ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Σχεδιασμός και Υλοποίηση Πλατφόρμας Ελαστικής Διαχείρισης Εργαλείων Επεξεργασίας Μεγάλων Δεδομένων σε Υπολογιστικά Νέφη

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Νικόλαου Α. Χαλβαντζή

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2015

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Σχεδιασμός και Υλοποίηση Πλατφόρμας Ελαστικής Διαχείρισης Εργαλείων Επεξεργασίας Μεγάλων Δεδομένων σε Υπολογιστικά Νέφη

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Νικόλαου Α. Χαλβαντζή

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 14/7/2015

(Υπογραφή)	(Υπογραφή)	(Υπογραφή)
.....	.....	.....
Νεκτάριος Κοζύρης	Γεώργιος Γκούμας	Δημήτριος Τσουμάκος
Καθηγητής Ε.Μ.Π.	Λέκτορας Ε.Μ.Π.	Επίκ. Καθηγητής Ιόνιο Παν.

Αθήνα, Ιούλιος 2015

.....

Νικόλαος Χαλβαντζής

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

In loving memory of my mother

Περίληψη

Οι εφαρμογές Μεγάλων Δεδομένων απαιτούν την ύπαρξη σημαντικών σε όγκο υποδομών προκειμένου να πραγματοποιήσουν αποδοτική επεξεργασία σε ένα εύλογο χρονικό διάστημα. Ωστόσο, η ποσότητα των υπολογιστικών πόρων που απαιτούνται δεν είναι πάντα γνωστή εκ των προτέρων και εξαρτάται τόσο από τον όγκο των δεδομένων που υποβάλλονται σε επεξεργασία όσο και από το είδος της επεξεργασίας αυτής. Τα υπολογιστικά νέφη προσφέρουν την ευελιξία της απόκτησης πόρων κατά παραγγελία, με άμεσο τρόπο καθώς και της χρησιμοποίησης τους για όσο χρόνο αυτό είναι απαραίτητο με το ανάλογο κόστος. Ως εκ τούτου, συχνά οι εφαρμογές επεξεργασίας Μεγάλων Δεδομένων χρησιμοποιούν υποδομές υπολογιστικού νέφους. Σε αυτή την εργασία παρουσιάζουμε το BARBECUE (a joB AwaRe Big-data Elasticity CloUd managEment system) – μια πλατφόρμα ελαστικής διαχείρισης πόρων σε υπολογιστικά νέφη για εφαρμογές Μεγάλων Δεδομένων. Επεκτείνουμε τις υπάρχουσες τεχνολογίες, Apache Hadoop και YARN ώστε να έχουν επίγνωση των εργασιών που τους υποβάλλονται προς εκτέλεση, υλοποιώντας μια μορφή αυτόματης, ελαστικής και άμεσης απόδοσης πόρων βασισμένη στη λειτουργία του TIRAMOLA, ένα εργαλείο ελαστικής διαχείρισης πόρων υπολογιστικών νεφών. Το σύστημα BBQ χρησιμοποιεί μια μέθοδο δημιουργίας προφίλ για κάθε πρόγραμμα επεξεργασίας Μεγάλων Δεδομένων. Χρησιμοποιώντας τα συγκεκριμένα προφίλ, μπορεί να προβλέπει πόσοι πόροι απαιτούνται για την εκτέλεση μιας συγκεκριμένης εργασίας (σε συγκεκριμένο χρόνο), 'παγώνει' την εκτέλεση της τελευταίας ώστε να αποκτήσει πρόσβαση σε αυτούς ζητώντας τους από τον εκάστοτε πάροχο της υπηρεσίας του νέφους μέσω του TIRAMOLA, τους μορφοποιεί ώστε να είναι έτοιμοι για χρήση και στη συνέχεια εκτελεί την υποβληθείσα εργασία. Μετά το πέρας αυτής οι επιπλέον πόροι επιστρέφονται στο υπολογιστικό νέφος. Στην παρούσα δουλειά παρουσιάζουμε τις εσωτερικές λειτουργίες των Hadoop και YARN και περιγράφουμε τις τροποποιήσεις στη διαδικασία εκτέλεσης που μας επέτρεψαν την απρόσκοπτη απόκτηση και ενσωμάτωση των επιπλέον υπολογιστικών πόρων. Επιπροσθέτως, τροποποιούμε τον TIRAMOLA ώστε να είναι

σε θέση με τη χρήση των προφίλ διάφορων προγραμμάτων ανάλυσης Μεγάλων Δεδομένων να υπολογίζει τους απαιτούμενους πόρους. Τέλος, διενεργούμε μια πειραματική αξιολόγηση του συστήματος και αποδεικνύουμε τη λειτουργικότητά του. Προσφέρουμε το σύστημα ως λογισμικό ανοιχτού κώδικα.

Λέξεις Κλειδιά: Hadoop, MapReduce, YARN, elasticity, cloud provisioning, TIRAMOLA, profiling

Abstract

Big Data applications require vast amounts of infrastructural resources in order to perform efficient processing in a timely manner. Nevertheless, the amount of required resources is not known a-priory and depends on both the size of the processed data and the type of the required processing. Cloud computing offers the flexibility of acquiring on-the-fly infrastructural resources and utilize them for as long as they are needed in an elastic and pay-as-you go manner. Therefore, Big Data Applications are typically deployed in a cloud setting. In this thesis we present BARBECUE, a joB AwaRe Big-data Elasticity CloUd managEment system by expanding Apache Hadoop and YARN, to perform job-aware on-the-fly resource allocation utilizing cloud elasticity based on TIRAMOLA, a state of the art elasticity framework. BBQ utilizes profiling to detect the correct amount of resources for a specific job, stalls its execution to acquire them from the cloud using TIRAMOLA, configure them, executes the job and after its completion returns the extra resources to the cloud. In this work we present the internals of job execution and scheduling of Hadoop and YARN and we describe our exact modifications in the job execution workflow to allow seamless acquisition and deployment of extra needed resources. We modify TIRAMOLA to employ profiling in order to detect the correct amount of resources. We perform a thorough experimental evaluation in an actual cloud deployment in which we showcase BBQ's functionality and we offer the entire BBQ system as open source.

Keywords: Hadoop, MapReduce, YARN, elasticity, cloud provisioning, TIRAMOLA, profiling

Table of Contents

Table of Figures

Ευχαριστίες

Θα ήθελα να ευχαριστήσω όλους τους καθηγητές μου στο Ε.Μ.Π., κυρίως όμως τον καθ. Νεκτάριο Κοζύρη, για την εμπιστοσύνη και την υπομονή που μου έδειξε καθώς και τις ιδιαίτερα θερμές ευχαριστίες μου στο πρόσωπο του Δρ. Ιωάννη Κωνσταντίνου, ερευνητή του Εργαστηρίου Υπολογιστικών Συστημάτων, ο οποίος υπήρξε εξαιρετικός αρωγός και υποστηρικτής στην προσπάθειά μου αυτή. Ευχαριστώ επίσης το διαχειριστή των υπολογιστικών συστημάτων του Εργαστηρίου, κ. Ιωάννη Γιαννακόπουλο για την άψογη συνεργασία καθώς και τον κ. Ευάγγελο Αγγέλου, μέλος της ερευνητικής ομάδας για τις καίριες παρατηρήσεις και την βοήθειά του. Τέλος, θα ήθελα να ευχαριστήσω την οικογένειά μου για την συμπαράσταση που μου παρείχε καθ' όλη τη διάρκεια των σπουδών μου και ιδιαίτερα στις – αρκετές – δύσκολες στιγμές.

1 *Εκτεταμένη Εισαγωγή*

Η εμφάνιση και καθιέρωση των *Υπολογιστικών Νεφών* (3.1), μόλις πριν από περίπου μια δεκαετία, έχει επιφέρει μια ριζική αλλαγή στον τρόπο με τον οποίο οι εφαρμογές αναπτύσσονται, διαχειρίζονται και λειτουργούν καθώς οι χρήστες μπορούν να έχουν άμεση και εύκολη πρόσβαση σε απεριόριστους πόρους σε ένα pay-as-you-go πλαίσιο. Επιπλέον, η έκρηξη των δεδομένων που δημιουργούνται, καταναλώνονται, αποθηκεύονται και επεξεργάζονται αλλάζει τον τρόπο με τον οποίο εμφανίζονται καινοτομίες προς μια πιο δεδομενο-κεντρική προσέγγιση, όπου οι ιδέες εξάγονται ως αποτέλεσμα της επεξεργασίας τεράστιων ποσοτήτων δεδομένων, όπως αποτυπώνεται στο έργο '*The Fourth Paradigm*' [1]. Έχοντας πρόσβαση σε – θεωρητικά – απεριόριστη υπολογιστική ισχύ και αποθηκευτικούς πόρους, χρήστες και επιχειρήσεις είναι πλέον σε θέση να ανταποκριθούν στην αυξανόμενη ανάγκη της διαχείρισης των τεράστιων ποσοτήτων δεδομένων. Με, σύμφωνα με την IBM, περίπου 2,5 τετράκις εκατομμύρια bytes να δημιουργούνται κάθε μέρα [2], ωστόσο, νέες προκλήσεις έχουν εμφανιστεί σχετικά με τον τρόπο που οι υποδομές νεφών διαχειρίζονται έτσι ώστε οι πόροι που προσφέρουν να χρησιμοποιούνται όσο το δυνατό καλύτερα (πιο αποδοτικά).

