Intro

Everything as a service

Infrastructure as a Service Rent cycles on machines (Amazon EC2)

Platform as a Service Nice API, they take care of maintenance and upgrades (Google App Engine)

Software as a Service Run the software for me (GMail, Salesforce, etc.)

Big Picture

- Architecture has tiers:
  - Tier one handles requests
  - Tier two is caching etc.
- Inner services (DB and index) are shielded from online load
- Replicate data within our cache to spread loads and provide fault-tolerance
- Not everything needs to be replicated

Second Tier Examples

- Memcached (in-memory key-value store)
- Distributed hash tables
- DynamoDB (Amazon service)
- BigTable (Google service)

Read vs Write

- Reading many values takes as long as the slowest read
- Writing many values is the same…

CAP Theorem

- "You can have just two from Consistency, Availability, and Partition Tolerance"
- Usually consistency is lowered so the other two are "true"
- Lower consistency can be okay if data non-essential

Programming Models

- Shared memory (pthreads)
- Message passing (MPI)

Design Patterns

- Master-Slave (Farm)
- Producer-Consumer
- Shared Work Queues

MapReduce

- Beyond von Neumann architecture
- Hiding system level details from developers
- Separating the wat from how

Big Ideas

- Scaling out not up
- Processing near data (not wasting bandwidth)
- Process data sequentially
- Add more machines => more scalable (data centre == computer)

Runtime

- Handles scheduling, data distribution, synchronisation, and errors
- Specify two functions `map -> <k, v>` and `reduce -> <k, [v1, v2, ...]>`
- All keys with same key go to same reducer
- Programmers can also specify `partition` and `combine` (mini-reducers in memory)

Implementations

- MapReduce (proprietary by Google)
- Hadoop (open-source by Apache)

Distributed Filesystems

- Move data to workers (not enough RAM, minimise traffic)
- High component failure rates
- Commodity hardware (cheap)
- "Small" number of big files
- Files are mostly appended to
- Sequential reads > Random reads

Implementations

- GFS (Google MapReduce) and HDFS (Hadoop)
- Files stored as chunks (64 MB)
- Reliability via replication
- One master coordinates access
- No data caching, simple API

Architecture

- There is a master (Google: GFS master, Hadoop: namenode)
- And there are "workers" (Google: GFS chunkservers, Hadoop: datanodes)

Namenode

- Hold file structure, metadata, permissions and file-to-chunk mapping
- Directs clients to specific datanodes for reads and writes
- Maintains health (heartbeats and such)

Hadoop

- Mapper
  - void setup(Mapper.Context context) - Called once at the beginning of the task
  - void map(K key, V value, Mapper.Context context) - Called once for each key/value pair in the input split
  - void cleanup(Mapper.Context context) Called once at the end of the task
- Reducer/Combiner
  - void setup(Reducer.Context context) - Called once at the start of the task
  - void reduce(K key, Iterable values, Reducer.Context context) Called once for each key
  - void cleanup(Reducer.Context context) Called once at the end of the task
- Partitioner
  - int getPartition(K key, V value, int numPartitions) -Get the partition number given total number of partitions
- Job
  - Represents a packaged Hadoop job for submission to cluster
  - Need to specify input and output paths
  - Need to specify input and output formats
  - Need to specify mapper, reducer, combiner, partitioner classes
  - Need to specify intermediate/final key/value classes
  - Need to specify number of reducers (but not mappers, why?)

