bounded process pauses, and bounded clock error (not realistic for most systems)
    - partially synchronous -> behaves like a synchronous system most of the time, but it sometimes exceeds the bounds for network delay, process pauses, and clock drift
    - asynchronous model -> no timing assumptions (no clock and timeouts - difficult model to design algorithms for)
- system models for crashes
    - crash-stop faults -> algorithm may assume that a node can fail in only one way, namely by crashing (and will never come back)
    - crash-recovery faults -> nodes may crash at any moment, and perhaps start responding again after some unknown time (nodes have stable storage but in-memory state is lost)
    - Byzantine (arbitrary) faults -> nodes may do absolutely anything, including trying to trick and deceive other nodes

Correctness of an algorithm

- uniqueness -> no two requests for a fencing token return the same value.
- monotonic sequence -> if request x returned token tx, and request y returned token ty, and x completed before y began, then tx < ty.
- availability -> a node that requests a fencing token and does not crash eventually receives a response
- more properties for correctness exist

Safety and liveness

- safety -> nothing bad happens
    - uniqueness
    - monotonicity
- liveness -> something good eventually happens
    - availability
    - eventual consistency
- actually have precise and mathematical definitions
    - if safety is violated, it happened at a point in time and cannot be undone
    - liveness is in some ways opposite -> may not hold for an extended time but may hold at some time in the future
- common for distributed algorithm to require that a safety property always holds

Mapping system models to the real world

- many other issues not mentioned occur
    - corrupted data
    - wiped out data
    - firmware bug

    - etc
- some algorithms state that certain conditions are assumed not to happen
- algorithm correctness != real-world implementation correctness

Summary

- many common issues
    - packet delay and packet loss and how to respond
    - clock issues between machines and unreliability of clocks
    - process pauses, e.g. garbage collector, and relation to node agreement
- partial failures are a defining characteristic of distributed systems
- partial fault tolerance needs to be built in
- first step is to detect partial faults
- handling faults is difficult due to no shared state and simple agreement
    - require protocols to handle faults
- many distributed problems are trivial when considered in the context of a single node
- distributed systems are not just good for scalability, also good for fault tolerance and low latency
- clock/timing issues are the result of a cost/benefit trade off
- supercomputers (e.g. scale up) are generally not fault tolerant in the face of hardware failure whereas distributed systems can ideally run forever

Ch. 9 Consistency and Consensus

- focus is finding abstractions that allow distributed systems to continue functioning in the face of faults
- consensus -> reaching agreement between nodes in distributed systems

Consistency Guarantees

- state of two nodes at any given moment is likely to be different regardless of the replication model used
- eventual consistency -> convergence is a weak guarantee as it doesn't give a time for convergence
- programming for weak guarantees requires never assuming too much
- stronger guarantees affect performance and fault-tolerance
- similarities between consistency and transaction isolation
    - linearizability
    - ordering of events
    - atomic commit

Linearizability

- aka
    - atomic consistency
    - strong consistency
    - immediate consistency
    - external consistency
- basic idea is to make a system appear as if there is only one copy of the data and all operations are atomic
- on successful write, all clients must be able to observe written value
- linearizability = recency guarantee

What Makes a System Linearizable?

- example 1
    - if a read request is concurrent with a write request it may return the old or new value
- due to variability of networks and clocks, events/requests/responses occur over a range
- generally two operations
    - read(x) => v ->
        - the client requested to read the value of register x, and the database returned the value v
    - write(x, v) => r ->
        - the client requested to set the register x to value v, and the database returned response r (which could be ok or error)
- possible responses
    - read operation happens before write so read must return old value
    - read begins after write completed so it must definitely return new value
    - if the read operation overlaps with the write operation it can return either the old or new value
- concurrent reads and writes can lead to a value toggling
- linearizable system requires one point in time in an operation in which the operation can be viewed as taking effect
- constraint
    - after any one read has returned the new value, all following reads (on the same or other clients) must also return the new value
- additional operation
    - cas(x, vold, vnew) => r ->
        - the client requested an atomic compare-and-set operation
        - if the current value of the register x equals vold, it should be atomically set to vnew
        - if x != vold then the operation should leave the register unchanged and return an error
        - r is the database’s response (ok or error)

- requirement of linearizability is that the global set of linearization points for operations on a system are monotonically increasing over time
- details
    - concurrent requests can be viewed as being arbitrarily ordered even though they may have a strict ordering when compared in absolute time
    - a response to a client will happen after the actual event and must be considered to avoid causal confusion
    - no transaction isolation is assumed
- refer to formal definition for more precision

Linearizability Versus Serializability

- important to distinguish differences
- serializability ->
    - isolation property of transactions with single transactions reading and writing to potentially many objects
    - guarantees that transactions behave as if they had executed in some serial order but does not guarantee that serial order is order in which they were actually run in absolute time
- linearizability ->
    - recency guarantee on reads and writes of a register (individual object)
    - no concept of operations on many objects within a larger operation
- strict serializability (strong one-copy serializability) -> combination of both guarantees within a database
    - 2P locking
    - actual serial execution
- serial snapshot isolation is not linearizable by design

Relying on Linearizability

Locking and leader election

- single-leader replication needs to guarantee against split brain
- can use a lock
- lock must be linearizable and agreed upon by all nodes
- coordination services
    - ZooKeeper
    - etcd
- used to implement distributed locks and leader election
- Apache Coordinator provides higher-level library on top of ZooKeeper

- Oracle Real Application Clusters uses more granular distributed locking

Constraints and uniqueness guarantees

- need linearizability for hard uniqueness constraints
- similar to a lock and atomic compare-and-set
- sometimes acceptable to treat uniqueness constraints loosely

Cross-channel timing dependencies

- race conditions can occur when two different message channels for resource access and persistence exist within a system
- linearizability is not the only method for avoiding these race conditions but it is the easiest to understand
- if the 2nd communication channel is under the application developer's control, race conditions can be avoided at the cost of additional complexity

Implementing Linearizable Systems

- to tolerate faults cannot take the simple approach of only storing one copy of the data, so must use replication
- replication models
  - single-leader -> potentially linearizable
  - consensus algorithms -> linearizable
  - multi-leader replication -> not linearizable
  - leaderless replication -> probably not linearizable

Linearizability and quorums

- seems that strict quorum reads and writes should be linearizable in a Dynamo-style model
- variable network delays cause race conditions in this model
- to make Dynamo-style quorums linearizable a reader must perform read repair synchronously and writer must read latest state of quorum before sending write
- affects performance
- Riak does not perform synchronous read repair
- Cassandra waits for read repair on quorum reads but loses linearizability due to LWW
- cannot implement linearizable compare-and-set with quorums because it requires a consensus algorithm

- safest to assume that Dynamo-style replication does not provide linearizability

The Cost of Linearizability

- split datacenters (no network communication between them) can lead to interesting problems for single-leader database (with multi-leader each datacenter can operate normally and sync up when network is restored)
- clients in single-leader that can only connect to the datacenter without the leader will experience an outage