Μια σειρά από εργαλεία επεξεργασίας Μεγάλων Δεδομένων αναπτύχθηκαν κατά τα προηγούμενα έτη και χρησιμοποιούνται από πολλές οργανώσεις και εταιρείες σε όλο τον κόσμο. Το πιο δημοφιλές από αυτά είναι το σύστημα Apache Hadoop [3], μια υλοποίηση ανοικτού κώδικα του *MapReduce* framework της Google [4]. Το Hadoop προσφέρει τα βασικά αρχέτυπα επεξεργασίας και αποθήκευσης για κλιμακούμενη κατανεμημένη επεξεργασία. Μια πληθώρα από πιο πολύπλοκα και εξειδικευμένα συστήματα παράλληλης επεξεργασίας, κατάλληλα για Μεγάλα Δεδομένα χρησιμοποιούν ως βάση το Hadoop: πλατφόρμες Μηχανικής Μάθησης όπως το *Apache Mahout* [5], πλαίσια επεξεργασίας

δομημένων δεδομένων, όπως τα *Hive* [6] και *Pig* [7], NoSQL συστήματα, όπως το *Apache HBase* [8] είναι μερικά από τα συστήματα που αποτελούν το οικοσύστημα του Apache Hadoop.

Ένα από τα πιο αξιοσημείωτα χαρακτηριστικά της χρήσης Υπολογιστικών Νεφών είναι η *ελαστικότητα* (3.1.1), δηλαδή, η δυνατότητα ενός συστήματος που βασίζεται σε μια τέτοια τεχνολογία να αποκτήσει ή να απελευθερώσει δυναμικά υπολογιστικούς πόρους ανάλογα με τη ζήτηση. Η ελαστικότητα επιτρέπει στους χρήστες να διαχειρίζονται τους διαθέσιμους υπολογιστικούς πόρους ανάλογα με τις ανάγκες τους, με τη συρρίκνωση ή την επέκταση των υποδομών που χρησιμοποιούν. Η ελαστικότητα είναι, επομένως, κρίσιμη για τη βελτιστοποίηση της διαχείρισης των πόρων. Προκειμένου να επιτευχθεί η βέλτιστη απόδοση χρήστες καλούνται να απαντήσουν σε ερωτήσεις όπως οι ακόλουθες:

- Πόσους πόρους χρειάζομαι;
- Πότε πρέπει να δεσμεύσω/αποδέσμευσω πόρους;
- Τι είδους πόρους πρέπει να ζητήσω από την άποψη του είδους και της ποσότητας (δηλαδή, χρειάζομαι περισσότερο αποθηκευτικό χώρο, ή χρειάζομαι περισσότερη επεξεργαστική ισχύ;)
- Πόσο θα πρέπει να περιμένω μέχρι οι πόροι τους οποίους χρειάζομαι και ζήτησα είναι διαθέσιμοι;

Καθώς η ανάλυση Μεγάλων Δεδομένων γίνεται όλο και πιο απαραίτητη για την έρευνα, οι χρήστες του Hadoop είναι επιστήμονες προέρχονται από τομείς εντελώς διαφορετικούς μεταξύ τους, με εντελώς διαφορετικό υπόβαθρο (όπως πχ. η βιολογία, η οικονομική επιστήμη, κλπ.) χωρίς βαθιά γνώση των συστημάτων, ώστε να κατανοούν τι ακριβώς συμβαίνει κατά τη διάρκεια της εκτέλεσης – στο πολύ χαμηλό επίπεδο, ή πόσοι πόροι είναι επαρκείς ή απαιτούνται για την εκτέλεση των εργασιών τους. Ο μέσος χρήστης ενδιαφέρεται μόνο για την εκτέλεση των εργασιών του με αποτελεσματικό τρόπο, όσον αφορά για παράδειγμα, το κόστος και τον χρόνο εκτέλεσης. Μια τυπική εκτέλεση εργασίας στο Hadoop αντιμετωπίζεται με batch τρόπο: οι χρήστες παρέχουν τον προς εκτέλεση κώδικα σε δυαδική μορφή, μια τοποθεσία όπου βρίσκονται δεδομένα επί των οποίων θα εκτελεστεί ο κώδικας, και ο χρονοπρογραμματιστής του Hadoop ξεκινά μια *εργασία* που αξιοποιεί τους διαθέσιμους πόρους της υποδομής (δηλαδή, έναν αριθμό κόμβων επεξεργασίας) ώστε να παραχθεί το ζητούμενο αποτέλεσμα. Οι τρέχουσες εκδόσεις του Hadoop δεν επιτρέπουν τη δυναμική, άμεση αλλαγή μεγέθους της υποδομής κατά τη διάρκεια μιας νέας υποβολής εργασιών. Ακόμα και στις πιο πρόσφατες εκδόσεις, όταν μια νέα εργασία υποβάλλεται για την εκτέλεση, το διαθέσιμο μέγεθος της υποδομής είναι στατικό και δεν μπορεί να αλλάξει μέχρι

η εργασία να ολοκληρώσει την εκτέλεσή της. Η αδυναμία αυτή ελαχιστοποιεί την ευελιξία και οδηγεί σε κακή διαχείριση.

Φυσικά, υπάρχουν "cloud-ready" εκδόσεις του Hadoop που προσφέρουν στο χρήστη τη δυνατότητα να αλλάξει το μέγεθος της υποδομής του, με την πιο αξιοσημείωτη αυτή της Amazon (Elastic Map Reduce [9]). Παρ' όλα αυτά, έχουν τα ακόλουθα μειονεκτήματα: α) Η αλλαγή μεγέθους γίνεται, όπως εξηγείται παραπάνω πριν από την εκτέλεση της εργασίας· δεν μπορεί να συμβεί δυναμικά κατά τη διάρκεια της εκτέλεσης και β) ο χρήστης πρέπει να αποφασίσει σχετικά με τις ποιοτικές και ποσοτικές απαιτήσεις της εφαρμογής του σε πόρους , κάτι που, όπως αναφέρεται παραπάνω, δεν μπορεί πάντα να γίνει σωστά.

Όλα τα παραπάνω μας οδηγούν στο συμπέρασμα ότι ένα σύστημα που θα επιτρέπει ένα περιβάλλον σε ένα περιβάλλον υπολογιστικού νέφους να προσαρμόζεται στις υποβληθείσες εργασίες που καλείται να διεκπεραιώσει μπορεί να αποδειχθεί ευεργετικό για τη διαχείριση των πόρων. Το σύστημα αυτό θα πρέπει να είναι σε θέση να απαντήσει στις ερωτήσεις που παρουσιάστηκαν παραπάνω για το χρήστη και να επωμισθεί τη λήψη αποφάσεων σχετικά με τη διαχείριση των πόρων.

1.1 Συνεισφορά

Για την αντιμετώπιση των θεμάτων που αναφέρονται παραπάνω, σε αυτή τη διατριβή παρουσιάζουμε το σύστημα BARBECUE (εν συντομία, BBQ) – a jo**B** **A**wa**R**e **B**ig-data **E**lasticity **C**lo**U**d manag**E**ment system – με την επέκταση του Apache Hadoop και YARN, τον ενσωματωμένο διαχειριστή πόρων και χρονοπρογραμματιστή του Hadoop, ώστε να εκτελεί αυτόματη ελαστική διάθεση των πόρων έχοντας επίγνωση των απαιτήσεων της εργασίας που εκτελείται. Έχουμε επίσης τροποποιήσει το σύστημα 'TIRAMOLA' [11], μια πλατφόρμα για την παρακολούθηση και δυναμική διαχείρηση NoSQL συστημάτων βασισμένα σε υποδομές Υπολογιστικών Νεφών, ώστε να μπορέσει να διαχειριστεί μια υποδομή Apache Hadoop.

Εξετάστε το ακόλουθο παράδειγμα: ένας χρήστης αποθηκεύει ένα μεγάλο σύνολο δεδομένων σε ένα μικρό αριθμό στιγμιότυπων του Hadoop Amazon EC2 (VMs). Ο χρήστης θέλει να εκτελέσει λειτουργία επεξεργασίας ιδιαίτερα απαιτητική σε υπολογιστικούς πόρους πάνω στα προαναφερθέντα δεδομένα, η οποία θα παράγει μια μικρή έξοδο δεδομένων: σε αυτή την περίπτωση, ο χρήστης πρέπει να αποφασίσει τη σωστή ποσότητα των επιπλέον πόρων που απαιτούνται, να τους ζητήσει από τον πάροχο της υπηρεσίας και να κάνει τις κατάλληλες ρυθμίσεις, να προχωρήσει στην εκτέλεση της εργασίας και στη συνέχεια να απελευθερώσει

τους επιπλέον πόρους. Το BBQ μπορεί να ελαχιστοποιήσει την προσπάθεια που απαιτείται για να εκτελεστεί η προαναφερθείσα εργασία και ταυτόχρονα να προσφέρει τη βέλτιστη χρήση των πόρων.