Some Useful Interfaces

- `Writable` - serialisation protocol (keys and values)

- `WritableComparable` - Defines sort order (all keys)
- `IntWritable`, `LongWritable`, … - Specific types

Complex Data Types

- Store them in text => regex them out (hacky) or JSON
- Implement said interfaces

Hadoop Architecture

- Master
  - Namenode: master node for HDFS
  - Jobtracker: gets job submissions and distributes them
- Worker
  - Tasktracker: contains task slots (assignments)
  - Datanode: contains HDFS file blocks
- Client creates a job, configures it and sends to jobtracker
- Job is divided into tasks and executed on tasktrackers which poll for them in a shared location

Shuffle and Sort

- Mapper
  - Outputs are buffered in a circular buffer
  - When buffer hits threshold, spill the contents on to disk
  - Contents are merged into a single file (within each partition), combiners run
- Reducer
  - Map outputs are copied over to reducer worker
  - Multi-pass sort (in memory and on disk), combiners run
  - Final merge pass goes into the reducer

MapReduce Algorithms

Scaling up vs out

- small cluster of SMP machines vs large cluster of commodity hardware
- intra-node latencies ~ 100ns
- inter-node latencies ~ 100micros

Optimising Computation

- Sort order of intermediate keys
- Control which reducer processes which keys
- Preserve state in mappers and reducers (local aggregation) => lower communication

Combiner Design

- Combiners and reducers have same method signature
- Combiners and mappers should write same value types

Large Counting Problems

- We want to emit bigrams
- Let mappers create partial counts and reducers aggregate them
- Two designs:
  - Pairs: Emit ((a, b), 1) for every pair of co-occuring words
    * Easy to implement but lot of shuffling and sorting
  - Stripes: Emit (a, [(b, 1), (c, 1), …])
    * Far less sorting and shuffling
    * Better use of combiners
    * Limited in memory size

Replication

- To continue working even if a fault occurs

- To improve performance:
  - Load sharing
  - Nearer location for data access
- Remote sites working when local fail
- Protection against data correction

Requirements

- Transparency: clients see logical objects not physical, each access return single object
- Consistency: All replicas are consistent for some condition

Synchronisation Models

- Non-explicit models:
  - Strict: All processes must see shared accesses in absolute time order
  - Linearisability: All processes must see shared accesses in the same order; accesses are ordered according to global timestamp
  - Sequential: All processes must see shared accesses in the same order; timestamps don't matter
  - Causal: All processes must see causally-related shared accesses in the same order
  - Fifo: All processes see writes from each other in order they were used; different processes may not always be seen in that order
- Explicit models:
  - Weak: Shared data is consistent only after synchronisation
  - Release: Shared data is made consistent when a critical region is exited
  - Entry: Shared data pertaining to a critical region is made consistent when a critical region is entered

Eventual Consistency

- Sacrifice global consistency, keep local consistency
- Read access => no problem
- Infrequent writes => ok as long as same client same replica

Fault Tolerance

- Availability: System is ready to be used immediately
- Reliability: System is always up
- Safety: Faillures are never catastrophic
- Maintainability: All failures can be fixed without noticing

Failure Models

- Crash: A node halts, but is working correctly before
- Omission: A node fails to respond to requests
- Timing: A node's response lies outside specified time interval
- Response: A node's response is incorrect
- Arbitrary: A server produces arbitrary responses at arbitrary times

Solutions

- Information redundancy: Error detection and recovery (hardware level)
- Temporal redundancy: start operation and if it does not complete start it again (transactions required)
- Physical redundancy: add extra software and hardware, have multiple instances

Issues associated with fault tolerance

- Process resilience: replicate processes into groups; agreement within a group?

- Reliable client/server communication: masking crashes and omissions
- Reliable group communication: processes coming/leaving the group
- Distributed commit: performed by all members or none at all
- Recovery strategies: recovering from an error

Byzantine fault tolerance

- Solution only if number of messages is more than 3 times the number of messages that were lost.