The CAP theorem

- any linearizable database has the above issue no matter how it is implemented
- trade-offs
  - required linearizability in a network disconnection will cause unavailability
  - multi-leader replication can remain available in the case of a network disconnection
- Eric Brewer -> CAP theorem 2000
- started as rule of thumb
- formal definition has a very narrow scope
- little practical value and has been superseded by more precise results

The Unhelpful CAP Theorem

- often presented as pick 2 out of 3
  - consistency
  - availability
  - partition tolerance
- misleading because network partitions are a kind of fault and will happen inevitably
- better way of phrasing would be consistent of available when partitioned
- several contradicting definitions of the term availability
- due to many misunderstandings around CAP it is best avoided

Linearizability and network delays

- few systems are linearizable in practice
  - even threaded systems can lead to non-linearizable results unless using a memory barrier or fence
- due to cache systems in modern architectures
- reason for dropping linearizability is performance and not fault tolerance
- proven that to have linearizability, the response time of read and write requests is at least proportional to the uncertainty of delays in the network

- weaker consistency models can be faster

Ordering Guarantees

- ordering is a recurring theme
  - single-leader replication determines order of writes to apply writes
  - serializability ensures sequential ordering in transactions
  - timestamps and clocks and distributed systems are mechanisms for adding order to difficult and disorderly problems
- deep connections between ordering, linearizability, and consensus

Ordering and Causality

- ordering helps preserve causality
  - consistent prefix reads -> out of order causality can cause problems for users (think chat app)
  - writes can overtake others due to network delays and modifications to a record may happen in reverse order which causes problems when the writes are not commutative
  - consistent snapshots mean consistent with causality or in a causally sound order
  - write skew has causal dependencies
  - cross-channel timing dependencies lead to causal dependencies
- causality imposes ordering
- causally consistent -> obeys causal ordering

The causal order is not a total order

- total ordering permits unequivocal comparison
- mathematical sets are sometimes incomparable and therefore partially ordered
  - some sets are subsets or supersets and thus can be compared
- total order vs. partial order
  - linearizability = total order
  - causality = partial order
- linearizability means no concurrent operations
- version control systems keep histories as a graph (possibly linearly or tree-like DAG)

Linearizability is stronger than causal consistency

- linearizability implies causality
- can harm performance and availability
- middle ground is possible
- causal consistency is strongest possible consistency model that does not slow down due to network delays and remains available in the face of network failures
- new research into database systems that only preserve causality

Capturing causal dependencies

- what happened before, partial order
- similar techniques to what is outlined in "Detecting Concurrent Writes"
- causal consistency has to track causal dependencies across an entire database (can be done with version vectors)

Sequence Number Ordering

- do not need to explicitly track all read data
- can use sequence numbers or timestamps (logical clock)
- provide a total order and can be consistent with causality
- replication log in single-leader replication provides a total order of write operations that is consistent with causality
- if follower applies writes in order of replication log it will maintain causal consistency

Noncausal sequence number generators

- various methods for generating timestamps in environments without a single-leader
    - each node can maintain own set of sequence numbers and encode node ID within it
    - sufficiently high-resolution time-of-day timestamps may not ensure sequential ordering but can provide causal ordering
    - each node can be given a particular block of sequence numbers which can be reallocated as needed
- not causally consistent

Lamport timestamps

- simple method for generating sequence numbers that are causally consistent
    - pair of (counter, node ID)
    - each node and client maintain the maximum timestamp seen so far and it is included on each request
    - node increases max on receipt of a request with higher timestamp
- using max ensures causal consistency

Timestamp ordering is not sufficient

- creation of conflicting resources on separate nodes may succeed on both nodes but then a winner will be selected by timestamp after the fact, not allowing concurrent creation would lead to a system bottleneck
- to implement something like a uniqueness constraint it is not sufficient to rely on a total ordering of operations
- also need to know when the resolution of the order of operations has occurred

Total Order Broadcast

- determining a total order of operations is simpler in a single-leader context
- total order broadcast or atomic broadcast -> a protocol for exchanging messages between nodes with the following safety properties
    - reliable delivery -> no messages are lost; if a message goes to one node it goes to all nodes
    - totally ordered delivery -> messages are delivered to every node in the same order
- ensure these safety guarantees using retries

Using total order broadcast

- ZooKeeper and etcd implement total order broadcast
- state machine replication ->
    - if every message represents a write to the database, and every replica processes the same writes in the same order, then the replicas will remain consistent with each other (aside from any temporary replication lag)
- exactly what is required for database replication
- order is fixed at point of message delivery
- similar to creating a log
    - replication log
    - transaction log
    - write-ahead log
- can be used to implement a lock service with fencing tokens (zxid in ZooKeeper)

Implementing linearizable storage using total order broadcast

- close links between the total order broadcast and linearizability
    - total order broadcast is async with no delivery timing guarantee
    - linearizability guarantees recency, i.e. a read will return the most recent write
- can build linearizable storage on top of total order broadcast
- linearizable compare-and-set with unique resource
    - append message to log with tentative operation indication
    - read log and wait for previous message to be returned
    - check for any other claims on unique resource
        - if none, commit
        - otherwise, abort
- a similar approach can be used to implement serializable multi-object transactions on top of a log
- procedure does not guarantee linearizable reads
    - provides sequential consistency or timeline consistency

- to linearize reads
    - sequence reads by first writing to log and reading when log is returned after write (similar to quorum reads in etcd)
    - if fetching latest log message is linearizable, can fetch and wait for all entries up to that point to be delivered and then perform the read
    - read from a replica that is synchronously updated on writes

Implementing total order broadcast using linearizable storage

- can invert order - build a linearizable compare-and-set operation from a total order broadcast
    - use linearizable register with an integer that has an atomic increment-and-get operation
    - atomic compare-and-set
- algorithm
    - increment-and-get the linearizable register for each message sent through the total order broadcast
    - send the message to all nodes (retrying for lost messages) and recipients will deliver messages consecutively by sequence number
- form sequence with no gaps (unlike Lamport timestamps)
- algorithms for linearizable sequence number generation lead to consensus
    - linearizable compare-and-set (or increment-and-get) registers and total order broadcast are both equivalent to consensus (provably)

Distributed Transactions and Consensus

- one of the most important and fundamental problems in distributed computing
- simple view -> get several nodes to agree on something
- requires a very subtle understanding which only developed over decades in academia
- situations where consensus is required
    - leader election
    - atomic commit

The Impossibility of Consensus

- FLP result (Fischer, Lynch, Paterson)
    - no algorithm that is always able to reach consensus if there is a risk that a node will crash
- this is proven in an asynchronous system model (no clocks or timeouts)
- in a system that can use timeouts or some other method for identifying suspected crashed nodes, consensus becomes solvable

Atomic Commit and Two-Phase Commit (2PC)

- transaction is for atomicity when making several writes and transaction is either committed or aborted