Το σύστημά μας επιτρέπει την άμεση, αυτόματη επέκταση ενός Hadoop cluster βασισμένο σε υποδομή υπολογιστικού νέφους (3.5.1) με στόχο τη βελτίωση της απόδοσης κατά την επεξεργασία τεράστιων ποσοτήτων δεδομένων, προκειμένου να ανταποκριθεί σε συγκεκριμένους περιορισμούς. Οι περιορισμοί μπορούν να οριστούν από το χρήστη και επηρεάζουν άμεσα τις ανάγκες της υπολογιστικής υποδομής σε υπολογιστική ισχύ. Το BBQ χρησιμοποιεί την ελαστική ιδιότητα του νέφους ώστε να επιτευχθεί η βέλτιστη αξιοποίηση των πόρων με ένα εντελώς διαφανή για το χρήστη τρόπο. Υποθέτουμε ότι οι περιορισμοί αυτοί του χρήστη συνήθως είναι ένας συνδυασμός ενός άνω ορίου στο χρόνο εκτέλεσης και ενός κάτω ορίου στο οικονομικό κόστος, δηλαδή, ένας περιορισμός του τύπου «*θέλω η εργασία μου να εκτελεστεί χωρίς να πληρώσω περισσότερα από **X** USD/χωρίς να περιμένω περισσότερο από **Y** ώρες*". Μετά την υποβολή της μια νέας εργασίας, το BBQ αυτόματα και διαφανώς ανιχνεύει τη σωστή ποσότητα των πόρων που απαιτούνται σύμφωνα με το μέγεθος του συνόλου δεδομένων και το είδος της εργασίας, δεσμεύει και διαμορφώνει αυτούς τους πόρους, καθυστερεί τη ροής της εργασίας μέχρι η διαμόρφωση να ολοκληρωθεί, την εκτελεί και, στη συνέχεια, αποδεσμεύει τους επιπλέον πόρους για να αποφευχθούν περιττές δαπάνες. Το BBQ αποτελείται από τρία διακριτές οντότητες:

- Μια τροποποιημένη έκδοση του Hadoop
- Μια τροποποιημένη έκδοση του TIRAMOLA
- Μια οντότητα Λήψης Αποφάσεων, βάσει της οποίας ο TIRAMOLA θα λάβει τις αποφάσεις για τη δέσμευση πόρων.

Οι τροποποιημένες εκδόσεις των TIRAMOLA και Apache Hadoop αποτελούν τις κύριες οντότητες της εφαρμογής μας. Η μονάδα λήψης αποφάσεων μπορεί να ενταχθεί στο υποσύστημα του TIRAMOLA. Μπορεί εύκολα να επεξεργαστεί ή να ξαναγραφεί, χωρίς να επηρεάζονται τα υπόλοιπα των κομμάτια του TIRAMOLA, σύμφωνα με τις ανάγκες του χρήστη. Για τους σκοπούς αυτής της εργασίας, έχουμε δημιουργήσει ένα μοντέλο κόστους, το οποίο – με βάση την εργασία του Tian και Chen [12] – αποτυπώνει τη σχέση μεταξύ του μεγέθους των δεδομένων εισόδου, τους διαθέσιμους πόρους του συστήματος, και της πολυπλοκότητα της εργασίας MapReduce που πρόκειται να εκτελεστεί. Αυτό γίνεται προκειμένου να υπολογιστεί η ποσότητα των πόρων που απαιτούνται για την ικανοποίηση των περιορισμών του χρήστη σχετικά με το χρόνο εκτέλεσης χρησιμοποιώντας τα αποτελέσματα μιας χειροκίνητης διαδικασίας δημιουργίας προφίλ με σύνολα δεδομένων διαφόρων μεγεθών. Οι παράμετροι του μοντέλου για κάθε πρόγραμμα MapReduce μπορούν

να υπολογιστούν από πειράματα σε ένα μικρό αριθμό κόμβων – μια διαδικασία γνωστή ως profiling [13]. Χρησιμοποιώντας αυτό το μοντέλο κόστους, μπορούμε να λύσουμε προβλήματα απόφασης, όπως ο καθορισμός της βέλτιστης ποσότητας πόρων που μπορούν να ελαχιστοποιήσουν το οικονομικό κόστος με μια χρονική προθεσμία ή την ελαχιστοποίηση του χρόνου υπό οικονομικό προϋπολογισμό. Έχουμε, επίσης, επιβεβαιώσει πειραματικά τα αποτελέσματα και αποδείξει ότι το σύστημα που παρουσιάζουμε δίνει, πράγματι, τα αναμενόμενα αποτελέσματα.

Το λογισμικό που χρησιμοποιείται και παράχθηκε για τις ανάγκες της παρούσας διπλωματικής εργασίας είναι ανοικτού κώδικα[2] και διαθέσιμο για πειραματισμό.

1.2 Δομή κειμένου

Στο πρώτο κεφάλαιο παρουσιάζουμε εν συντομία στον αναγνώστη το πρόβλημα που έχουμε επιλέξει να αντιμετωπίσουμε και σύντομα περιγράφουμε τη συμβολή μας. Οι τεχνολογίες που αναφέρονται εδώ θα παρουσιαστούν επίσης με μεγαλύτερη λεπτομέρεια και σε βάθος στα επόμενα κεφάλαια. Στο δεύτερο κεφάλαιο, θα γίνει μια εκτενέστερη εισαγωγή στις τεχνολογίες και τις έννοιες που ήδη πιο πάνω αναφέρθηκαν, για λόγους σαφήνειας και πληρότητας. Οι πιο σημαντικές ιδιότητες τους θα επισημανθούν, ενώ θα εξηγήσουμε με ποιον τρόπο η κάθε μια από αυτές είχε κάποιο αντίκτυπο στο σχεδιασμό και την υλοποίηση του συστήματός μας. Κλείνοντας το κεφάλαιο, υποβάλλουμε τις παρατηρήσεις μας και κάνουμε μια περιγραφή των χαρακτηριστικών που διαφοροποιούν το έργο μας από ό, τι υπάρχει ήδη διαθέσιμο ως υπηρεσία. Το τρίτο κεφάλαιο παρουσιάζει το έργο άλλων επιστημόνων και μηχανικών που έχουν αντιμετωπίσει και επιχειρήσει να δώσουν λύσεις σε παρόμοια προβλήματα, όπως η διαχείριση διάθεσης πόρων σε περιβάλλοντα υπολογιστικών νεφών και η πρόβλεψη χρόνου εκτέλεσης για προγράμματα MapReduce. Στο τέταρτο και πέμπτο κεφάλαιο έχουμε προβεί σε ενδελεχή εξέταση των τεχνολογιών που χρησιμοποιούνται, δηλαδή τα Apache Hadoop και TIRAMOLA. Παρουσιάζουμε την αρχιτεκτονική τους και κάνουμε παρατηρήσεις σχετικά με τη λειτουργία τους – παρατηρήσεις που χρησιμοποιήθηκαν κατά το σχεδιασμό και την υλοποίηση του BBQ. Στο έκτο κεφάλαιο παρουσιάζουμε την πειραματική μας διάταξη και αξιολογούμε τα αποτελέσματά των πειραμάτων, συγκρίνοντάς τα με τις αρχικές μας προσδοκίες. Τέλος, στο έβδομο κεφάλαιο προσφέρουμε συμπεράσματά μας σχετικά με τη λειτουργικότητα του συστήματος, προτείνουμε κάποιες βελτιώσεις και τέλος κάνουμε μια σύντομη ανασκόπηση της προόδου μας σε σχέση με τους στόχους που είχαν αρχικά τεθεί.

2 *Introduction*

The emergence of *Cloud Computing* (3.1), just about a decade ago, has brought a radical change in the way applications are deployed, managed and operated, as application owners can have instantly and hassle-free access to unlimited resources in a pay-as-you go manner. Moreover, the explosion of the data being created, consumed, stored and processed is changing the way discoveries are being performed, towards a more data-centric approach, where insights are being extracted from vast amounts of data, as illustrated in the Fourth Paradigm book [1]. Having access to – theoretically – unlimited computing power and storage resources, users and companies are now able to respond to the growing need of managing huge amounts of data. With, according to IBM, approximately 2.5 quintillion bytes created every day [2], however, new challenges have appeared concerning the way cloud infrastructures are being managed so that their resources are utilized to the full extent.

A number of Big Data processing tools have been developed in the previous years and they are being used by many organizations and companies worldwide. The most popular tool is the Apache Hadoop system [3], an open-source implementation of Google's *MapReduce* framework [4]. Hadoop offers the basic processing and storage primitives for scalable distributed processing. On top of Hadoop a number of more complex systems that utilize the basic "nuts and bolts" are being employed: Machine Learning Frameworks such as *Apache Mahout*[5], Data processing Frameworks such as *Hive* [6] and *Pig* [7], NoSQL systems such as *Apache HBase* [8] are a few of the systems that form the Apache Hadoop ecosystem.

One of the most notable strengths of Cloud Computing is *elasticity* (3.1.1), i.e., the ability to dynamically acquire or release computing resources in response to demand. Cloud Elasticity allows users to manage available computing resources according to their needs by shrinking or expanding the infrastructure they are using. Elasticity is, thus, crucial in optimizing

resource management. To achieve optimum performance users are asked to answer questions such as the following:

- How many resources do I need?
- When should I ask for/release resources?
- What kind of resources should I ask for in terms of type and quantity (i.e., do I need more storage, or do I need more processing power?)
- How long do I have to wait until the resources I asked for are available?

As Big Data analysis is becoming more and more essential to research, Hadoop users can be scientists from different fields, such as biology, finance, etc., without deep systems knowledge in order to understand exactly what is happening during the execution, or how many resources are sufficient/needed to execute their tasks. They are only interested in executing their work in an efficient manner, in terms, for instance, of cost and execution time.

A typical task execution in Hadoop is treated as a batch job submission: users provide the execution code in a binary format, a location to the data over which the code will be executed, and the Hadoop scheduler launches a job that harness the available infrastructure resources (i.e., a number of processing nodes) in order to produce the output. Current Hadoop implementations do not allow for dynamic on-they-fly infrastructure resizing during a new job submission. Even in the latest Hadoop versions, when a new job is being submitted for execution, the available size of the infrastructural resources is static and cannot be changed until the job finishes its execution. This shortcoming minimizes flexibility and leads to suboptimal deployments, where the exact resources need to be configured prior to or after the job execution.