Recovery

- Backward recovery (more common): return system to some previous correct state
  - Continually take snapshots of the system
  - When to delete snapshots?
- Forward recovery: bring system to correct state and continue
  - Account for all errors upfront => have strategy

Virtualisation

- Technique to separate hardware, OS, and applications

CPU and Architecture Virtualisation

- User ISA and system ISA
- ISA virtualisation, instruction interpretation, trap and emulate, binary translation, and hybrids
- Virtualisation needs to translate guest state into host state as well as transformations that advance state

User ISA

- Application state: Virtual memory, registers

System ISA

- The inner rings: 0 (and maybe 1)
- Control registers of the CPU
- System clock
- Memory management unit: page table, TLB
- Device I/O
- Virtualisation monitor (hypervisor)
  - Monitor supervises the guest and virtualises calls to the System ISA
  - Whenever the guest wants to access System API, the monitor takes over
  - Shares address space with address space it virtualises
  - It handles page faults

Virtualisation Types

- Trap and emulate: execute normal instructions, trap privileged instructions and emulate running them
  - Could be ran in host kernel/extension level
- Binary translation:
  - Compile programs to intermediate representation (Java, llvm)
  - Transform instructions on the fly
  - Separate model for host and guest accesses
- Hybrid models: kernel is binary translated, user code is trapped and emulated

Further Virtualisation

- We can emulate CPU & memory but I/O devices as well
  - Uniformity: Remote hard drive or RAID
  - Isolation: Devices operate as if they were alone

- Performance: Lower level entities optimise I/O path
- Multiplexing: Parallelise processes (e.g. replication)
- System Evolution: Connecting new drive while system is alive

- Three main techniques:
  - Direct Access
    - No changes to guest but specialised hardware for host
    - Hardware interface needs to be visible to guest
  - Device Emulation
    - Drivers are in monitor or host, no special hardware
  - Paravirtualisation
    - Expose monitor and allow guest to make monitor calls
    - Implement guest-specific drivers (one each)

NoSQL

- State unlikely to fit on a single machine, must be distributed
- Three core ideas:
  - Partitioning (sharding): Scalability and latency
  - Replication: Availability and throughput
  - Caching: latency

RDBMS

- Relational mdoel
- Transactional semantics: ACID
- Multiple machines => Distributed protocol: Two-phase commit
  - Coordinator sends prepare, subordinates reply with OK or no
  - If one subordinate replies no, coordinator sends aborts
  - If all OK then coordinator sends commit
  - All subordinates reply with ack
  - If onen subordinate doesn't reply, coordinator sends rollback
- 2PC needs a write-ahead-log and persistent storage at every node
- 2PC is blocking, slow, and if coordinator dies => problem

NoSQL

- Scale simple operations horizontally
- Weaker model than ACID
- Flexible schemas and replicated over many servers

Key-value Stores

- Keys are usually primitives (hashable)
- Values can also be complex
- API: Get, Put (usually atomic)
- Can be persistent or non-persistent

Dealing with Scale

- Partition key space across many machines
- Store key `k` as follows: `hash(k) mod n`
- We need to also hash the machines so we know who to contact

Chord

- Distributed hash table (DHT) arranged in a ring
- Every machine has a successor and a predecessor $O(n)$
- Every machine can have a *finger table* (+2, +4, ...) $O(logn)$

Google BigTable

- Distributed, sparse, persistent, multi-dimensional sorted map
- Map indexed by row, column and a timestamp => unique key
- Support lookups, inserts, deletes
- Single row transactions only
- Rows are maintained in sorted order
- Row ranges grouped into *tablets*
- Columns grouped into families with `family:qualifier` as id
- Uses GFS, Chubby (master file), and SSTables

SSTable

- Basic building block of Bigtable
- Group of blocks + index stored in GFS (can be mapped to memory)
- Supports key lookup or iteration

Tablets

- Partitioned range or rows
- Built from multiple SSTables
- One SSTable can be in more tablets

Architecture

- One master server, many tablet servers
- Master assigns tablets to servers
  - Detects addition and expiration of servers
  - Balances tablet server load
  - Handles garbage collection and schema evolution
  - Table merging, creation, deletion
- Tablet servers manage a set of tablets and handles reads and writes
  - Each tablet belongs to one server at a time
  - Table splitting when become too big

CAP Tradeoffs

- CA = consistency and availability (parallel databases with 2PC)
- AP = availability and partition tolerance (DNS, web caching)
- Replication possibilities (eventual consistency)
  - Update sent to all replicas
  - Update sent to master (sync & async)
  - Update sent to one replica
- Partitioning
  - Single record: easy
  - Arbitrary transactions: 2PC
  - Entity groups: group of entities that share affinity
  - Provide transaction support within entity groups
- Caching
  - wat

