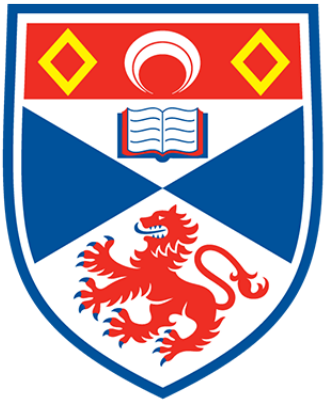

University of
St Andrews

CS5052 Data Intensive Systems

Revision notes

May 7, 2019

Lecturer:
Adam Barker

Submitted By:
140011146

Contents

1 Parallel Processing

1.1 MapReduce

MapReduce: Simplified Data Processing on Large Clusters `https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf`

1.1.1 Key points

Traditional systems before MapReduce tended to use large centralised servers for storing and retrieving data. Furthermore, various types of computation on large data required special-purpose implementations, including how to distribute the computation in parallel, which cost a lot of engineering effort.

Problems:

1. Large amount of data
2. Parallelise the computation, distribute the data and handle failures
3. Centralised systems created too much of a bottleneck when processing multiple files simultaneously

MapReduce solves these problems by providing a programming interface that handles the complexity, distribution and fault tolerance while allowing an engineering to write in a generic manner with the **map** and **reduce** primitives.

Related work
Systems before MapReduuce such as the BSP pattern or some MPI primitives were either implemented on a smaller scale or parallelism or do not handle failures automatically.

Since the MapReduce paper was released, an open source version - Hadoop - has been used by companies around the world for big data analytics. It has since grown into a large framework for processing *and* storing massive datasets, using MapReduce as the programming model to process the data stored in Hadoop.

Solution

- Interface that enables automatic parallelisation and distribution of large-scale computations.
- Achieve high performance on large clusters of commodity hardware
- Frequent disk I/O and data replication limits its usage in iterative algorithm and interactive data queries

1.1.2 Specifics/details

Programming model
MapReduce takes a set of **input key/value pairs**, and produces a set of **output key/value pairs**. A user expresses the computation as two functions: **Map** and **Reduce**.

- Map - Takes an input pair and produces a set of **intermediate** key/value pairs. The library then groups together intermediate values associated with the same intermediate key and passes them to the reduce function
- Reduce - Accepts an intermediate key and a set of values for that key. It merges together the values to form a smaller set of values (typically zero or one output value per Reduce invocation).

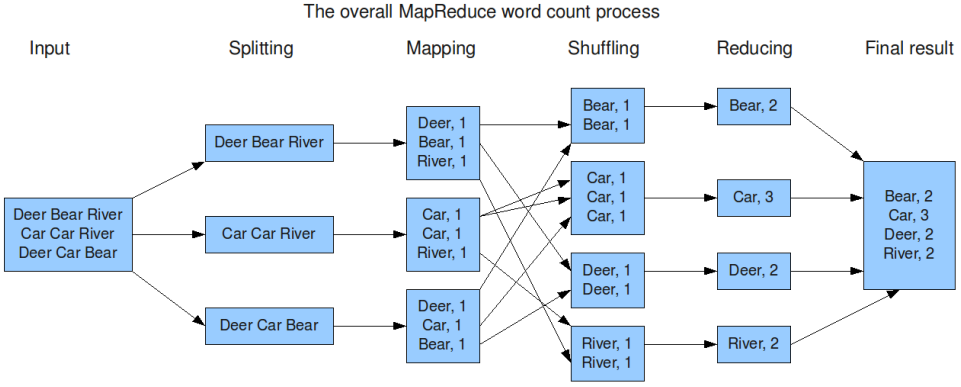

Figure 1: MapReduce example counting occurrences of words.

Architectural overview and components MapReduce follows a sequence with master and worker nodes to achieve its operation. The master must make $O(M + R)$ scheduling decisions and keep $O(M + R)$ state in memory.

1. Data is separated into M splits, decided by the user
2. A machine is chosen as the **Master**. The master allocates the M splits to selected machines (Map workers)

3. Map workers periodically write their intermediate pairs to R number of regions on local disk and return their location to the master
4. The master tells Reduce workers the location, which can start to read R splits while the Map workers are still working
5. When all intermediate data is read by the Reduce workers, it is sorted by Key and the Reduce functions are run
6. The output of the Reduce machines are appended to the output file

To handle fault tolerance, the master periodically pings every worker. After some time, if no reply is received, the master marks the worker as failed. Assigned work is reset and picked up by other workers, even if the work is completed because it is written to local disk.

MapReduce also uses a **backup** mechanic to deal with stragglers. It assigns another machine on the in-progress tasks as a backup in case the original machine takes too long to finish the work.