- especially important for multi-object modifications and secondary indexes

From single-node to distributed atomic commit

- on a single node transaction commitment crucially depends on the order in which data is durably written to disk
    1. data
    2. commit record
- thus creating an atomic commit is dependent on the drive
- for distributed databases, it can easily happen where a commit succeeds on some nodes and fails on others which leads to inconsistencies
- transactions are irrevocable and a node must only commit once it is sure all other nodes in the transaction are going to commit
    - once data has been committed, it becomes visible to other transactions, and thus other clients may start relying on that data
- basis of read committed isolation
- (possible for transaction rollback based on compensating transactions which are separate transactions)

Introduction to two-phase commit

- method for achieving atomic transaction commit across multiple nodes
    - used in some databases (XA transactions)
    - used in SOAP web services
- 2PC is very different from two-phase locking
- process
    - coordinator or transaction manager (can be library within application process or a separate process or service)
    - start with reading/writing on multiple nodes or participants
    - on commit begin coordinator
        a. sends a prepare request to each node
            - if all acks, coordinator sends commit request in phase 2
            - if any failed acks, coordinator sends an abort request in phase 2

A system of promises

- process in detail
    i. transaction start request from application is given UUID from coordinator
    ii. application sends transaction to each node with UUID, abort on error
    iii. on commit prepare, coordinator sends prepare request with UUID, aborts on error
    iv. on prepare request, each node checks for conflicts or errors and acks back
    v. when all responses are received by coordinator, coordinator logs (commit point)
    vi. once coordinator logs commit, it must ensure all nodes receive notification (by retrying forever) to commit also

- main points in process
  - participant's decision to confirm request
  - coordinator's decision to commit after receiving all positive responses

Coordinator failure

- crash before sending prepare request -> participant abort
- after acking back, node can only abort or commit if all others abort or commit
- if coordinator crashes after commit but before sending commit request to nodes then the only way to continue is for the coordinator to recover

Three-phase commit

- 2PC = blocking atomic commit
- three-phase commit (3PC) is a method for non-blocking atomic commits but assumes bounded network delay (requires perfect failure detector)

Distributed Transactions in Practice

- often avoided in practice
- can kill performance
  - additional network round-trips
  - additional forced disk sync (fsync)
- two types of ditributed transactions
  - database-internal distributed transactions -> same software on all nodes
  - heterogeneous distributed transactions -> different systems (software, etc) communicating via message brokers
- internal can specify own protocol
- heterogeneous is much more complicated

Exactly-once message processing

- message brokers and databases can communicate based on atomic commits and distributed transaction support
- all systems affected by the transactions must be able to use the same atomic commit protocol

XA tranasactions

- X/Open XA (eXtended Architecture) -> standard for 2PC across heterogeneous technologies
- supported in
  - PostgreSQL
  - MySQL
  - DB2
  - SQL Server

    - Oracle
- message brokers that support it
    - ActiveMQ
    - HornetMQ
    - MSMQ
    - IBM MQ
- just a C API for interfacing with transaction coordinator
- more info for it here

Holding locks while in doubt

- system cannot continue when a doubtful transaction is stuck due to locking on the related data (both written and read)
- other parts of the system cannot continue because if they want to access the data they will be blocked
- can cause large parts of application to become unavailable

Recovering from coordinator failure

- on coordinator crash orphaned in-doubt transactions can occur which can lead to locks and blocking
- requires an administrator to manually commit or rol-back in-doubt transactions
- this often has to be done under high-pressure situations
- many XA implementations provide an emergency escape hatch
    - heuristic decisions (probably breaks atomicity)

Limitations of distributed transactions

- transaction coordinator becomes a form of database
    - HA? potential single point of failure
    - removes the potential for stateless application servers
    - XA is a lowest common denominator for compatibility
        - no deadlock detection
        - not compatible with SSI
    - 2PC and requirement that all participants must respond can lead to amplifying or cascading failures

Fault-Tolerant Consensus

- normally formalized as
    - one or more nodes propose values
    - consensus algorithm decides on values
    - satisfies following properties
        - uniform agreement -> no two nodes come to different decision

        - integrity -> no node decides twice
        - validity -> if a node decides value v, then v was proposed by some node
        - termination -> every node that does not crash eventually decides some value
- first three properties are relatively easy to satisfy without fault tolerance
- termination formalizes idea of fault tolerance
    - guarantees progress even when some nodes fail
- assumption is that node disappears when it crashes (not satisfied by 2PC)
- mathematically provable that any consensus algorithm requires a majority of nodes to be functional in order to assure the termination property
- most consensus algorithm implementations ensure the safety properties are always met
- most assume no Byzantine faults
- possible to harden consensus against Byzantine faults if fewer than 1/3 of nodes are Byzantine faulty

Consensus algorithms and total order broadcast

- best known
    - Viewstamped Replication
    - Paxos
    - Raft
    - Zab
- difficult to implement on your own
- most decide on a sequence of values (e.g. total order broadcast)
- total order broadcast = repeated rounds of consensus
    - due to the agreement property of consensus, all nodes decide to deliver the same messages in the same order
    - due to the integrity property, messages are not duplicated
    - due to the validity property, messages are not corrupted and not fabricated out of thin air
    - due to the termination property, messages are not lost
- Viewstamped Replication, Raft, and Zab implement total order broadcast directly
- Multi-Paxos (Paxos variant) is more efficient for total order broadcast

Single-leader replication and consensus

- manual choice of single-leader results in dictatorial consensus and total order broadcast
- automatic leader election and failover is closer to fault-tolerant total order broadcast, and thus to consensus

- need consensus to prevent split brain which leads to a conundrum loop between total order broadcast, single-leader replication, and consensus

Epoch numbering and quorums

- no guarantee of unique leader in consensus algorithms
- epoch number -> time period within which each leader is unique
- aka
  - ballot number
  - view number
  - term number
- description of leader election algorithm here
  - involves two rounds of voting (superficially similar to 2PC)

Limitations of consensus

- benefits come at a cost
- leader election is a form of synchronous replication which can lead to data loss on failover when misconfigured
- requires strict majority which means requires at least 3 nodes for one failure or 5 for two failures
- dynamic membership consensus algorithms are much less understood than static membership algorithms
- the reliance on timeouts can lead to frequent mistaken leader re-elections on highly variable networks which will damage performance
- Raft has unpleasant edge cases in face of network problems
  - leadership can bounce between two nodes
  - leader may continually be forced to resign, stopping progress
- other algorithms have similar problems

Membership and Coordination Services

- ZooKeeper & etcd
  - distributed key-value stores
  - coordination and configuration services
- basically databases with consensus but rarely useful directly for an application developer
- applications that use ZooKeeper
  - HBase
  - Hadoop
  - YARN
  - OpenStack Nova
  - Kafka