There are Apache Hadoop "cloud-ready" versions that offer the user the ability to resize his infrastructure, with the most notable one that of Amazon's (Elastic Map Reduce [9]). Nevertheless, they have the following shortcomings: a) The resizing is done prior to a job execution, i.e., it cannot happen dynamically during the execution and b) the user must decide on its own the correct amount of resources, something that, as previously described, cannot always be done correctly.

All of the above lead us to the conclusion that systems allowing a cloud environment to adapt to the submitted task(s) it is requested to perform can prove beneficial to resource management. These systems should be able to answer the questions presented above for the user and make decisions concerning resource provisioning and management accordingly.

2.1 Thesis Contribution

To address the issues stated above, in this thesis we are introducing BARBECUE (in short, BBQ) a jo**B** **A**wa**R**e **B**ig-data **E**lasticity **C**lo**U**d manag**E**ment system by expanding Apache Hadoop and YARN [10], its generic resource scheduler, to perform job-aware on-the-fly elastic resource allocation. We have also modified TIRAMOLA [11], a modular, cloud-enabled framework for monitoring and adaptively resizing NoSQL clusters, changing its monitoring and decision making modules to allow it to manage Hadoop clusters.

Consider the following motivating example: a user stores a large dataset in a small number of a Hadoop Cluster of Amazon EC2 instances (VMs). The user wants to execute a resource intensive processing operation over the aforementioned dataset, which will produce a small data output: in that case, the user must manually decide the correct amount of extra resources needed, deploy and configure them, execute the job, and then terminate the extra resources. BBQ can minimize the effort required to perform the aforementioned task and perform the optimal resource allocation at the same time.

Our system allows the on-the-fly automatic expansion of a Hadoop cluster on the cloud (3.5.1) to improve performance while processing huge amounts of data in order to meet constraints. Constraints can be set by the user and directly affect the provisioning needs of the cluster, using the elastic property of the cloud to achieve optimum resource utilization in a completely user-transparent manner. We assume that these user-set constraints typically are a combination of an upper bound in execution time and lower bound in financial cost[1], i.e., a constraint of the type "*I want to execute my job without paying more than **X** USD and without waiting more than **Y** hours*". Upon the submission of a new job, BBQ automatically and transparently detects the correct amount of resources needed according to the dataset size and job type, deploys and configures these resources, stalls the job execution workflow until the deployment is finished, executes the workload and then terminates the extra launched resources to avoid extra unnecessary costs.

BBQ consists of three modules:

- A modified – cloud aware – version of Hadoop
- A modified – job aware – version of TIRAMOLA
- A pluggable Decision Making module, based on which TIRAMOLA will make the provisioning decisions.

[1] In the virtual machine (VM) based cloud infrastructure (e.g., Amazon EC2), the cost of cloud resources is calculated based on the number of VM instances used in time units (usually, in hours).

The modified versions of TIRAMOLA and Apache Hadoop platform are the primary modules of our implementation. The Decision Making module can be viewed as a sub-module of TIRAMOLA. It is pluggable and can be easily edited or rewritten, without affecting the rest of TIRAMOLA's components, according to the user's needs. For the purposes of this work, we have built up a cost model, which – based on the work of Tian and Chen [12] – describes the relationship between the size of input data, the available system resources (container capacity in the cluster), and the complexity of the target MapReduce job. This is done in order to estimate the amount of resources required to meet user-set execution time constraints using a manual profiling procedure and datasets of various sizes. The model parameters for any MapReduce programs can be learned from test runs with a small number of nodes – a process known as profiling[13]. Using this cost model, we can solve decision problems, such as the optimal amount of resources that can minimize the financial cost with a time deadline or minimize the time under certain financial budget. We have also experimentally confirmed the results and established that the system we are introducing can gives us, indeed, the expected results.

The software used and produced for the needs of this diploma thesis is Open Source[2] and can be downloaded for experimentation.

2.2 Text Structure

In the first chapter we briefly introduce the reader to the problem we have chosen to tackle and shortly presented our contribution. The technologies mentioned here will also be presented in length in the following chapters. In the second chapter, the reader will be introduced to some of the technologies and concepts already mentioned, for clarity and completeness. Their most important properties will be highlighted; we will also explain in which way each one of them has had an impact in the design and implementation of our system. Closing it out, we present our observations and make a statement describing the features that differentiate our work from what is already available as a service. The third chapter presents the work of other scientists and engineers who have encountered and addressed similar problems, such as cloud provisioning and execution time prediction. In the fourth and fifth chapters we make a thorough examination of the technologies used, namely the Apache Hadoop framework and TIRAMOLA, an automated elasticity provisioning framework over cloud management platforms. We present their architecture and make observations on their functionality – observations which were used during the designing and

[2] http://sourceforge.net/projects/bbqproject/

implementation of BBQ. In the sixth chapter we present our experimental setup and evaluate our results, also comparing them to our initial expectations. Finally, in the seventh chapter we offer our conclusions regarding the system's functionality suggest some improvements and last but not least review our progress in comparison to the objectives set.

3 Background

3.1 Cloud Computing

According to the US National Institute of Standards and Technology, cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [14].

In the Cloud Computing model, a user can unilaterally provision computing resources, such as server time and network storage, automatically without requiring human interaction with the service provider. Resources can be elastically provisioned and released to scale rapidly depending on demand. To the user, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time.

The most common Service Model in which cloud computing is used is *Software as a Service (SaaS)*, according to which, the user is provided the capability to use the provider's applications which run on a cloud infrastructure. *SaaS* applications can be accessible from various client devices through either a thin client interface, such as a web browser (e.g., web-based email), or a program interface. The user does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings.

3.1.1 Cloud Elasticity

Elasticity has originally been defined in physics as a material property capturing the capability of returning to its original state after a deformation. In economical theory,

informally, elasticity denotes the sensitivity of a dependent variable to changes in one or more other variables. In both cases, elasticity is an intuitive concept and can be precisely described using mathematical formulas. The concept of elasticity has been transferred to the context of cloud computing and is commonly considered as one of the central attributes of the cloud paradigm. According to a paper published in 2013 by the Karlsruhe Institute of Technology [15], elasticity is the degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible. Normally, resources of a given resource type can only be provisioned in discrete units like CPU cores, virtual machines (VMs), or physical nodes.

3.2 Big Data

Big data is an evolving term that describes any voluminous amount of structured, semi-structured and unstructured data that has the potential to be mined for information. Although big data doesn't refer to any specific quantity, the term is often used when speaking about petabytes and exabytes of data. Typically these massive amounts of data are collected over time and are difficult to analyze and handle using common database management tools. The data are analyzed for marketing trends in business as well as in the fields of manufacturing, medicine and science. The types of data include business transactions, e-mail messages, photos, surveillance videos, activity logs and unstructured text from blogs and social media, as well as the huge amounts of data that can be collected from sensors of all varieties, such as black box (consisting of data such as voices of the flight crew, recordings of microphones and earphones, and the performance information of aircrafts), stock exchange, power grid data.

The first documented use of the term “big data” appeared in a 1997 paper by scientists at NASA [16], describing the problem they had with visualization. According to it, graphic representation “provides an interesting challenge for computer systems: data sets are generally quite large, taxing the capacities of main memory, local disk, and even remote disk. We call this the problem of big data. When data sets do not fit in main memory (in core), or when they do not fit even on local disk, the most common solution is to acquire more resources.”

In 2008, a number of prominent American computer scientists used the term in their paper “Big-Data Computing: Creating revolutionary breakthroughs in commerce, science, and society” [17], predicting that “big-data computing” will “transform the activities of companies, scientific researchers, medical practitioners, and our nation’s defense and

intelligence operations." The term "big-data computing," however, is never defined in the paper.

However, he now mainstream definition of big data was articulated as far back as 2001 by industry analyst Doug Laney and is since known as the three Vs of big data [18]: volume, velocity and variety.

- **Volume.** In the past, excessive data volume was a storage issue, but with decreasing storage costs, other issues emerge, including how to determine relevance within large data volumes and how to use analytics to create value from relevant data.
- **Velocity.** Data is streaming in at unprecedented speed and must be dealt with in a timely manner. There is an increasing need to deal with torrents of data in near-real time. Reacting quickly enough to deal with data velocity is a challenge for most organizations.
- **Variety.** Data today comes in all types of formats (e.g., structured, numeric data in traditional databases, data created from line-of-business applications, unstructured text documents, email, video, audio data and financial transactions). Managing, merging and governing different varieties of data is something many organizations still grapple with.

These three aspects of what we usually simply refer to as "Big Data" are the reasons why traditional processing algorithms have proved ineffective with them. The sheer volume and variety of these data makes processing them a very demanding task, even more so if the requirement for velocity is taken into account. In 2004, a groundbreaking paper presented by researchers working for Google, introduced the MapReduce programming model [4], which is presented in the following section, which has since been a very popular for Big Data processing using a cluster worker nodes.

3.3 MapReduce Programming Model

MapReduce is a programming model for processing large data sets with a parallel, distributed algorithm on a cluster. The model is inspired by the *Map* and *Reduce* functions commonly used in functional programming, although their purpose in the MapReduce framework is not the same as their original forms. It was developed at Google, initially for indexing Web pages and replaced their previous indexing algorithms and heuristics in 2004. MapReduce works by breaking the processing into two phases: the map phase and the reduce phase. Each phase has key-value pairs as input and output, the types of which may be chosen by the programmer. The programmer also specifies two functions:

- Map, a function that parcels out work to different nodes in the distributed cluster.
- Reduce, another function that collates the work and resolves the results into a single value.

The key contributions of MapReduce are not the actual map and reduce functions, but its scalability – up to clusters of thousands of workers – and fault-tolerance. For a better understanding of the MapReduce programming model, an example is presented in the next section.