1.1.3 Problems and evaluation

Advantages

- Simple and easy to use via the Map and Reduce interface
- Flexible - No dependency on data model and schema
- Independent of storage - Works independent of underlying storage layers
- High fault tolerance
- High scalability

Disadvantages

- No high-level language - Does not support high-level languages like SQL in databases or any query optimisation techniques
- No schema and index
- A single fixed dataflow - More complex algorithms which cannot fit into the Map and Reduce functions are hard to implement
- Low efficiency - Not optimised for I/O efficiency. Transition to the next stage cannot be made until all tasks of the current stage are finished

1.2 Resilient Distributed Datasets (RDD)

Resilient Distributed Datasets is a distributed memory abstraction that lets programmers perform in-memory computations on large clusters *with* fault tolerance. One major disadvantage with MapReduce is the need to write to local disk, which is significantly slower than keeping data in memory. However, fault tolerance is difficult when keeping all the data in memory.

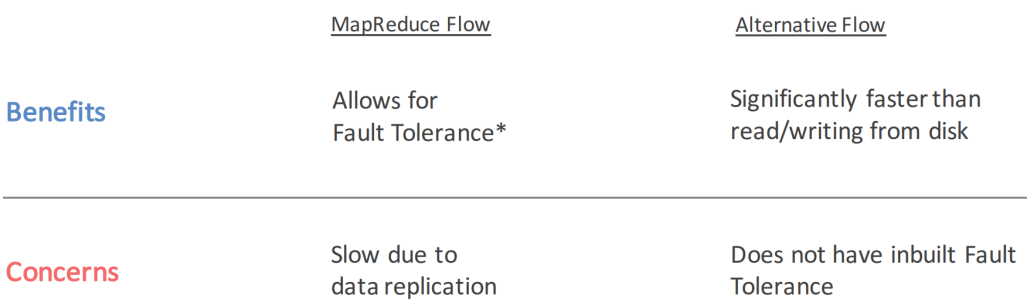

Figure 2: Benefits and concerns with MapReduce.

1.2.1 Key points

Most frameworks at this time lacked abstractions for leveraging distributed memory, which makes them inefficient for applications that *reuse* intermediate results across multiple computations. Data reuse is common in iterative applications such as machine learning.

The key to RDD is to enable efficient data reuse in a broad range of applications and to be fault-tolerant, allowing users to

- Persist intermediate results
- Control partitioning to optimise data placement
- Allow a rich set of operators for manipulation

RDD provides the following features:

- **Resilient** - Fault-tolerance using the RDD lineage graph to recompute missing or damaged partitions due to node failures
- **Distributed** - Data lived on multiple nodes in a cluster
- **Dataset** - Collection of partitioned data with primitive values, tuples or other objects

It is also advertised with the following traits:

- **In-memory** - Data is stored in memory as much and as long as possible
- **Immutable** - Data is read-only and does not change once created, only being transformed into new RDDs
- **Lazy evaluation** - The data is not available or transformed until an action is executed that triggers the execution
- **Cacheable** - Data can be held in a persistent storage, either in memory (by default) or on disk
- **Parallel** - Data is processed in parallel
- **Typed** - RDD records have types
- **Partitioned** - Records are partitioned (split into logical partitions) and distributed across nodes in a cluster
- **Location-stickiness** - Placement preferences can be manually defined to compute partitions as close to the records as possible.

RDD solves the challenge of efficient fault tolerance by logging the data transformation used to build the dataset, rather than the data itself. This is called the **lineage** and it provides enough data to recover a lost partition, by showing how it was derived.

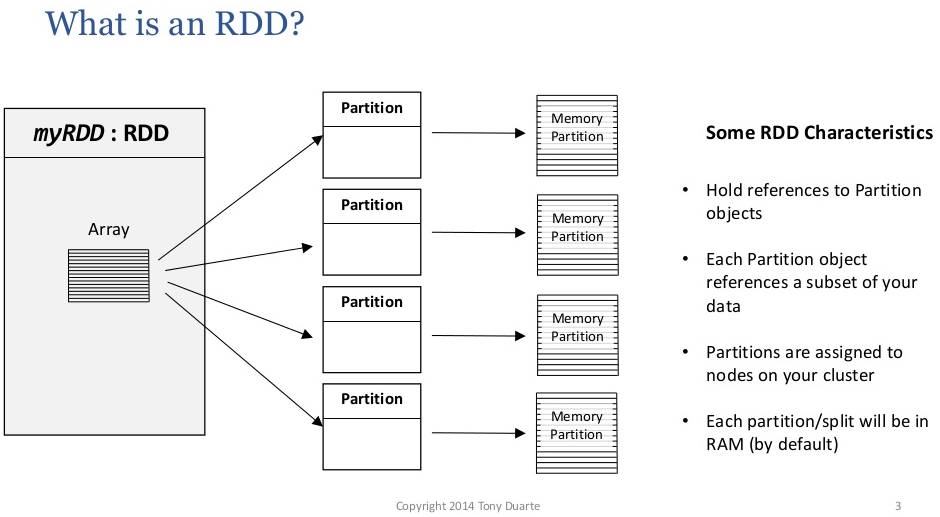

Figure 3: RDD partitions.