- hold small amounts of data that fit in memory
    - replicated across nodes using fault-tolerant total order broadcast
- ZooKeeper modeled after Google's Chubby lock service
- other included features
    - linearizable atomic operations
    - total ordering of operations
    - failure detection
    - change notifications
- only linearizable atomic operations require consensus

Allocating work to nodes

- useful for leader election in single-leader database but also for job schedulers and similar stateful systems
- partition assignment to nodes
    - database
    - message streams
    - file storage
    - distributed actor system
- operations can be achieved by use of atomic operations, ephemeral nodes, and notifications in ZooKeeper
- not easy to use but easier than starting from scratch
- ZooKeeper works with a fixed number of nodes (3 or 5 usually) and can outsource coordination work
- ZooKeeper generally manages information that changes slowly
- Apache BookKeeper is a good option for replicated data that represents application state

Service discovery

- service discovery -> find out IP/port to connect to to connect to a particular service
    - ZooKeeper
    - etcd
    - Consul
- variable nodes can register in central service registry
- unclear whether service discovery actually requires consensus
- DNS is traditional method for discovering IP
    - not linearizable
    - more important for DNS to be reliable and robust to network issues
- leader election does require consensus

Membership services

- ZooKeeper, etcd, Consul are part of history of research into membership services

- determines active and live members of a cluster
- coupling failure detection with consensus makes it possible to agree on whether a node should be considered alive and a member or not
- can still result in incorrect dead declarations

Summary

- linearizability is like access a variable through coarse-grained locks which affects performance
- linearizability
  - puts all operations in a single, totally ordered timeline,
- causality
  - weaker consistency model -> some things can be concurrent, so the version history is like a timeline with branching and merging
- ensuring uniqueness led to consensus
- achieving consensus means deciding something in such a way that all nodes agree on what was decided, and such that the decision is irrevocable
- equivalent problems
  - linearizable compare-and-set registers
    - register needs to atomically decide whether to set its value, based on whether its current value equals the parameter given in the operation
  - atomic transaction commit
    - database must decide whether to commit or abort a distributed transaction
  - total order broadcast
    - messaging system must decide on the order in which to deliver messages
  - locks and leases
    - several clients are racing to grab a lock or lease, the lock decides which one successfully acquired it
  - membership/coordination service
    - given a failure detector (e.g., timeouts), the system must decide which nodes are alive, and which should be considered dead because their sessions timed out
  - uniqueness constraint
    - when several transactions concurrently try to create conflicting records with the same key, the constraint must decide which one to allow and which should fail with a constraint violation

Ch. 10 Batch Processing

- three types of systems for data processing
  - services (online systems) -> request/response, measured in response time
  - batch processing (offline systems) -> runs a job on large amount of data periodically, measured in throughput

    - stream processing (near-real-time systems) -> consumes input, produces outputs like batch, but operates on events close to their point of occurrence in time
- MapReduce -> algorithm published in 2004
    - Hadoop
    - CouchDB
    - MongoDB
- MapReduce is important still but declining
- many batch processing concepts come from Unix tools

Batch Processing with Unix Tools

- servers and systems store log files with complex, line-by-line info on events/requests

Simple Log Analysis

```
cat /var/log/nginx/access.log |
     awk '{print $7}' |
     sort             |
     uniq -c          |
     sort -r -n       |
     head -n 5
```

- pipeline for data processing the log file
- sometimes obscure syntax but very powerful
- many forms of data analysis can be done quickly using
    - awk
    - sed
    - grep
    - sort
    - uniq
    - xargs

Chain of commands versus custom program

- can write simple imperative script to do same thing (Python, Ruby, etc)
- big difference in execution flow especially when running on a large file

Sorting versus in-memory aggregation

- script pulls data in memory so for large datasets the data may need to be split across several files
- GNU `sort` automatically handles data sets larger than memory
- mergesort can be utilized
- simple Unix commands above scale more easily than Ruby script

The Unix Philosophy

- connect programs like plumbing and have utilities that do one thing and do it well
- outline
    i. Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new “features”
    ii. Expect the output of every program to become the input to another, as yet unknown, program. Don’t clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don’t insist on interactive input.
    iii. Design and build software, even operating systems, to be tried early, ideally within weeks. Don’t hesitate to throw away the clumsy parts and rebuild them.
    iv. Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you’ve finished using them.
- striking similarity to Agile and DevOps
- Unix shell enable composable programs

A uniform interface

- need to be able to connect output of one program to input of another
- file descriptors (socket, stdin, stdout, stderr)
- often use ASCII text with common "\n"
- generally a line is a record but many tools expose options for selecting various record separators
- still elegant many decades later and few programs now work to interoperate as easily as Unix tools
- many databases with the same data model are difficult to interoperate

Separation of logic and writing

- can read/write from/to stdin/stdout or files
- facilitated by pipes
- form of loose coupling, late binding, or inversion of control
- programs can be limited when more input or output sources are needed

Transparency and experimentation

- very easy to see the underlying processes of each tool
- often idempotent -> immutable input files
- can pipe to less to debug
- can write output at one stage to file and use file for input to next stage
- biggest limitation is that Unix tools only run on a single machine

MapReduce and Distributed Filesystems

- takes one or more inputs and produces one or more outputs but can operate distributed across potentially thousands of machines
- writes output to files once sequentially
- operates on files on a distributed filesystem
- for Hadoop -> HDFS (Hadoop Distributed File System) = open source Google File System (GFS)
- other distributed file systems
  - GlusterFS
  - QuantcastFS (QFS)
  - NFS
  - (https://en.wikipedia.org/wiki/List_of_file_systems#Distributed_file_systems)
- object storage is similar
  - Amazon S3
  - Azure Blob Storage
  - OpenStack Swift
- Hadoop -> shared nothing (in contrast with shared-disk NAS and SAN possibly using Fibre Channel)
- HDFS
  - daemon process exposing network service on each machine
  - central server NameNode
  - uses replication (both copies on machines or erasure coding -> possibly Reed-Solomon)

MapReduce Job Execution

- general possible process for MapReduce
  i. read set of input files, broken into records
  ii. called map function extracts key-value from each record
  iii. sort key-value pairs by key
  iv. call reducer
- implement two callbacks; mapper and reducer

Distributed execution of MapReduce

- parallelizes jobs similar to Unix utilities across many machines
- Hadoop MapReduce uses Java classes
- MongoDB and CouchDB use JavaScript functions
- scheduler runs on each machine and backs up input file to run mapper and reducer in partitioned manner

- to combine and sort data set
    - each map task partitions its output by reducer based on hash of key and is stored on mapper's machine
    - reducers do similar operation in process called shuffle

MapReduce workflows

- one single MapReduce job is limited in what it can achieve
- common to chain jobs into workflows
    - done implicitly by directory name, filenaming protocols
    - less like Unix pipelining and more like usage of temporary files between jobs
- Hadoop workflow schedulers
    - Oozie
    - Azkaban
    - Luigi
    - Pinball
- schedulers also have management features for maintaining large collections of batch jobs
- additional higher-level tools
    - Pig
    - Hive
    - Cascading
    - Crunch
    - FlumeJava