3.3.1 An example of MapReduce

The map function takes a value and outputs key:value pairs. For instance, if we define a map function which processes a string and outputs the length of the word as the key and the word itself as the value value delimited with the symbol ":" then map('*mapreduce*') would return 9:mapreduce and map(hadoop) would return 6: hadoop. The map function is stateless and only requires the input value to compute its output value. This allows us to run the map function against values in parallel and provides a huge advantage. Before we examine the reduce function, the MapReduce framework groups all of the values together by key (using a function called the "combiner"[4]), so if the map functions output the following key:value pairs:

3 : the
3 : and
3 : you
4 : then
4 : what
4 : when
5 : nikos
5 : where
6: hadoop
8 : savannah
8 : research
9 : mapreduce

They get grouped as:

3 : [the, and, you]
4 : [then, what, when]
5 : [nikos, where]
6 : [hadoop]

8 : [savannah, research]

9 : [mapreduce]

Each of these lines would then be passed as an argument to the reduce function, which accepts a key and a list of values. In this instance, we might be trying to figure out how many words of certain lengths exist, so our reduce function will just count the number of items in the list and output the key with the size of the list, like:

3 : 3

4 : 3

5 : 2

6 : 1

8 : 2

9 : 1

The reductions can also be done in parallel, again providing a huge advantage. As an analogy, you can think of map and reduce tasks as the way a census was conducted in Roman times, where the census bureau would dispatch its people to each city in the empire. Each census taker in each city would be tasked to count the number of people in that city and then return their results to the capital city. There, the results from each city would be reduced to a single count (sum of all cities) to determine the overall population of the empire. This mapping of people to cities, in parallel, and then combining the results (reducing) is much more efficient than sending a single person to count every person in the empire in a serial fashion. In the next section, we will present WordCount, another example of a MapReduce program which has been used as a benchmark for the experiments conducted for the needs of this thesis.[3]

3.3.2 WordCount with MapReduce

The most common example of MapReduce is for counting the number of times words occur in a corpus. Suppose our working dataset is a copy of the internet, and we want a list of every word on the internet as well as how many times it occurred.

The way this problem would be approached would be to tokenize the documents to be processed, and pass each word to a mapper. The mapper would then emit the word along with a value of 1. The grouping phase would iterate through all the keys (in this case words), and make a list of 1's. The reduce phase would then take a key (the word) and a list (a list of 1's for every time the key appeared on the internet) as input, and sum the list. The reducer would

[3] Example as it appears in : http://www-01.ibm.com/software/data/infosphere/hadoop/mapreduce/

then output the word, along with its count. When this process would have been completed, we would have a list of every word on the internet, along with how many times it appeared.

3.4 Apache Hadoop

MapReduce frameworks provide a specific programming model and a run-time system for processing and generating large datasets that is amenable to a variety of real-world tasks. For programs written in this model, the run-time system automatically parallelizes the processing across large-scale clusters of machines, handles machine failures, and schedules inter-machine communication to make efficient use of the network and storage. The Apache Hadoop software library is such a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures. Hadoop was derived from Google's MapReduce [4] and Google File System (GFS) [19] papers. In the earlier versions of Hadoop, the framework was strictly on to be used with MapReduce applications. In 2013, *Apache YARN* (YARN: Yet Another Resource Negotiator) [10] was integrated in the Hadoop project, allowing for execution of any type of applications – not just MapReduce.

A number of enterprises now use Hadoop in production deployments for applications such as Web indexing, data mining, report generation, log file analysis, financial analysis, scientific simulation, and bioinformatics research. MapReduce frameworks are well suited to run on cloud computing platforms. Cloud-based services are now available that make it easy to set up and run MapReduce programs using Hadoop. Amazon Elastic MapReduce [9] is a hosted framework that runs dynamically-provisioned Hadoop clusters using Amazon Elastic Compute Cloud (EC2) [20] and Simple Storage Service (S3) [21] – dynamic provisioning in this case indicating that users can ask for any number of compute instances running Hadoop. Alone, Hadoop is a software market which in 2012 IDC predicted will be worth $813 million in 2016, but it’s also driving a Big Data market the research firm predicts will hit more than $23 billion by 2016 [22].

3.5 Review

The sections above make the reasons why Hadoop was our platform of choice quite clear. Being designed to support “embarrassingly parallel” algorithms, such as MapReduce

programs, a Hadoop cluster is perfectly suited to be the cornerstone of any system which is designed for the processing of Big Data. It is also clear why a cloud service could be an ideal infrastructure to host a Hadoop cluster, since its properties fit the needs of the Big Data processing platform.

Cloud Services (i.e. Amazon Elastic MapReduce) have made access to a Hadoop cluster on a cloud possible for many users for whom owning a proprietary cluster is not a viable option but still a Hadoop cluster takes a lot of effort to configure and manage. Configuration has to be made after the purpose of the cluster has been very well defined. Such a cluster would need careful provisioning – meaning that it would require the user to possess the expertise to predict the resources which are suitable for both the size of data to be processed and the programs and algorithms which will be used to do the processing.

Although cloud services do offer elasticity, they still require the user to be an expert and perform the cluster provisioning. They also fail to use the elastic property of their cloud infrastructure in a dynamic way. Our system offers a different form of elasticity, which lifts provisioning duties off the user's responsibilities and can allow anyone. The following paragraph will focus on that aspect of our work and explain the term "*automated, on-the-fly*" elasticity.

3.5.1 "Automated, on-the-fly" Elasticity: a fine line

A Hadoop cluster is, of course, elastic – meaning that:

a. It can scale out, since scalability is one of the framework's signature properties
b. The set up of a cluster can be modified at any time by adding or removing nodes.

Taking advantage of the elastic property of a Hadoop cluster, however, is not simple. Adding a new node to an existing cluster would mean that its software is up-to-date, and its cluster-specific configuration matches that of the rest of the nodes of the cluster. Similarly, removing existing nodes from a Hadoop cluster should guarantee that no data loss might incur as a consequence. Assuming that these conditions are met, elasticity can be accomplished. In services such as Amazon Elastic MapReduce, these requirements are already met, albeit with some restrictions (e.g. single node clusters cannot be expanded) [23].

Our definition of the term *automated, on-the-fly* elasticity, though, exceeds the simple operations of expanding or shrinking an idle Hadoop cluster by provisioning or deprovisioning cloud resources. We mean a *job-specific* form of elasticity, which allows the acquisition of resources to boost the performance of a specific MapReduce job. When a MapReduce job is submitted to the Hadoop platform in order to be executed against a large dataset, BBQ will work as a fully integrated system and automatically:

- Chose to either request resources or not, leaving the cluster as is,
- Configure the new cluster nodes, if one or more are eventually added,
- Configure the MapReduce program to utilize the cluster in an optimal way.

Automatic resource provisioning takes into account a set of constraints to be met, provided by the user in a high-level context (as mentioned above, either an upper bound for execution time or a lower bound for financial cost), as well as the complexity of the program to be executed and the size of the data to be processed.

4 Related Work

Researchers have proposed ways to predict the resources a Hadoop MapReduce framework would require to execute a certain MapReduce program in order to use the cloud in an optimal way since the popular MapReduce platform started becoming the Big Data analytics go-to tool. Most of these studies involve building profiles of the programs in question by running sample executions of them with different datasets/configurations/deployments and recording performance metrics for each execution such as completion time, etc. Before we move on, we present a short description of profiling techniques.

4.1 Profiling

Producing a system that could make predictions about cluster performance and execution time solely based on the size of the input data would be ideal. Unfortunately, the variety of operations the incoming (Big) Data can be put through is vast and the algorithms that can implement those are countless. This renders one-fits-all solutions practically impossible.

In order to be able to make accurate estimations though, it is possible to build profiles of certain programs. Profiles are often used by Big Data engineers. Profiling data on a MapReduce program can itself be considered Big Data, as there are countless variables to be possibly taken into account. A profile can be built by running the program in question multiple times, changing a number of (independent) variables that could affect its performance – such as input data size or type (compression, format), cluster size, configuration settings etc.) - and then keeping records of metrics that are considered important. The profiling data can then be processed and help create a mathematical model which can produce predictions about actual jobs.

When a system which relies on profiling is set up for the first time, the profiling data have to be collected through experiments, executed precisely for that purpose. A program's profile can however continue to evolve as long as the system is alive, by collecting data and reviewing the profiles built every time a job that matches it, is executed.

Another interesting aspect of profiling is the fact that more than one program can match a profile. Once a program has been matched to an existing profile, future predictions can become even more accurate, supposing that the initial model was accurate itself. In order to match a new program to an existing profile, a preliminary run would have to be completed, collecting profiling data and then comparing them against those of the existing profiles.

4.2 Cluster provisioning for MapReduce Big Data analysis

Tian and Chen [12] introduce and evaluate a method to optimize resource provisioning to minimize the financial charge for a specific job. In this paper, the whole process of MapReduce processing is studied and the authors build up a time cost function that explicitly models the relationship between the amount of input data, available system resources, and the complexity of the program to be executed for the target MapReduce job. We have used this cost function in order to design our system's decision making module – which uses the aforementioned cost function to solve optimization problems.

Kambatla et al. present a platform which uses profiling in order to detect the ideal configuration setup for MapReduce1.0 programs [24]. Their system uses a database which matches a set of profile fingerprint signatures, unique for every MapReduce program, to each program's ideal configuration, detected by running a number of experiments; incoming programs are executed using a small fraction of the original dataset and a portion of the resources of the cluster in order for their own unique signature to be produced. The new profiling data is then compared against the database of known profiles and the best configuration of the program with the closest matching profiling signature is used. This approach is based on the MapReduce1.0 framework, where map and reduce slots-per-node values where explicitly set by the user. Such settings cannot be used in a Hadoop2.0 context, where the concept of separately defined map and reduce slots has been obsolete. A similar context can be found in the work of Babu [25].