1.2.2 Specifics/details

Partitions are RDDs way of dealing with parallelism and distributing the data. Users manually control the number of partitions and support two kinds of operations:

- **transformations** - lazy operations that return another RDD
- **actions** - operations that trigger computation and return values

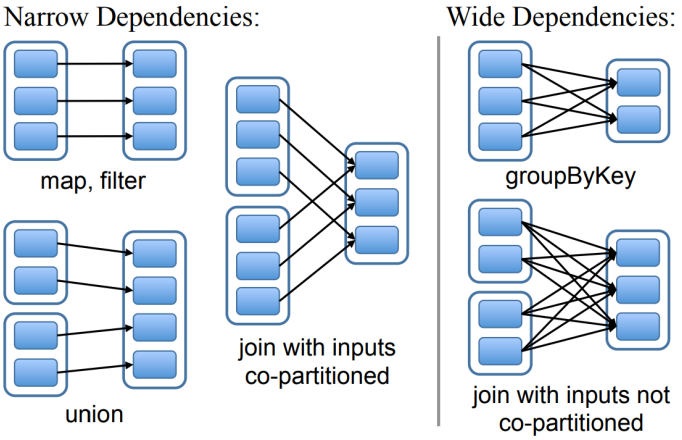

Figure 4: Examples of narrow and wide dependencies.

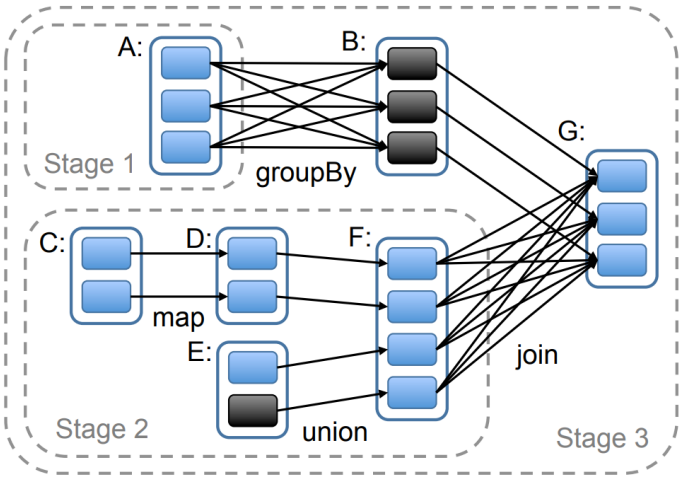

Figure 5: Example spark job in stages. Each box is a separate RDD

1.2.3 Problems and evaluation

Advantages

- Not affected by network bandwidth
- Straightforward and simple programming interface
- Allows manual control

Disadvantages

- Not applicable to asynchronous applications
- Slow for non-JVM languages

1.3 Google File System (GFS)

The Google File System is a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware and delivers good performance to a large number of clients.

1.3.1 Key points

The key ideas behind GFS are:

- Constant monitoring and error detection
- Ability to withstand massive file sizes
- Performance optimisation through appending
- Allow multiple clients to concurrently append to a file without extra synchronisation

GFS was designed to meet the growing demands of Google's processing needs. It is designed with the following aspects in mind:

- Component failures are the norm rather than the exception
- Files are huge by traditional standards
- Most files are mutated by appending rather than overwriting existing data

The problem arises from the need to efficiently store and access large files in a distributed manner, so GFS optimises for the particular use cases that Google needs, in particular the efficient appending.

Assumptions

- Modest amount of large files (multi-GB files should be managed efficiently), which means not optimised for smaller files
- Small writes at an arbitrary offset are not optimised
- Essential to have atomicity with minimal synchronisation when appending
- High bandwidth is more important than low latency

1.3.2 Specifics/details

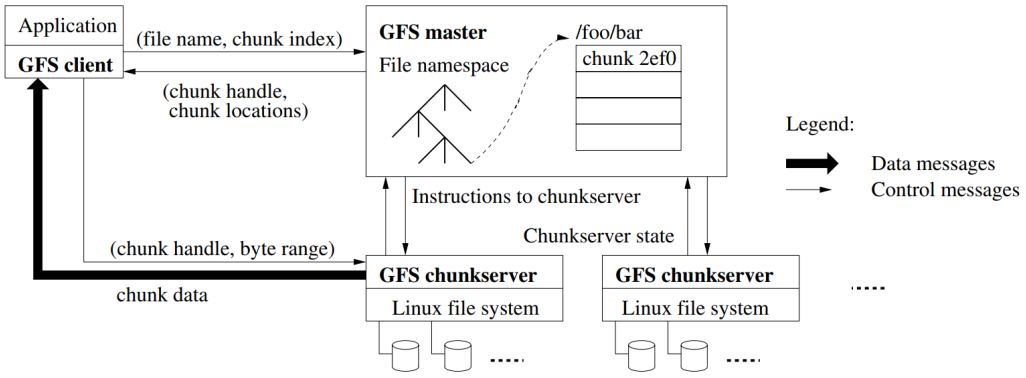

Figure 6: GFS architecture

The GFS architecture consists of a single **master** with multiple **chunkservers**, accessed by multiple clients.