Reduce-Side Joins and Grouping

- an association within one record to another record
    - foreign-key in a relational model
    - document reference in document model
    - edge in graph model
- smaller queries grab records by index
- MapReduce has no concept of indexes -> uses full table scans over set of input files
- joins in batch processing = resolving all references to an association within a full data set

Example: analysis of user activity events

- good throughput in batch processing requires localizing requests to one machine
- random access requests over the network will slow the process drastically

Sort-merge joins

- secondary sort -> mapper can arrange sorted output such that reducer always sees record first followed by timestamp

- sort-merge join -> reducer merges together sorted lists of records from both both sides of joind

Bringing related data together in the same place

- kind of like mappers sending messages to reducers
- MapReduce separates physical network communication aspects of logic (required when accessing traditional database systems) from application logic (logic required to process retrieved data)
- also shields programmer from handling failures, transparently managing retries

GROUP BY

- common action is bringing related data to the same place
    - all records with the same key form a group
    - aggregation of some form by groups
        - counting
        - adding fields
        - ranking
- grouping and joining look very similar inside MapReduce
- sessionization -> use of grouping to collate activity events for a particular user session to find sequence of actions taken by user during the session
- user's actions may be spread across log files on many servers

Handling skew

- grouping can break down if there is a very large amount of data related to a group
- linchpin objects or hot keys -> database records with a disproportionately high number of connections to other records
- can lead to skew or hot spots -> reducers that must process a significantly higher amount of data than others
- there are a few algorithms to handle this problem
    - Pig -> skewed join method samples first to determine hot spots
    - Crunch -> sharded join - similar but requires explicitly specifying hot keys
    - Hive -> hot keys stored in metadata, stores separately, and performs mapside join

Map-Side Joins

- reduce-side joins -> joins performed during reducer step
- advantage
    - no need for assumptions about input data
- disadvantage
    - sorting, copying to reducers, and merging of reducer input is expensive

- map-side join -> cut-down MapReduce job with no reducers and no sorting
    - mapper just reads one input file from distributed filesystem and writes one output file

Broadcast hash joins

- mapper starts, reads (small enough to fit into memory) user database from distributed filesystem into an in-memory hash table, and scans over hash table by IDs/keys
- broadcast -> each mapper for a partition of the large input reads the entirety of the small input which is broadcast to large input
- supported by
    - Pig -> replicated join
    - Hive -> MapJoin
    - Cascading
    - Crunch
    - Impala

Partitioned hash joins

Map-side merge joins

MapReduce workflows with map-side joins

- choice of map-side or reduce-side affects the structure of the output
    - output of a reduce-side join is partitioned and sorted by the join key
    - output of a map-side join is partitioned and sorted in the same way as the large input
        - one map task is started for each file block of the join's large input, regardless of whether a partitioned or broadcast join is used
- knowing about the physical layout of datasets in the distributed filesystem becomes important when optimizing join strategies
    - not sufficient to just know the encoding format and the name of the directory in which the data is stored
    - must also know the number of partitions and the keys by which the data is partitioned and sorted
    - maintained in HCatalog and Hive metastore in Hadoop ecosystem

The Output of Batch Workflows

- purpose of running jobs?
- batch processing isn't transaction processing or analytics, often leads to something other than a report on the data

Building search indexes

- Google originally used MapReduce to build indexes for its search engine
- abandoned by Google but still good for building indexes for Lucene/Solr

- search index is like Lucene term dictionary but with additional metadata etc
- building search index with batch operation is a good operation and results in immutable, read-only files which are good for this operation
- can periodically rerun or build indexes incrementally

Key-value stores as batch process output

- common to build machine learning systems such as classifiers
- can run batch jobs and write results back into database through driver for MapReduce job language within mapper or reducer
- bad idea
    - network requests for each record is very slow
    - reduces benefits of parallelization within MapReduce and potentially cause other operational problems within the system
    - takes away clean success or failure result in MapReduce jobs
- better to build new database inside batch job and write it as output files
- supported in
    - Voldemort
    - Terrapin
    - ElephantDB
    - HBase

Philosophy of batch process outputs

- explicit input to immutable output files avoids side effects and facilitates maintenance
    - easy to roll back issues (human fault tolerance)
    - feature development can proceed quickly, minimizing irreversibility
    - automatic retries on failure with immutable inputs and discarded failed tasks
    - can use input files for many jobs which allows metrics and comparison
    - separation of concerns, code reuse
- can reduce issues from less structured text storage in Hadoop by using Avro, Parquet, etc

Comparing Hadoop to Distributed Databases

- like Unix with HDFS = filesystem and MapReduce = process/shell utility
- many parallel components in MapReduce were not new
- MPP (massively parallel processing) allows execution of queries whereas HDFS and MapReduce are more like a general purpose OS with arbitrary utility program execution

Diversity of storage

- Hadoop allows indiscriminate data dumping into HDFS
- MPP requires careful schema planning

- difference of value is in practice and some due to human nature tendencies
- similar idea to data warehousing
- shifts responsibility from producer to consumer
- sushi principle -> raw data is better

Diversity of processing models

- MPP is associated with tightly coupled software components and SQL queries
- not all kinds of processing can be expressed with SQL queries
- MapReduce allows engineers to easily run custom code over large data sets
- can build SQL query engine on top of MapReduce and HDFS
  - Hive
- MapReduce can be limiting and perform poorly on certain types of operations
- led to creation of more data processing models with Hadoop
- additional parts of ecosystem
  - HBase -> OLTP random-access database
  - Impala -> MPP-style analytic database

Designing for frequent faults

- MapReduce vs. MPP databases -> handling faults vs. use of memory and disk
- MPP
  - node crash causes re-execution of query
- TODO: more here
- open source cluster schedulers
  - YARN's CapacityScheduler
  - Mesos
  - Kubernetes

Beyond MapReduce

- MapReduce is a useful learning tool but is not as hyped as it once was
- provides simple abstraction on top of distributed filesystem
- abstractions on MapReduce
  - Pig
  - Hive
  - Cascading
  - Crunch
- higher-level programming abstractions provide ways to make batch processing and workflow tasks even easier

- MapReduce can be slower than other tools because of issues with the execution model

Materialization of Intermediate State

- each job is independent, need to use directories for job output to job input for workflow scheduler to connect
- intermediate state -> intermediate job output files are not going to be used beyond that workflow (materialization)
- Unix pipes do not generally materialize intermediate state but rather stream output to input (using small memory buffer)
- downsides of MapReduce's materialization
  - must be sequential
  - mappers are often redundant
  - intermediate state in distributed filesystem means replicas of intermediate state as well