BASE vs ACID

ACID (model)

- model for correct database behaviour
- **Atomicity**: Either it all succeeds or all fails
- **Consistency**: Transaction on a correct database leaves it in correct state
- **Isolation**: Looks as if each transaction runs by itself
- **Durability**: Once commmited, cannot be rolled back
- Developers do not worry about leaving in partial state
- Transactions cannot glimpse partially completed state of other one
- ACID is costly:

  - Either use locks (contention)
  - Snapshot mechanisms keep a history of each data item
- Two types of transactions:
  - Embarassingly easy ones
  - Conflict-prone ones (bad scalability)

Serial and Serialisable

- **Serial**: At most one transaction at a time, commit or abort before next
- **Serialisability**: Ilusion of serial execution (identical outcome)

BASE (methodology)

- "Basically Available Soft-state Services with Eventual Consistency"
- Transactions that scale well in cloud systems
- **Basically Available**: Rapid responses even when some replicas can't be contacted
- **Soft-state Service**: Runs in first tier, cannot store data, passes it along
- **Eventual consistency**: Could use cached data, make guesses, skip locks

How BASE is Used

- Use transactions but remove Begin/Commit
- Fragment into steps that can be done in parallel
- Each step can be associated with a single event that triggers that step (multicast)
- Leader stores events in a message queue
- Mask the asynchronous side effects to the user

Amazon Dynamo

- Key-value storage based on DHT (like Chord)
- Dynamo is in tier 2 for every data centre (everything for a given user)
- Dynamo was impacted if a component was slow or overloaded
- Node K wants to use finger table to route to $K+2^i$ but gets no acknowledgement
- Dynamo tries again with $K+2^{i-1}$
- If a target node doesn't respond, do Get/Put on the next node that responds
- On misrouting and miss-storing, confusing things can happen but eventually repaired

BitTorrent

- Large-scale P2P network
- Incentive-based: the more you give the more you get
- Cannot use IP Multicast
  - Not supported by many ISPs

End-host Based Multicast

- Multiple uploaders
- Lots of nodes want to download => use them for upload
- Application-level multicast
- Single tree:
  - Node dying or being slow affects whole tree
  - Leaf nodes do no work

BitTorrent

- File split into smaller pieces
- Nodes request desired pieces from neighbours
- Not downloaded in sequential order
- BitTorrent does not support streaming

Swarm

- Set of peers all downloading the same file

- Organised as a random mesh
- Each node knows pieces downloaded by other nodes in swarm

Implementation

- Tracker keeps track of all peers downloading the file
- Torrent file contains
  - URL of tracker
  - Piece length
  - SHA-1 hashes of each piece
  - Filenames

Piece Chossing Strategy

- **Rarest First Piece**: Look at all pieces at all peers and request piece that's owned by fewest peers
  - Increases diversity, throughput, likelihood of completion
- **Random First Piece**: Request random piece
- **End Game Mode**: When requests sent for all sub-pieces, send requests to all peers

Incentive to Upload

- Peer A said to choke peer B if decided not to upload to B
- Peer A unchokes at most 4 interested peers at any time
  - Three with largest upload rates to A
  - One randomly chosen one
- A peer is snubbed when each of its peers chokes it

BitTorrent Advantages

- Pull-based transfer
- Slow nodes do not slow down fast nodes
- Allows upload from hosts with parts of a file
- Rewards fastest uploaders
- Does not do search

Trackerless BitTorrent

- Using a DHT
- Ran by a normal end host (not a webserver)

Data Warehousing

- Two types of database workloads
  - **OLTP (online transaction processing)**
    * e-commerce, banking (user facing, concurrent)
    * small set of standard transactional queries
  - **OLAP (online analytical processing)**
    * business intelligence, data mining (back-end)
    * complex analytical queries, often ad hoc
- Need to separate two workloads into separate databases
- **OLTP** is (ETL) extracted, transformed and loaded into **OLAP**
- OLAP Cubes: Lots of group bys and aggregations