- Files are divided into fixed-sized **chunks**
- Chunkservers store chunks on local disks
- Chunks are replicated on multiple chunkservers for reliability

The master maintains the file system metadata, which includes the namespace, access control information, mapping from files to chunks and the location of chunks. A **heartbeat** is used to communicate between the master and chunkserver to collect state and instructions.

Chunk size
The chunk size is fixed and can be changed, though it defaults to 64MB. A large chunk size offers several advantages:

- Reduces a clients' need to interact with the master because reads and writes to the same chunk require only one request to the master
- Reduces network overhead as more operations are performed on the same chunk
- Reduces the size of the metadata sorted on the master (less chunks = less metadata)

Fault tolerance

- Chunks are replicated on different racks, stale replicas are detected and garbage collected
- The master state is also replicated for reliability
- Checksums detect corrupt data

1.3.3 Problems and evaluation

The drawbacks to GFS are inherent in its assumptions and optimisations. For example, it is optimised for appending to files, so random writes are inefficient. Another example is small files lead to inefficient storage due to large chunk sizes, and small chunk sizes are inefficient for metadata purposes.

2 Storage and databases

2.1 CAP and BASE

2.1.1 ACID

ACID databases are the old traditional databases which adhere to being consistent and reliable. ACID stands for:

- **Atomicity** - Atomic operations to ensure operations complete and are uninterrupted by other operations
- **Consistency** - Consistency here means that a transaction preserves all the database rules, such as unique keys. Importantly, ACID's consistency *cannot* be maintained across partitions.
- **Isolation** - Isolation ensures transactions cannot interfere or see each other
- **Durability** - Once a transaction is committed, it can never be lost. This means transactions are not reversible under any circumstance.

ACID becomes complex to manage when databases have to be scaled horizontally. A 2PC protocol is used often.

1. Commit-request phase - Each database involved precommits the operation to indicate whether the commit is possible. If all are in agreement, move to phase 2. Otherwise rollback to the previous state.
2. Commit phase

2.1.2 BASE

BASE is an alternative to ACID that goes to the opposite end of the spectrum, like acids and bases in chemistry. It stands for:

- **Basically Available** - The system is always responsive, but there is a possible loss of time latency, or loss of functionality
- **Soft state** - State of the system may change over time, even without input due to the eventually consistent model
- **Eventually consistent** - The system will become consistent over time, given no new inputs are received

In fact, BASE is a type of system under the CAP theorem, that trades-off consistency for high-availability and partition tolerance.

2.1.3 CAP theorem

The CAP theorem states that any networked shared-data system can have at most two of three desirable properties:

1. Consistency (C) - A guarantee that every node in a distributed cluster returns the same, most recent, successful write
2. High availability (A) - Every non-failing node returns a response for all read and write requests in a reasonable amount of time
3. Partition tolerance (P) - The ability of a system to continue to process data, even if a network partition causes communication errors between subsystems.

The idea behind CAP is the tradeoffs involved in a networked data system. If partitions exist, there are two options:

- Allow only one side to update, sacrificing consistency for availability
- Have the side that receives the response act as though it is unavailable, preserving consistency

This is somewhat misleading as it oversimplifies the tension among the properties. If partitions are *rare*, there is no need to choose between consistency and availability if no partition currently exists, as long as they are tolerated.

In other words, a system should allow consistency and availability most of the time until partitions occur. Then a strategy is needed to detect and account for partitions to preserve consistency and availability. There are three main steps to detect and recover from partitions:

1. Detect the start of a partition
2. Enter an explicit partition mode with limitations
3. Initiate partition recovery

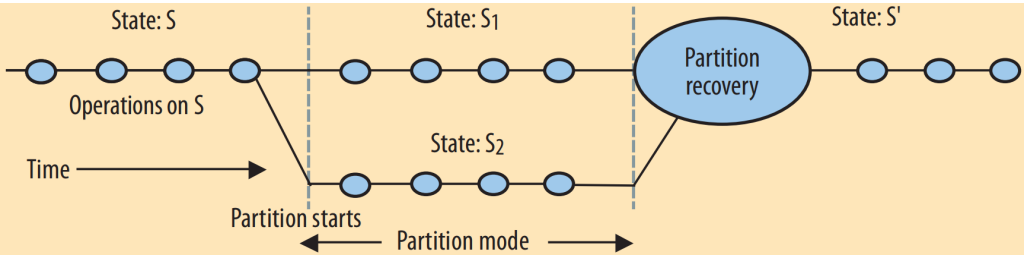

Figure 7: Enter partition mode and partition recovery when one is detected

Evidentally, some operations cannot proceed while in partition mode, so designers need to decide which operations can and cannot proceed. This will limit consistency and availability for some time, but can be somewhat mitigated by letting the user know. A **history** of operations can be logged and executed after partition recovery to maintain consistency.

2.2 Dynamo

Dynamo is a highly available key-value store that sacrifices consistency under certain failure scenarios to maintain high availability. It makes use of extensive object versioning and application-assisted conflict resolution while providing an interface for developers to use.