Dataflow engines

- new engines for distributed batch computations
  - Spark
  - Tez
  - Flink
- handle entire workflow as one job
- more flexible methods for operating on data using operators
  - can repartition and sort
  - can partition without sorting
  - same output from one operator can be sent to all partitions for join operator in broadcast hash joins
- based on research systems like Dryad and Nephele
- benefits
  - only perform expensive operations where required
  - do not necessarily need map, can perform map and reduce in one operation
  - scheduler can optimize over a workflow
  - can keep intermediate state in memory
  - operators can operate on input asap so no blocking for task end
  - JVM instances can be reused between operations

Fault tolerance

- intermediate state has the advantage of being able to be used after a fault
- Spark, Flink, Tez fault tolerance
  - if a machine fails and the intermediate state on that machine is lost, it is recomputed from other data that is still available

- Spark uses the resilient distributed dataset (RDD) abstraction for tracking the ancestry of data
- Flink checkpoints operator state, allowing it to resume running an operator that ran into a fault during its execution
- must know computations are deterministic
- cheaper to materialize the intermediate data to files than to recompute if the intermediate data is much smaller than the source data or the computation is very CPU-intensive

Discussion of materialization

Graphs and Iterative Processing

- often interesting to run offline computations on graph-like data
  - machine learning
  - ranking systems (PageRank)
- Spark, Flink, Tez run operators in DAG which is not the same as what is being discussed here
- can store graph in a distributed filesystem
- must run graph traversal/processing in iterative style
  - scheduler runs batch for each step of algorithm
  - on batch completion scheduler checks for total completion and runs again if necessary

The Pregel processing model

- bulk synchronous parallel (BSP) model of computation has become popular (aka Pregel model)
  - Apache Giraph
  - Spark's GraphX API
  - Flink's Gelly API
- mappers send messages to reducers and similarly vertices can send messages to vertices along edges
- similar to actor model

Fault tolerance

- only waiting is between iterations because messages sent in one iteration are delivered in next iteration
- Pregel guarantees messages are processed only once at their destination
- recovers from faults by periodically checkpointing state of all vertices between iterations

Parallel execution

- framework partitions graph and run in parallel requiring many cross-machine communications

- if graph fits in memory on single machine, the processing will probably perform better on a single machine
- GraphChi can spread graph on disk of single machine

High-Level APIs and Languages

- high-level APIs and languages related to frameworks discussed here use relational-style building blocks to express computations

The move toward declarative query languages

- framework can analyze components involved in joins and optimize accordingly
- Hive, Spark, and Flink have cost-based query optimizers that can do this
- just specifying joins in a declarative way can remove potential inefficiencies
- freedom to run arbitrary code is what distinguishes batch from MPP
- query optimizer can take advantage of column-oriented storage
  - Hive, Spark DataFrames, and Impala
    - use vectorized execution
    - iterate over data in a tight inner loop that is friendly to CPU caches
    - avoid function calls
  - Spark generates JVM bytecode
  - Impala uses LLVM to generate native code for these inner loops

Specialization for different domains

- MPP databases have traditionally served the needs of business intelligence analysts and business reporting
- batch processing is used in that domain as well as many other domains
  - statistical and numerical algorithms
  - machine learning
  - spatial algorithms such as k-nearest neighbors
- reusable implementations
  - Mahout
  - MADlib (MPP in Apache HAWQ)

Summary

- Unix tools are like MapReduce batch jobs with piping and related filesystems
- problems for distributed batch processing
  - partitioning
  - fault tolerance
  - sort-merge joins
  - broadcast hash joins

    - partitioned hash joins
- callback functions in distribute batch jobs are assumed to be stateless and have no side effects
- input date must be bounded or of a known fixed size

Ch. 11 Stream Processing

- batch processing resulted in output of derived data and works on the assumption of bounded input
- much data today is unbounded because it arrives gradually over time
- to work with this using batch processors, the data must be artificially divided over time
- stream processing -> processing data at a highly granular rate (e.g. every second) or continuously by processing every even that happens
- stream concept appears in many other places

Transmitting Event Streams

- record is more commonly known as an event in a stream processing context
    - user actions
    - machine system events
    - log append
- may be encoded as text, JSON, or binary
- event generated by producer/publisher/sender
- processed by potentially multiple consumers/subscribers/recipients
- related events are usually grouped together by a topic or stream
- can use polling by consumers but this becomes expensive when moving towards continual updates so better to use notifications in that case
- DBs traditionally used triggers which can react to change but they are limited

Messaging Systems

- common to use producer-consumer pattern
- most expand on Unix pipe or TCP socket usage to multiplex each side
- publish/subscribe differentiations
    i. handling faster production than consumption
        - drop messages
        - buffer messages in queue
            - be aware of how this will be handled in program and by system as the queue approaches and reaches capacity
        - apply backpressure/flow control
            - used by Unix pipes and TCP sockets

    ii. handling node crash or offline node
        - may require system for durability or persistence at a cost
        - if losing messages is acceptable, it will probably allow for higher throughput and lower latency
- some applications, like metrics tracking, can handle dropped messages
- message counters cannot handle dropped messages

Direct messaging from producers to consumers

- many systems use direct messaging rather than employing an intermediary
    - financial industry uses UDP multicast for stock update
    - ZeroMQ & nanomsg are brokerless and use TCP or IP multicast
    - StatsD & Brubeck use UDP for metrics
    - using services, producers can mae HTTP or RPC requests (webhooks)
- direct messaging may require additional application code to handle dropped requests

Message brokers

- message broker aka message queue -> essentially database optimized for message handling
- some only store in-memory and some persist to disk
- many allow unbounded queueing as opposed to dropping messages or backpressure
- consumers are generally asynchronous

Message brokers compared to databases

- some participate in 2PC protocols using XA or JTA
- practical differences between message brokers and DBs
    - DBs keep data indefinitely whereas brokers remove messages on successful delivery
    - brokers assume a small working set and throughput may degrade as the set increases past a point
    - DBs support secondary indexes and querying, brokers support subscribing to a queue or topic
    - DB query results in a snapshot, brokers do not support queries but send updates on data change
- standards
    - JMS
    - AMQP
- software
    - RabbitMQ
    - ActiveMQ
    - HornetQ,
    - TIBCO Enterprise Message Service

    - IBM MQ
    - Azure Service BUs
    - Google Cloud Pub/Sub

Multiple consumers

- load balancing
    - each message goes to one consumer
    - consumers share work of processing message in topic
    - broker may assign messages arbitrarily to consumers
    - useful for parallelizing expensive message processing
    - see shared subscription
- fan-out
    - each message delivered to all consumers
    - each consumer "tunes in" to broadcast messages
    - topic subscriptions/exchange bindings
- can be combined

Acknowledgements and redelivery

- problems
    - crashed consumers
    - partial processing
    - dropped messages
- acknowledgements -> client must explicitly tell the broker when it has finished processing a message so that the broker can remove it from the queue
- used for redelivery when necessary
- may lead to reordering of messages at point of delivery
    - attempts to preserve ordering will be overridden by load balancing
    - message ordering is not a problem if messages are completely independent of each other but will lead to causal issues otherwise