Starfish: A Self-tuning System for Big Data Analytics [26] is a powerful, complex tool for predicting optimal provisioning and configuration settings for Hadoop clusters, which relies on building detailed profiles of programs. Profiling can happen both dynamically – during the execution of a MapReduce job – or virtually, by using heuristics as explained in [27]. It then

uses the profiling data to make predictions on optimum configuration settings and cluster provisioning. It also includes a what-if-engine, which can predict the performance of a MapReduce job given a cluster configuration and input. Additionally, Verma et al. have designed and documented a similar system which also uses profiling in order to make provisioning predictions [28].

All of the systems mentioned above are implemented on a level on top of Hadoop. The BBQ system offers a simpler implementation which is fully integrated within Hadoop, also integrating an automatic provisioning platform in BBQ TIRAMOLA. Additionally, the systems described above focus on modifying Hadoop configuration settings – something only possible when launching MapReduce programs on dedicated clusters, created especially for that purpose. BBQ Hadoop is designed in the perspective of YARN, as a persistent infrastructure layer offering a framework for the execution of MapReduce programs. Hadoop configurations are considered as bound to the infrastructure (i.e. we allow 3 YARN `Containers` – 5.2 – per node in a cluster with quad-core VMs). The fact that the decision making module is pluggable means that a more complex, robust implementation can replace the current one and expand BBQ's functionality.

The existence of so many works on the same topic proves that in effect the problem of cluster provisioning for Big Data processing cannot have an all-for-one solution. Different types of applications need to use the provided resources in very different ways – some are compute-heavy, spending a large portion of execution time to process input data while others are read/write-heavy, spending a large portion of execution time to read or write data down to persistent storage. It makes perfect sense that different types of applications require a different approach when it comes to cluster provisioning and configuration. Some of the systems mentioned above give the user the option to build a profile of a certain MapReduce program based on which configuration and performance will be optimized in the future. We argue that in most cases such an option gives very little advantage, since users who need access to Big Data analysis tools usually use specific ones, chosen to fit their requirements. There are cases where pre-configuring the system for specific applications, as BBQ currently does, might be the best practice. BBQ detaches the decision making module and the policies which resource provisioning follows in an elegant way, which might provide no automation during the profiling process but can manage and provision a Hadoop cluster used with these specific applications more efficiently instead.

5 *Technical Overview of the Hadoop Platform*

5.1 Apache Hadoop1.0 and Mapreduce1.0

Before the Hadoop MapReduce framework is presented in detail, some background information about the evolution of the project would be useful to be presented.

In the first versions of the project, Hadoop MapReduce could be broken down into three major facets:

- The end-user *MapReduce API* for programming the desired MapReduce application.
- The *MapReduce framework*, which is the runtime implementation of various phases such as the map phase, the sort/shuffle/merge aggregation and the reduce phase.
- The *MapReduce system*, which is the backend infrastructure required to run the user's MapReduce application, manage cluster resources, schedule thousands of concurrent jobs etc.

This separation of concerns had significant benefits, particularly for the end-users – they could completely focus on the application via the API and allow the combination of the MapReduce Framework and the MapReduce System to deal with issues such as resource management, fault-tolerance, scheduling etc.

The Apache Hadoop MapReduce System was composed of the `JobTracker`, the master, and the per-node slaves called `TaskTrackers`.

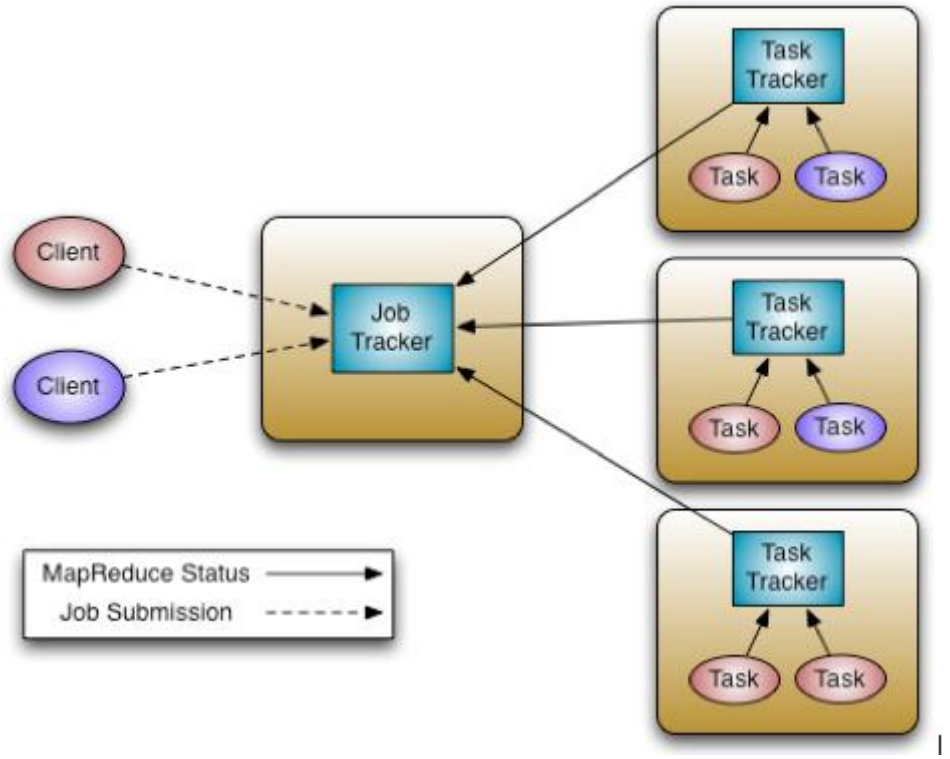

Figure 5-1. Hadoop MapReduce1.0 Architecture[29]

The `JobTracker` was responsible for resource management (managing the worker nodes i.e. `TaskTrackers`), tracking resource consumption and availability and also job life-cycle management – scheduling individual tasks of the job, tracking progress, providing fault-tolerance for tasks etc.

The `TaskTrackers` had simple responsibilities, namely launching or tearing down tasks on orders from the `JobTracker` and heartbeating task-status information to the `JobTracker`.

As Hadoop became more and more successful, it was also becoming obvious that a radical overhaul of the framework was needed. The main problem with Hadoop1.0 was the fact that it was intertwined with MapReduce. In fact, MapReduce Applications were the only kind of programs supported by the early versions of Hadoop. In addition to that, several other aspects needed to be addressed, regarding scalability, cluster utilization, ability for customers to control upgrades to the stack i.e. *customer agility*.

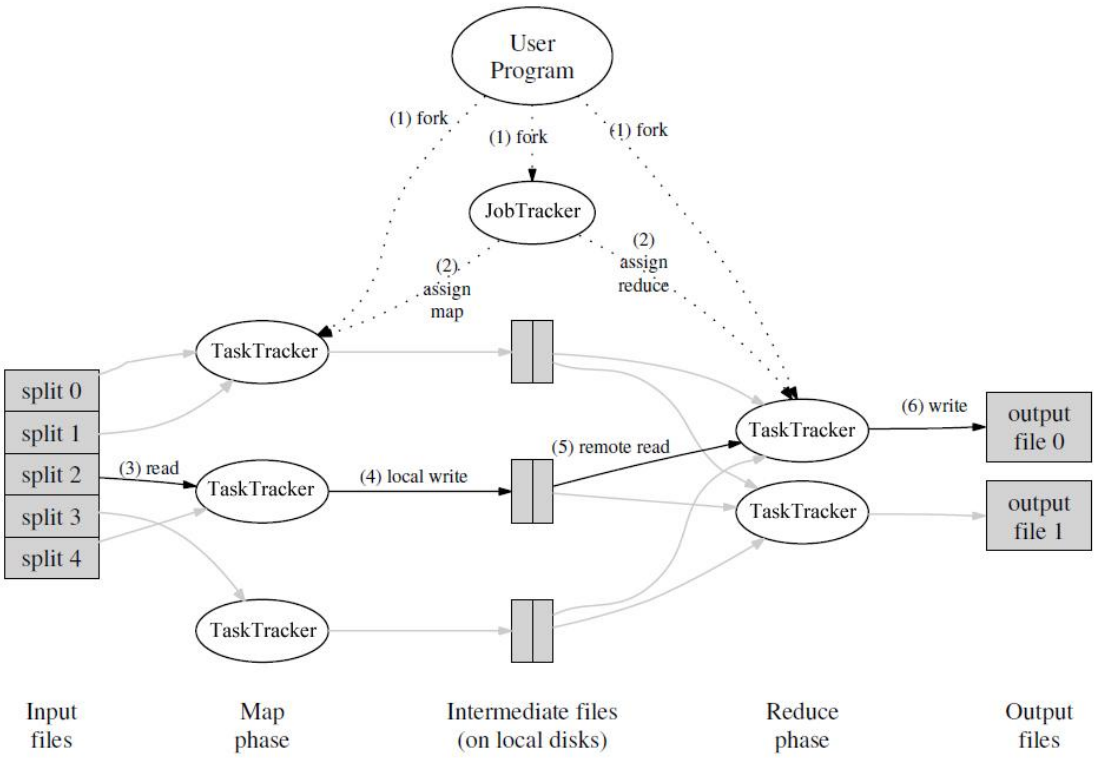

Figure 5-2. Hadoop MapReduce1.0 Execution Overview form the Google paper[4]

5.2 Apache Hadoop2.0 YARN and Mapreduce2.0

From version 2.0 on, the Hadoop project included Apache YARN [10]. The fundamental idea of YARN is to split up the two major responsibilities of the `JobTracker`, resource management and job scheduling/monitoring, into separate daemons: a global `ResourceManager` (`RM`) and per-application `ApplicationMaster` (`AM`), completely separating the framework from the application layer. This way, some of the former `JobTracker's` functions are now moved to a central allocator which can use an abstract description of tenants' requirements, but remains ignorant of the semantics of each allocation.