2.2.1 Key points

One of Amazon's biggest challenges is reliability at a massive scale. For their services, availability is massively important for both partner and customer services. Additionally, scalability is important for continuous growth.

Before Dynamo, relation management databases (RMDBs) were the most popular solution. However, many of a RMDBs' features were not being taken advantage of, and costs are high to run these databases

- 70% of database use is key-value only
- 90% of database use did not need table joins

Dynamo is an example of a system Amazon designed with high availability and scalability in mind, in a distributed data store. S3 (Amazon Simple Storage System) is another example of such a distributed store. Dynamo focuses on the following features:

- Treating failure handing as the normal case
- Tight control over the trade-off between availability, consistency and performance
- Provide a simple interface to handle services that only need primary-key access

This leads to the following key design aspects of Dynamo:

- Consistent hashing with virtual nodes provides
  - Evenly distributed and balanced work-loads across heterogeneous hardware
  - Flexibility to scale up and down
- Trade-offs between reliability and consistency enabled by
  - Inconsistency rules for individual applications
  - Quorum systems for conflict resolution

2.2.2 Specifics/details

Consistent hashing
Dynamo uses **consistent hashing** to distribute the load of data across multiple storage hosts by assigning each node to a random value in the ring that consistent hashing produces.

- Virtual nodes are interspersed across equal partitions
- Because the hashing is uniformly distributed, each node is responsible for an equal number of data items
- When nodes are added or removed, the hash keys are recalculated

Figure 8: Consistent hashing over the data when nodes are added/removed.

Data versioning
Verioning is required to deal with issues where different versions of the data exist.

- Updates are propagated to all replicas asynchronously

- A **vector clock** is associated with each object version, which can be compared to determine version order. A vector clock is a list of (node,counter) tuples.

Quorum-likeness
A quorum is a minimum number of online virtual nodes

- Read and write operations are driven by two parameters: R, the minimum number of replica nodes to read and W, the minimum number of replica nodes to write
- $R + W > N$ yields a *quorum-like* system
- Latency is dictated by the slowest R or W replicas

Dynamo uses a **sloppy quorum** which compromises between consistency and availability:

- Application sets R or W to be very low to ensure constant availability
- Replicas can be stored on healthy nodes downstream in the ring, with metadata specifying that the replica should be sent to the intended recipient later

Replicas are then synchronised using a **Merkle tree**, which is a hash tree where the leaves are the hashed of the values of individual keys. The disadvantage is that many key ranges range when a node joins or leaves and the trees have to be recalculated.

2.2.3 Problems and evaluation

- **Requires scale** - Many of the trade-offs made by the system do not make sense when operating at a smaller scale. For example, sloppy quorum assumes outages are very short
- **Limited queries** - Difficult to add new types of query, as only key/value pairs are allowed in Dynamo
- **Not always consistent** - Cannot be used in cases where consistency is important, such as banking applications.

3 Graph and stream processing

3.1 Pregel

Pregel is a computational model suited for processing graphs and social networks. Programs are expressed as a sequence of iterations, in each of which a vertex can

- receive messages sent in the previous iteration
- send messages to other vertices
- modify its own state and that of its outgoing edges
- mutate the graph topology

The model was designed for efficient, scalable and fault-tolerant implementation on clusters of commodity computers.

3.1.1 Key points

Graphs are an efficient way to represent information on the internet, such as web page links and social networks. It can also be used for processing the shortest path or cluster of nodes in these networks.

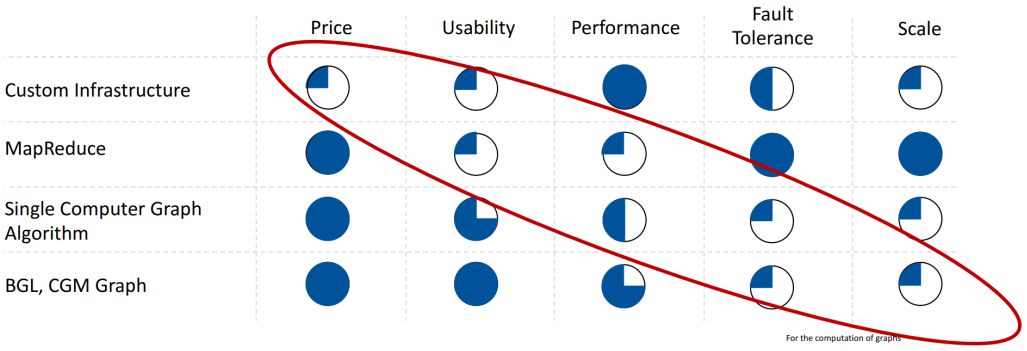

Figure 9: Existing solutions all underperform in some desirable attribute.

Pregel uses a Bulk Synchronous Parallelism (BSP) model where each thread carries out operations in a series of supersteps. Results are exchanged at the end of each superstep. In the Pregel implementation of BSP:

- Input: vertices and edges forming a graph
- Output: a value for each vertex
- Instead of threads, each unit of execution is a vertex in the graph. Vertices can:
  - Send messages to other vertices
  - Alter their own value
  - Alter the value of outgoing edges
  - Request changes to the topology of the graph
- In the exchange step, messages are distributed to target vertices and topology changes are resolved.