Partitioned Logs

- can log messages but normally they are transient and leave no permanent trace
- DBs and filesystems imply persistence
- cannot approach message processing with the same mindset as batch processing as input is not bounded and read-only
- a new consumer will only have a view of the state from the point it was added

- hybrid approach is idea behind log-based message brokers

Using logs for message storage

- append-only sequence of records on disk
- with messaging
  - a producer sends a message by appending it to the end of the log
  - consumer receives messages by reading the log sequentially
- can increase throughput by partitioning a log and making a topic correspond to a group of partitions
- a broker assigns a monotonically increasing sequence number or offset to every message
- no ordering guarantee across different partitions
- used by
  - Kafka
  - Amazon Kinesis Streams
  - Twitter's DistributedLog
- Google Cloud Pub/Sub is architecturally similar but exposes a JMS-style API rather than a log abstraction

Logs compared to traditional messaging

- trivially supports fan-out
- load balancing by assigning consumers to partition nodes
  - downsides
    - number of nodes sharing consumption of a topic is limited to number of log partitions
    - slow message in a partition delays processing of subsequent messages

Consumer offsets

- broker can only track acks for current consumer offsets which reduces overhead and increases throughput
- very similar to log sequence number in single-leader replication
  - message broker = leader
  - consumer = follower

Disk space usage

- append-only leads to running out of disk space
- log is divided into segments and old segments are occasionally discarded
- essentially implements a circular/ring buffer on disk

- log retention does not relate to throughput as throughput remains more or less bounded by disk write time

When consumers cannot keep up with producers

- slow consumers lead to messages removed from broker which means dropped messages
- can use human intervention fix a slow consumer
- slow consumers do not affect other consumers

Replaying old messages

- for log-based message brokers consuming messages is like reading from a file and is non-destructive, like AMQP and JMS-style message brokers
- logs can be used to replay messages
- this allows easier recovery from errors and bugs

Databases and Streams

- writes to databases can be seen as events that can be captured, stored, and processed
- replication log is stream of database write events describing a stream of changes over time
- state machine replication represents states over time that can be represented as a stream of events

Keeping Systems in Sync

- no single system can satisfy all data storage, querying, and processing needs
- duplicated data needs to be kept in sync
  - ETL processes
  - batch processes
  - periodic full database dumps
- dual writes -> application code explicitly writes to each of the systems when data changes
- many problems including race condition issues and mismatched failures

Change Data Capture

- most DB's replication logs are seen as an internal implementation detail and not public API
- change data capture (CDC) -> process of observing all data changes written to a database and extracting them in a form in which they can be replicated to other systems
- more recent development
- interesting approach is to capture changes as a stream as they are written

Implementing change data capture

- log consumers = derived data systems

- DB where changes are observer = leader, rest are followers
  - can use log-based message broker to transport changes to other DBs
- triggers can be used but are not robust and have performance overheads
- used by
  - LinkedIn's Databus
  - Facebook's Wormhole
  - Yahoo!'s Sherpa
  - Bottled Water for PostgreSQL (API decodes WAL)
  - Maxwell and Debezium for MySQL (using binlog)
  - Mongoriver for MongoDB (using oplog)
  - GoldenGate for Oracle
- usually asynchronous

Initial snapshot

- cannot store full set of changes indefinitely
- without full log history need to start with a consistent snapshot which will correspond to a known position within a change log

Log compaction

- either need to manually snapshot system and delete logs occasionally or use compaction
  - storage engine periodically looks for log records with the same key
  - throws away duplicates
  - keeps only recent update for each key
  - compaction and merging runs in the background
- uses tombstones and garbage collection to occasionally clean values so only the most recent value is retained
- same idea used in log-based message brokers and change data capture
- can then rebuild a system of derived data by scanning from 0
- supported by Kafka

API support for change streams

- change streams are supported as a first-class interface in
  - RethinkDB
  - Firebase
  - CouchDB
  - Meteor uses MongoDB oplog
  - VoltDB allows continuous data export as a stream

    - Kafka Connect can be used for change data capture with wide range of DBs

Event Sourcing

- developed in domain driven design community
- store changes to application state as a log of change events but applied at different level of abstraction
    - CDC uses DB mutably and application writing to DB does not need to know about CDC
    - event sourcing application logic is built on immutable events that are logged
- event sourcing record's user actions rather than result of actions
- facilitates
    - evolving applications over time
    - debugging
    - guarding against bugs
- similar to chronicle data model and fact table in star schema
- Event Store -> specialized event sourcing database

Deriving current state from the event log

- applications need to translate between log of events to system and a readable format for users to interact with (s/b deterministic)
- can replay state
- compaction is handled differently
    - generally not possible because state modifications are not tracked but rather interactions are observed/snapshotted and written
- requires some mechanism for snapshotting current state to speed restoring from a point

Commands and events

- command -> initial request from user which may still fail
- event -> state of command after validation which represents a durable and immutable action executed on application
- fact -> event at point after initial event generation
- validation of command is synchronous before creating event

State, Streams, and Immutability

- immutability is powerful
- changing state is result of mutations of state over time
- no matter how state changes, it is always result of unchangeable set of events over time
- log of all changes (changelog) represents evolution of state over time
- application state is result of integrating an event stream over time
- change stream is result of differentiating the state by time

Advantages of immutable events

- old idea, like account ledgers which are append-only
- provides auditability
- immutable events capture more information that current state
    - events that cancel each other out are still observable

Deriving several views from the same event log

- Kafka is an example of event log that is used to view same data in many different ways
- translation step from log to DB makes application feature addition and change over time easier
- often easier to run old and new systems side-by-side than performing a schema migration
- command query responsibility segregation (CQRS) -> method for gaining flexibility in software development by separating the form in which data is written from the form in which data is read
- fallacy to assume that the form in which data is written must correspond to the form in which data is read

Concurrency control

- event sourcing and change data capture is asynchronous by nature so leads to all issues discussed previously with concurrent reads and writes
    - can combine writes to read view and write log into one single atomic transaction (either within same system or across a distributed system)
- using a single event log simplifies many concurrency control issues
- for an event log that is partitioned in the same way as application state, a single-threaded log consumer needs no concurrency control for writes

Limitations of immutability

- various databases use immutable data structures or multi-version data internally to support point-in-time snapshots
- VCS for example
    - Git
    - Mercurial
    - Fossil
- easy for rarely updated or deleted data
- high rates of modification lead to
    - fragmentation
    - need for performant compaction
    - need for performant garbage collection
- custom admin modification abilities to correct errors may be required
    - rewriting history = excision of shunning

- deleting data may be difficult so often deleting data means making data impossible to retrieve

Processing Streams

- options
    1. store event data to some storage system and allow clients to query from there
    2. push events to users as notifications or real-time dashboard/monitoring
    3. pipeline input streams to output streams
- option 3 is focus and is similar to batch processing facilities but data is unbounded
    - sorting does not make sense
    - fault tolerance mechanisms must change