The `ResourceManager` is the ultimate authority that arbitrates resources among all the applications in the system. The per-application `ApplicationMaster` is, in effect, a *framework specific* entity and is tasked with negotiating resources from the `ResourceManager` and working with the `NodeManager`(s), the per-node slave(s), to execute and monitor the component tasks.

The `ResourceManager` has a pluggable `Scheduler`, which is responsible for allocating resources to the various running applications subject to familiar constraints of capacities, queues etc. The `Scheduler` performs its scheduling function based on the *resource*

requirements of the applications, which are declared to the `RM` through the `AM`; it does so based on the abstract notion of a *Resource Container* which incorporates resource elements such as memory, CPU, disk, network etc. Depending on the application demand, scheduling priorities, and resource availability, the RM dynamically allocates Container – to applications to run on particular nodes. The `Container` is a logical bundle of resources (e.g., 2GB RAM, 1 CPU) bound to a particular node. YARN completely departs from the static partitioning of resources for mappers and reducers; it treats the cluster resources as a (discretized) continuum, which allows for better resource utilization.

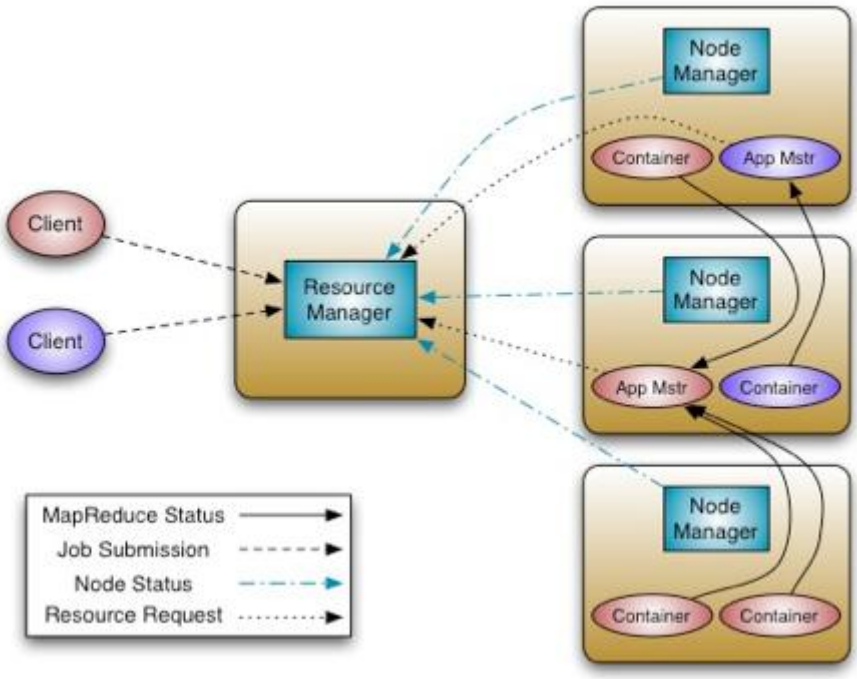

Figure 5-3. Hadoop MapReduce2.0 Execution Overview[29]

The per-application `ApplicationMaster` has the responsibility of negotiating appropriate resource `Containers` from the `Scheduler`, tracking their status and monitoring for progress. It coordinates the logical plan of a single program by requesting resources from the `RM`, generating a physical plan from the resources it receives, and coordinating the execution of that plan around faults. From the system perspective, the `ApplicationMaster` itself runs as a normal `Container`.

The `NodeManagers` are responsible for launching the applications' containers, monitoring their resource usage (CPU, memory, disk, network) and reporting these metrics to the `ResourceManager`, through a periodical heartbeat. `NMs` are also responsible for monitoring resource availability, reporting faults, and container lifecycle management (e.g., starting,

killing). The RM assembles its global view of the cluster state from these snapshots of individual NMs' states.

5.2.1 MapReduce Application Execution Overview with YARN

In this section an overview of the execution of a Hadoop MapReduce program as a YARN application will be presented, highlighting the points of significance which allowed our modifications to help achieve our initial objectives. These modifications will be presented and thoroughly explained in following chapters.

Jobs are submitted to the RM via a public submission protocol and go through an admission control phase during which security credentials are validated and operational and administrative checks are performed. Accepted jobs are passed to the Scheduler to be run. Once the scheduler confirms that it has enough resources, the application state is changed from ACCEPTED to RUNNING. Besides internal bookkeeping, this involves the allocation of a container for the AM and its spawn on a node in the cluster. A record of accepted applications is kept in persistent storage and can be recovered in case of an RM restart or failure.

The ApplicationMaster is the "head" of a job, managing all lifecycle aspects and execution pipeline events including dynamically increasing and decreasing resources consumption, managing the flow of execution – e.g., running reducers against the output of maps, handling faults and computation skew, performing other optimizations. In fact, the AM can run user code, which can allow for the development of new types of applications – but can also cause security issues, exactly because of the fact that they can be user-created. YARN assumes that ApplicationMasters are buggy or even malicious and therefore treats them as unprivileged code (does not give them permissions to YARN configuration files; they can only access application-relative resources). ApplicationMasters can be written in any programming language since all communication with the RM and NM is encoded using extensible communication protocols. However YARN comes with a default ApplicationMaster for MapReduce applications, a class named MRAppMaster. Typically, an AM will need to harness the resources (CPUs, RAM, disks etc.) available on multiple nodes to complete a job. As mentioned previously, to obtain containers AM issues resource requests to the RM. The RM will attempt to satisfy the resource requests coming from each application according to availability and scheduling policies. When a resource is allocated on behalf of an AM, the RM generates a lease for the resource, which is pulled by a subsequent AM heartbeat. Once the ApplicationMaster discovers that a container is available for its use, it encodes an application-specific launch request with the lease. In MapReduce, the code

running in the container is either a map task or a reduce task. Overall, a YARN deployment provides a basic, yet robust infrastructure for lifecycle management and monitoring of containers, while application-specific semantics are managed by each framework.

5.2.2 Evaluating YARN

With the advent of YARN, the user is no longer constrained by the MapReduce development model, but can instead design and develop more complex distributed applications. In fact, the MapReduce model can simply be one more in the set of possible applications that the YARN architecture can execute, in effect exposing more of the underlying framework for customized development. This is powerful because the usage model of YARN is potentially limitless. YARN can become a layer on top of which a number of distributed applications can run, rendering the YARN cluster a multipurpose entity with an exciting potential of Big Data processing – essentially undertaking the role of a cluster Operating System.

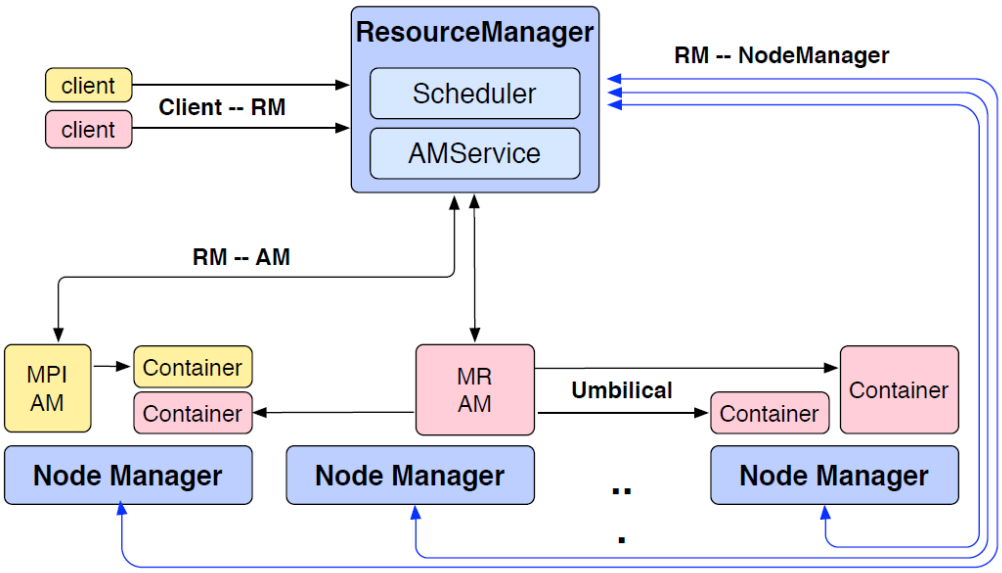

Figure 5-4. YARN Architecture (in yellow and pink two applications running.)[30]

5.2.3 Hadoop MapReduce 2.0 Execution Overview

In this section an overview of the execution of a Hadoop MapReduce program as a YARN application will be presented, highlighting the key points.

YARN treats each application submitted for execution in the same way, regardless of its type, in our case MapReduce. The resource management layer has no access to any kind of information relative to the application. Instead, the ApplicationMaster is responsible for managing all lifecycle aspects and submit resource requests to the ResourceManager.

As mentioned above, when a client submits an application to the `ResourceManager`, the latter allocates a container for the `AM`, which spawns on a node in the cluster. The `AM` is responsible for the execution of the application. The `RM` is a single point of failure in YARN.