Pregel is designed for sparse graphs where communication occurs mainly over the edges. Therefore it has lower performance when most vertices continuously send messages to most other vertices (large synchronise step).

3.1.2 Specifics/details

Combiners

- Problem: Sending many messages can incur a large overhead
- Solution: Combiners are a user defined way to combining multiple messages to the same vertex
- Example: Combining integer messages by adding them

Aggregators

- Problem: It is difficult to carry out computations involving the whole graph
- Solution: Aggregators combine information from each vertex and makes it available in the next superstep
- Example: Counting total edges in the graph

Implementation

- The master assigns partitions of the graph to worker nodes
- Workers compute on the vertices and edges they are assigned to
- Messages are exchanged between workers during synchronisation
- Checkpointing is used for fault tolerance if workers fail. Partitions are re-assigned and work starts from the beginning of the superstep of the last checkpoint

3.2 Borg

Borg is a cluster management system that runs hundreds and thousands of jobs from different applications across a number of cluster, each with tens of thousands of machines.

3.2.1 Key points

Borg addresses how to *deploy* the various cloud applications onto the actual clusters and manage those clusters. It also deals with resource allocation and tracking of the applications while they are running.

Borg provides three main benefits to users:

1. Hides the resource management details of a cluster
2. Operates with high reliability and availability
3. Enables execution of workloads across tens of thousands of machines effectively

Users can simply operate on jobs by issuing remote procedure calls.

Related work

- Apache Mesos - a cluster management system that decouples resource management from resource placement
- YARN - each application has a resource manager that negotiates with the central master node

3.2.2 Specifics/details

Architecture

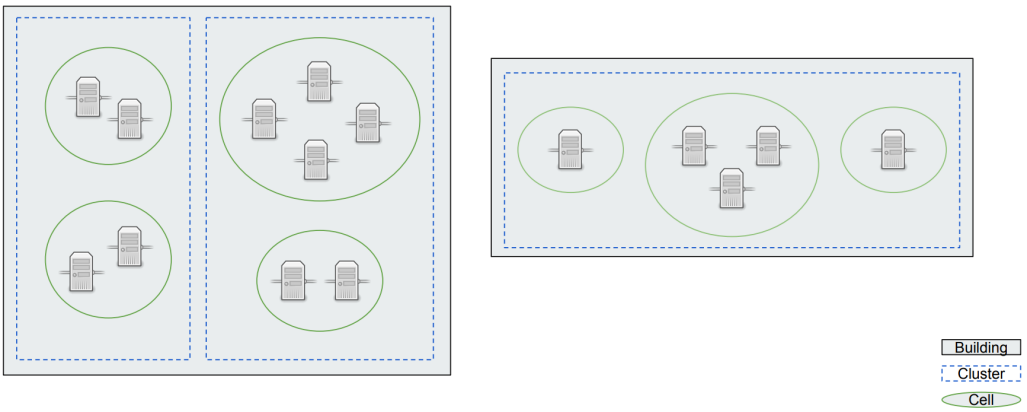

Figure 10: Borg infrastructure and how it is split between buildings, clusters and cells.

The Borg architecture is split between a **Borgmaster** and **Borglets**. Each Borgmaster is associated with one cell.

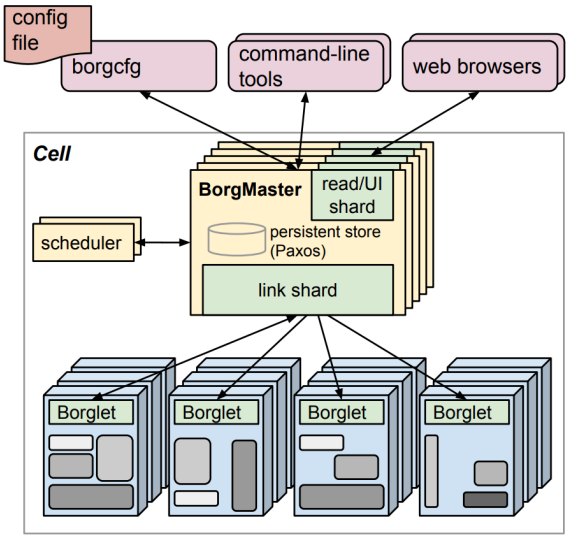

Figure 11: High-level Borg architecture for each cell.

Other details

- Every job has a priority specified by the user
- **Quota** is used to decide which jobs to admit for scheduling. It is expressed as a vector of resources (CPU, RAM, etc.) and associated with a priority
- High-priority quota costs more
- Jobs are admitted only if they have sufficient quota
- Tasks contain a built-in HTTP server that publishes logging data and other performance metrics

3.2.3 Problems and evaluation

What Borg does well

- Allocation with priority and quotas provide a good mapping for resource management
- Borg handles additional application tasks like auto-scaling
- It is a separation of master dependencies and can be seen as just a kernel for other services

What Borg could do better

- Jobs are limited in how they group tasks together
- One IP address per machine is difficult to manage
- Borg optimises more for large scale ‘power users’

4 Resource management

4.1 Mesos

Mesos is a platform for sharing commodity clusters between multiple cluster computing frameworks. Sharing clusters helps improve utilisation and avoids data replication.