ETL Bottleneck

- Usually done overnight: what if takes longer?
- Use Hadoop to do this => better performance/scalability

Secondary Sorting

- Use part of the value as a key => Hadoop will sort it out

Projection in MapReduce

- Mappers choose only appropriate attributes

Selection in MapReduce

- Mappers choose only tuples that satisfy criteria

GroupBy and Aggregation

- Map over tuples emit value and group by key
- Reducer computes the aggregate value

Joins in MapReduce

- Reduce-side Join
  - Map over tuples and emit join key as secondary key
  - Perform join in reducer
  - Need to ensure that `R` comes before `S`
  - In Many-to-many, make sure we can hold them in memory
- Map-side Join
  - Assume the tables are sorted by join key
  - Map over one dataset, read from other partition
- In-memory Join
  - Load one dataset into memory, iterate over other dataset
  - Distribute `R` to all nodes
  - Map over `S`, look up join key in `R`
  - **Striped variant**
    * Divide `R` into partitions, join each with `S`, then merge
  - **Memcached variant**
    * Load R into memcached
    * Instead of in-memory lookup do in-memcached lookup
- In-memory (memory) > map-side join (sorted, partitioning) > reduce-side join

High Level Languages

- Hive
  - Data warehousing in Hadoop
  - Uses HQL, variant of SQL, stored in HDFS
- Pig
  - Scripts written in Pig Latin
  - Focused on data transformations

MapReduce Disadvantages

- Misses Schemas, separation from application
- Poor implementation (brute force)
- Misses indexing, transactions, etc.
- Not compatible with DBMS tools
- Maturity, not capability

ETL Redux

- Put Hadoop between OLTP and OLAP
- Maybe load first then extract and then transform

RDBMS vs. MapReduce

- RDBMS
  - Multi-purpose: analysis and transactions; batch and interactive
  - Integrity via ACID transactions
  - Lots of tools and SQL
  - Failures infrequent
- MapReduce
  - Large clusters, fault tolerant
  - Data is accessed in “native format”
  - Supports many query languages
  - Failures are common

Data Streams

- Large data volume, arriving at a high rate, continuous, ordered sequence of items

- Event detection, reaction, and analytics
- Timestamping
  - Explicit: date/time
  - Implicit: Given when it arrives
- Time Representation
  - Physical: date/time
  - Logical: Order integer

DBMS VS DSMS

- DSMS: at multiple observation points, voluminous streams in, reduced streams out
  - Model: transient relations
  - Relation: tuple sequence
  - Data update: appends
  - Query: persistent
  - Query answer: approximate
  - Query evaluation: one pass
  - Query plan: adaptive
- DBMS: outputs of DSMS can be treated as input to DBMS
  - Model: persistent relations
  - Relation: tuple set/bag
  - Data update: modifications
  - Query: transient
  - Query answer: exact
  - Query evaluation: arbitrary
  - Query plan: fixed

Windows

- Mechanism for extracting a finite relation from infinite stream
- Restricting scope by:
  - Window based on ordering attributes (time)
    - Sliding window: overlap windows
    - Tumbling window: "partition" windows
  - Window based in item counts (take first 100, etc.)
    - Could be sliding or tumbling
    - Problematic for non-unique timestamps => could be non-deterministic
    - Fluctuating input rates could cause problems
    - Can be converted to time-based if we know rate
  - Window based on explicit markers (e.g. punctuations)
    - Application inserts "end-of-processing" markers
    - Variable length windows
    - Windows could become too small or large

Data Stream Mining

- Mining query streams: Google wants to know what's popular
- Mining click streams: Yahoo wants to know which pages are popular