Uses of Stream Processing

- long used for monitoring
    - fraud detection in financial systems
    - trading systems
    - manufacturing systems monitoring
    - military and intelligence systems

Complex event processing

- complex event processing (CEP) -> approach developed in the 1990s for analyzing event streams, especially geared toward the kind of application that requires searching for certain event patterns
- often use high-level declarative language
- observed patterns of events results in emission of a complex event with details of pattern detected
- inverts system of query and data from traditional DB with queries being stored long term
- used in
    - Esper
    - IBM InfoSphere Streams
    - Apama
    - TIBCO StreamBase
    - SQLstream
- distributed stream processors
    - Samza

Stream analytics

- analytics -> aggregation of event data vs. recognition of pattern of events
- usually observed over fixed time intervals using windowing

- probabilistic algorithms used
  - Bloom filters for set membership
  - HyperLogLog for cardinality estimation
  - various percentile algorithms
  - approximation algorithms
- nothing inherently lossy in stream processing despite use of approximation algorithms
- related tools
  - Apache Storm
  - Spark Streaming
  - Flink
  - Concord
  - Samza
  - Kafka Streams
- hosted services
  - Google Cloud Dataflow
  - Azure Stream Analytics

Maintaining materialized views

- can view stream of changes to database for derived systems as a form of maintaining materialized views (same as event sourcing event log application)
- event sourcing often requires windowing from beginning of time start to current state
- Samza and Kafka Streams support this use case

Search on streams

- conventional search engines first index the documents and then run queries over the index
- searching a stream inverts process
  - the queries are stored
  - documents run past the queries, like in CEP
- possible to index queries and documents

Message passing and RPC

- message-passing systems and RPC can be thought of as stream processors due to some crossover
- Storm supports distributed RPC
  - RPC queries are spread across nodes and results are interleaved back to user
- can process streams with actor frameworks

Reasoning About Time

- many stream processing frameworks use the local system clock on the processing machine (the processing time) to determine windowing

- simple approach but breaks down in face of processing lag

Event time versus processing time

- example causes
  - queueing
  - network faults
  - contention
  - stream consumer restart
  - reprocessing of past events resulting from failure recovery
- delays can lead to unpredictable ordering of messages which may lead to bad data

Knowing when you're ready

- windowing makes it impossible to know when stream of events for current window is complete
- need to be able to handle straggler events
  - ignore straggler and handle risk of dropped events
  - publish correction and possibly retract old output

Whose clock are you using anyway?

- many devices have incorrect clocks
- can use three timestamps
  - time at which the event occurred, according to the device clock
  - time at which the event was sent to the server, according to the device clock
  - time at which the event was received by the server, according to the server clock
- can estimate offset between device clock and server clock by subtracting 2nd timestamp from third
- same problems of reasoning about time arise in many places

Types of windows

- tumbling window -> fixed length, and every event belongs to exactly one window
  - implement a fixed time interval tumbling window by taking each event timestamp and rounding it down to the nearest fixed time interval to determine the window that it belongs to
- hopping window -> fixed length, but allows windows to overlap in order to provide some smoothing
  - implement by first calculating fixed time interval tumbling windows, and then aggregating over several adjacent windows
- sliding window -> capture all events that occur within some interval of each other
  - implement by keeping a buffer of events sorted by time and removing old events when they expire from the window

- session window -> no fixed duration; defined by grouping together all events for the same user that occur closely together in time; window ends when the user has been inactive for some time

Stream Joins

- same need for joins on streams of data
- more challenging

Stream-stream join

- two separate streams of data with information about a related resource (e.g. URL, UUID, etc) are combined to retrieve related information
- stream processor must maintain state and when either event arrives, the stream processor checks the other event stream for a matching event and emit event with info about matching events or failure to find matching events

Stream-table join (stream enrichment)

- basically adds table information attached to a related (foreign key-like) piece of information in an incoming stream event
  - can query DB
  - can load DB copy locally into stream processor
- difference from stream-stream join ->

Table-table join (materialized view maintenance)

- track one-to-many table relationships in real-time by caching incoming data for the "many" portion and associate it with the "one" record
  - incoming data to a record within the "many" portion is cached with each other record that has a link to that record (and deleted/modified as necessary)
- requires multiplexing streams of events to both the record persistence layer and the caching layer

Time-dependence of joins

- commonalities
  - require the stream processor to maintain some state based on one join input, and query that state on messages from the other join input
- ordering of events is important
- if ordering is undetermined across streams, the ordering is nondeterministic

- slowly changing dimension (SCD) -> use UID for particular version of nondeterministically joined records

Fault Tolerance

- need batch processing-like exactly-once semantics
- less straightforward in stream processing
    - cannot wait until a task is finished before exposing output because stream is infinite

Microbatching and checkpointing

- can break the stream into small blocks, and treat each block like a miniature batch process
- leads to performance compromise when choosing batch size
- implicitly implies tumbling window at batch size
- used in Spark Streaming
- can periodically generate rolling checkpoints of state and write them to durable storage
- used in Flink

Atomic commit revisited

- need to ensure processing output of event takes effect if and only if the processing is successful
- brings back atomic commit issue
- implementations keep results of transactions across heterogeneous technologies internally by managing both state changes and messaging within the stream processing framework rather than attempting to sync transactions across them
- used in
    - Google Cloud Dataflow
    - VoltDB
- plans to add to Kafka

Idempotence

- idempotent (operation) -> operation that can be performed multiple times but result is identical to if the operation had been performed only once
- can make operations idempotent by storing metadata about operations using UUIDs for operations
- Storm Trident
- several related assumptions
    - restarting a failed task must replay the same messages in the same order (a log-based message broker does this)
    - processing must be deterministic, and no other node may concurrently update the same value

Rebuilding state after a failure

- stream processes that require state must ensure that this state can be recovered after a failure
  - any windowed aggregations (such as counters, averages, and histograms)
  - any tables and indexes used for joins
  - etc
- can keep state local to a stream processor and replicate periodically
- Flink periodically captures snapshots of operator state and writes them to durable storage such as HDFS
- Samza and Kafka Streams replicate state changes by sending them to a dedicated Kafka topic with log compaction, similar to change data capture
- VoltDB replicates state by redundantly processing each input message on several nodes
- can sometimes avoid replicated state because state can be rebuilt from input streams
- all require trade-offs on performance characteristics

Summary

- message brokers and event logs serve as the streaming equivalent of a filesystem
- two main types
  - AMQP/JMS-style message broker
  - Log-based message broker
- sources
  - user activity events
  - sensors providing periodic readings
  - data feeds (e.g., market data in finance)
- purposes for stream processing
  - searching for event patterns (complex event processing)
  - computing windowed aggregations (stream analytics)
  - keeping derived data systems up to date (materialized views)
- types of joins
  - stream-stream joins
  - stream-table joins
  - table-table joins
- fault tolerance
- exactly once semantics

Additional Notes

- https://en.wikipedia.org/wiki/Join_(SQL)