4.1.1 Key points

Sharing cluster resources gives high utilisation and is cheaper for companies, but current solutions involve either a static partition or allocated VMs per framework. These solutions are not able to achieve high utilisation nor efficient data sharing.

Mesos takes the approach of a **resource offer**

- Offers encapsulate a bundle of resources that a framework can allocate on a cluster node to run tasks
- Mesos decides *how many* resources to offer each framework based on organisational policies
- Frameworks decide *which* resources to accept and which tasks to run on them

4.1.2 Specifics/details

Architecture

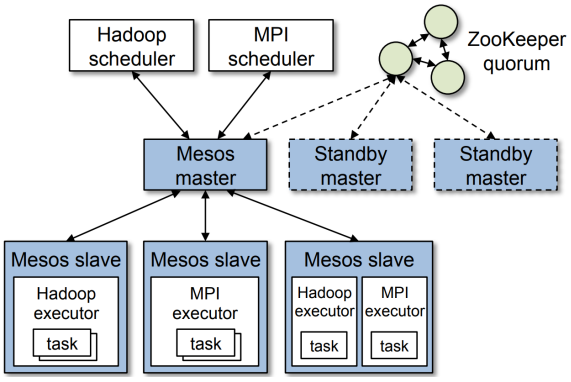

Figure 12: Mesos architecture running the Hadoop and MPI frameworks

Resource offer

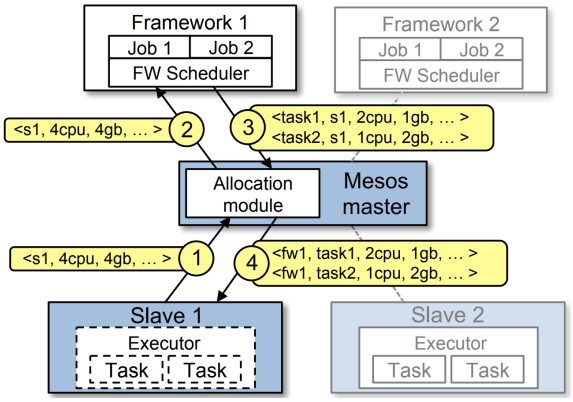

Figure 13: An example resource offer

5 Scheduling

5.1 Patterns in the Chaos

Patterns in the Chaos was a paper that studied the patterns of underlying performance variations in cloud provider systems. The goal of the paper was to be able to predict VM performance and it criticises much of previous cloud performance evaluation work.

- Existing papers do not make all parameters that impact reported results available
- Current papers do not compared their results to previous work, often starting from a clean state
- It is unclear to what extent previous work has been able to stand the test of time

5.1.1 Hypotheses

15 hypotheses were formed related to the nature of performance predictability in the cloud.

1. **Performance varies between instances** - the performance of cloud instances using the same configuration tends to vary
2. **CPU-Bound applications** - CPU-bound applications depends strongly on the served CPU model and tends to vary due to hardware heterogeneity
3. **IO-Bound applications** - IO-bound applications depend strongly on the behaviour of other co-located tenants
4. **Variability of IO-Bound applications** - the performance of IO-bound applications tends to vary within the same instance
5. **Stability of CPU-bound applications** - the performance of CPU-bound applications tends *not* to vary within the same instance
6. **Variability of bursting instance types** - the performance of *any* application using a bursting instance tends to vary within the same instance
7. **Impact of time of day on performance** - performance of a cloud instance depends significantly on the time of the day
8. **Impact of time of day on predictability** - predictability depends significantly on the time of the day

9. **Impact of day of the week on performance** - performance of a cloud instance depends significantly on the day of the week
10. **Impact of day of the week on predictability** - predictability depends significantly on the day of the week
11. **Impact of region on performance**
12. **Impact of region on predictability**
13. **Diseconomies of scale of larger instance types** - the ratio of performance and costs for any application tends to decrease with increasing instance type costs
14. **Stability of larger instance types** - the predictability of performance tends to increase with increasing instance type costs
15. **Price of specialisation** - specialised instances tend to have a better ratio of performance and cost related to their specialisation and worse ratio otherwise

5.2 HotSpot

HotSpot is a system that dynamically selects and migrates applications on the spot markets of cloud providers. By using spot instances, the prices of running applications are cheaper and hopping onto different instances mitigates additional cost and the 'risk' of running on the spot market.

5.2.1 Key points

The core concept behind HotSpot is **automated server hopping**. This consists of two main parts:

- Migrating to spot VMs
- Predicting prices of spot VMs

5.2.2 Specifics/details

There are two metrics to measure to decide whether to migrate to a new VM:

- Cost-efficiency of VMs - The cost-efficiency of a certain spot VM is calculated as the "cost *per unit of resources an application utilises* per unit time", which can be thought of as price/utilisation/hour.
- Cost-Benefit of migration - This is the transaction (overhead) cost of migrating from one VM to another, which takes into account the performance overhead and cost overhead. There is a cost overhead as one must pay for both VMs initially when migrating.