Frequent Pattern Mining

- Finding patterns that occur more often than a threshold
  - Patterns refer to items, item-sets, or sequences
  - Threshold refers to percentage of pattern occurrences to total number of transactions
- Finding association rules: A->B (item sequence pattern)
- **Confidence**: Measure that says B exists, given A exists
- Cannot afford multiple passes
  - Minimised requirements for memory
  - Trade off between storage, complexity and accuracy

Lossy Counting

- Deterministic technique
- Stream is divided into buckets, each one is given a label starting from 1
- Two parameters: Support (s), error (ϵ)
- Data structure tracking item, frequency, and maximum possible error
- New entry:
  - Increase frequency if exists
  - Add new entry with frequency 1 and error $1 - bucketlabel$
  - The error is the maximum number of times the item could have occurred in previous buckets
  - An entry gets deleted if its $frequency + error < bucketlabel$
- $Frequencyerror \leq Numberofwindows(\epsilon N)$
- Usually set $\epsilon = 10\% of supports$
- Output is elements with counter values exceeding $sN - \epsilon N$
- Frequencies are underestimated by at most ϵN
- No false negatives
- False positives have a frequency at least $sN - \epsilon N$

Sticky Sampling

- Probabilistic technique
- Three parameters: Support (s), error (ϵ), probability of failure (δ)
- Data structure tracking item, frequency
- Sampling rate decreases with increase in number of processed data elements
- New entry:
  - Increase frequency if exists
  - If not, sample item with current sampling rate, if selected add new entry, otherwise ignore
- With every change in sampling rate toss a coin for each entry
  - Decreasing the frequency for each unsuccessful toss
  - If frequency down to zero, release it
- Sampling rate: $2/N\epsilon \times \log(1/s\delta)$
- Same guarantees but non-deterministic
- Number of counters is independent of N

Storm & Low Latency Processing

- Distributed system, results as quickly as possible
- Algorithmic trading, event detection

Beyond MapReduce

- We want schemas => no parsing, auxiliary structures
- Relational algorithms have been optimised for underlying system

Apache Thrift

- Data Definition Language with numerous language bindings
- Provide RPC mechanisms for services
- Compact binary encoding of typed structs

Storage Layout

- **Row store**: row after row sequentially
  - Easy to modify a record (indexing)
  - Might read unnecessary data when processing
- **Column store**: column after column sequentially
  - Only read necessary data when processing
  - Multiple writes when writing a row
  - Read efficiency: if only few columns, no need to drag rest of values

- Better compression: Repeated values appear more frequently in a column than "repeated rows"
- Vectorised processing: CPU architecture-level support
- Operate directly on compressed data

Pig

- Sequence of statements manipulating relations
- Data model: atoms, tuples, bags, maps, json
- Pig user-defined functions (UDFs) make it extensible

HadoopDB

- Parallel databases focused on performance
- Hadoop focused on scalability, flexibility, and fault tolerance
- Co-locate a RDBMS on every slave node
- Push operations into the DB

HaLoop

- MapReduce under-performs in iterative algorithms
- Java verbosity, long startup time, data shuffling
- Loop-aware scheduling
- Caching reducer input and reducer output

Pregel

- Based on Bulk Synchronous Parallel
- Computational units encoded in directed graph
- Computation proceeds in series of supersteps
- Each vertex, at each superstep:
  - Receives messages directed at it from previous superstep
  - Executes a user-defined function
  - Emits messages to other vertices (next superstep)
- Terminates
  - A vertex can choose to deactivate
  - Woken up if new messages received
  - Computation halts when all vertices inactive
- Master-Slave architecture
  - Vertices are hash-partitioned and assigned to workers
  - Everything happens in memory
- Processing cycle
  - Master tells all workers to advance a single superstep
  - Worker delivers messages from previous superstep, executing vertex computation
  - Messages sent asynchronously
  - Worker notifies master of number of active vertices
- Fault tolerant

YARN: Hadoop 2.0

- Yet-Another-Resource-Negotiator
- Provides API to develop any generic distributed application
- Handles scheduling and resource request
- Hadoop is one such application on top of YARN