6 Edge computing

6.1 Fog computing

The Foglets system is a programming infrastructure for the geo-distributed computational continuum represented by fog nodes and the cloud. Foglets provides an API for a spatio-temporal data abstraction for storing and retrieving application generated data on local codes and primitives for communication.

6.1.1 Key points

Assumptions

- The existence of a computational continuum that includes sensors and sensor platforms and a complete IaaS interface
- Each device is associated with a certain geophysical location
- Physical devices (*fog computing nodes*) are placed in the network infrastructure
- Fog provides a programming interface that allows managing on-demand computing instances

IoT and cloud computing typically generates event data at the edges, but does the bulk of the processing on the cloud. The issue with this is an explosion in the number of devices and sensors, where the underlying network constraints poses challenges.

The Foglets model focuses on **situation awareness applications** which are especially vulnerable to the network constraint problems with concerns regarding

- Latency
- Scalability
- Security

The idea of **fog computing** is to bring computing closer to the edge as an extension of the cloud. This reduces the round trip latency from sensing source to actuator point.

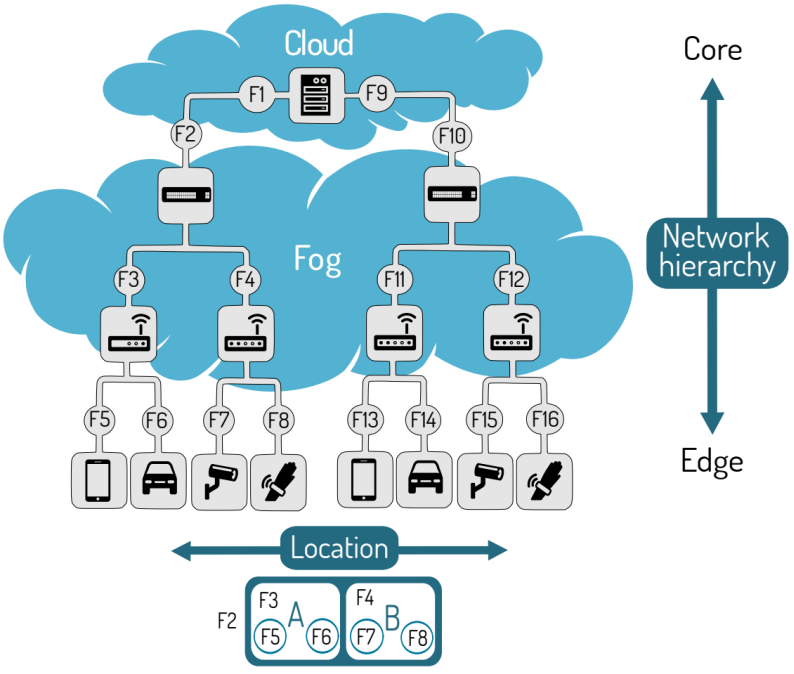

Figure 14: Fog and cloud infrastructure. F[i] denotes an application component launched on a specific computational resource.

Foglet goals:

- Provide a high-level programming model that simplifies development on a large number of heterogeneous devices distributed over a wide area
- Provide an execution environment that enables incremental and flexible provisioning of resources from sensors to the cloud

6.1.2 Specifics/details

Runtime system

- Discovery server - maintains a list of available nodes for a given network hierarchy level and geographical location
- Docker registry - key/value store which contains binaries of applications launched on Foglets
- Entry point daemon - runs on each non-leaf node listening for requests from one layer below and communicates with the discovery server and participates in discovery and migration protocols
- Worker process - executes an application component functionality

Launching applications follows a "two-step" process

1. Write the logic for each component and their handlers
2. Create binaries from this logic and register them with the docker registry
3. Use the API to launch the application

Discovery is done by finding suitable nodes for a given application component. The entry point daemon aids by providing status information of each node as one of BUSY(B), READY(R), READY-DEPLOYED(RD. Nodes which already have the component deployed on them are prioritised.

Deployment works by starting a docker container on a chosen node which will run a worker process.

Migration Migration is important for improving quality of service and managing workloads. There are two aspects to migration

- Computation migration - where handlers are required at old parent and new parent nodes to allow for transfer of process
- State migration - where persisted data must be available to allow for transfer to a new node

A **QoS-driven** migration focuses on an upper bound on parent-child latency. It can either be *proactive* and find a new node to migrate when the latency passes the threshold or *reactive* where the child actively seeks a new parent, and the new parent contacts the old parent.

A **Workload-driven** migration has the entry point daemon monitoring the nodes. Occasionally, when some process becomes more resource intensive, Foglets will attempt to migrate other processes to new parent nodes to give the resource intensive process more room/resources.

7 Machine learning

7.1 Learning scheduling algorithms

7.1.1 Key points