The input data is then automatically partitioned into a set of M splits, called `FileSplits`. The `ApplicationMaster` will launch one `MapTask` for each map split. Typically, there is a map split for each input file. If the input file is too big (bigger than the HDFS block size) then we have two or more map splits associated to the same input file, which is the most common case.

5.2.3.1 Map Phase

The `MRAppMaster` immediately asks for containers needed by all `MapTasks`: one `MapTask` container required for each `MapTask`. All container requests for `MapTasks` try to exploit data locality. A request for a new container will be served by the allocation of:

This is however only a hint to the `ResourceScheduler`, which is free to ignore data locality if the suggested assignment is in conflict with its goal.

When a container is finally assigned, it is populated by an instance of the class `YarnChild`, which is responsible for the invocation of both map and reduce tasks. In the case of a `MapTask`, an instance of the program provided by the user as the map function becomes initiated and starts executing using the new container's resources.

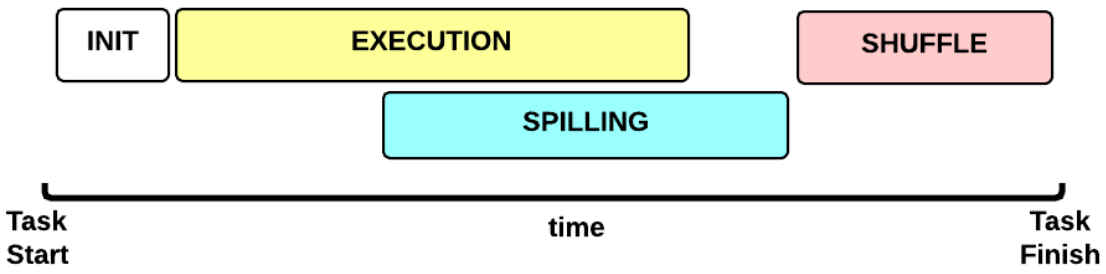

Figure 5-5. Map phase [30]

Each MapTask's execution timeline can be analyzed in 4 phases:

- **Initiation phase**: the environment in which the `MapTask` will run is being created. This includes the instantiation of metadata classes relevant to the input, output and execution context,
- **Execution phase**: for each (key, value) tuple within the map split, the `Mapper.run()` function, provided by the user is executed,
- **Spilling phase**: it runs on a separate thread; the map output is stored in an in-memory buffer; when this buffer is *almost* full then it starts spilling its content into local files, using a user-defined partitioning function to split the output in partitions, each corresponding to a single reducer[4],
- **Shuffle phase**: after all map output has been spilled it is merged and packaged for the reduce phase.

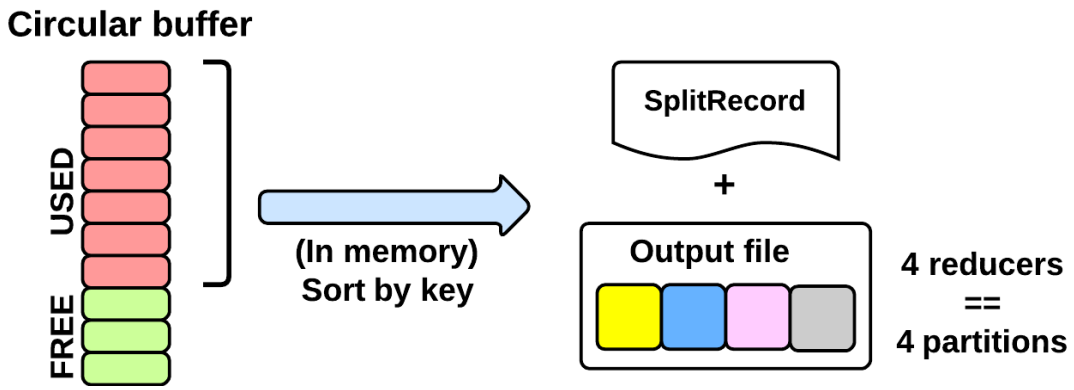

Figure 5-6. Map phase: spilling [30]

[4] For efficiency reasons, sometimes it makes sense to use a combiner class to perform a reduce-type function. If a combiner is used then the map key-value pairs are not immediately written to the output. Instead, they will be collected in lists, one list per each key value. When a certain number of key-value pairs have been written, this buffer is flushed by passing all the values of each key to the combiner's reduce method and outputting the key-value pairs of the combine operation as if they were created by the original map operation.

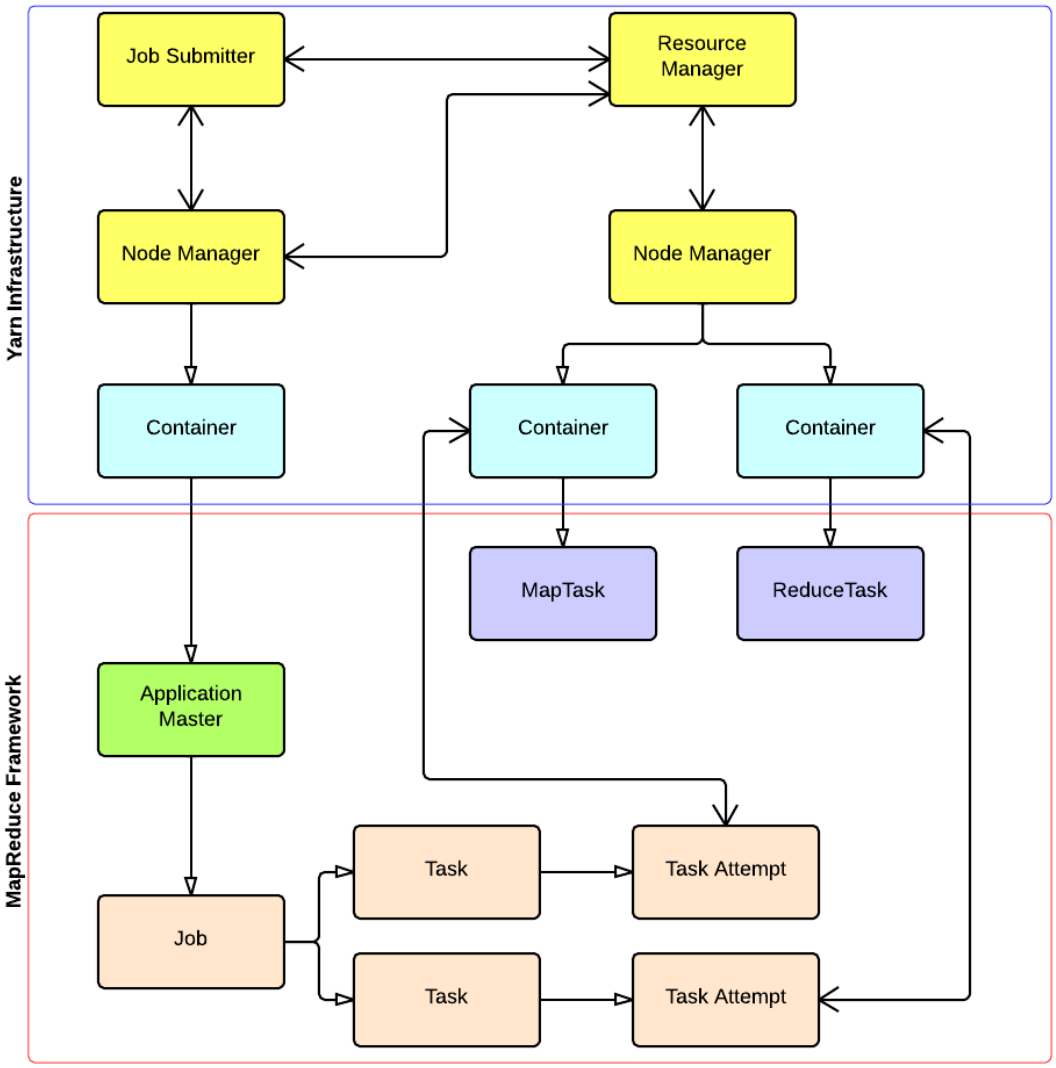

Figure 5-7. YARN Infrastructure vs MapReduce framework [30]

5.2.3.2 Reduce Phase

The MapReduce `ApplicationMaster` (`MRAppMaster`) waits until a preset portion of `MapTasks` has finished. Then, if all `MapTasks` have containers assigned and are running or have already finished then all remaining reducers are being scheduled for execution. Otherwise, the `MRAppMaster` tries to assign containers to either new `MapTasks` or `ReduceTasks`. `MapTask` requests have a higher priority than `ReduceTasks` requests, as all `MapsTasks` have to be completed for the `ReduceTasks` to finish as well. Data locality is not a factor in assigning `ReduceTask` containers, as map output files are scattered around every node of the cluster.

Similarly to what happens when a Container for a `MapTask` is finally assigned, it is populated by an instance of the class `YarnChild`. In the case of a `ReduceTask`, an instance of the

program provided by the user as the reduce function becomes initiated and starts executing using the new container's resources.

In proportion to what we saw with the `MapTasks` above, each `ReduceTask's` execution timeline can be analyzed in 3 phases:

- **Initiation phase**: the environment in which the `ReduceTask` will run is being created. This includes the instantiation of metadata classes relevant to the input, output, possible compression codecs and the shuffle phase context,
- **Shuffle phase**: several parallel fetchers are spawn and start collecting the map output files from remote `NodeManagers`, copying them either in memory or in the local disk; all input files are added in a queue of files to be merged in order to compile the input for the Reduce function
- **Execution phase**: some more `ReduceTask` metadata classes are instantiated; for each (key, [<values>]) tuple, the `Reducer.run()` function, provided by the user, is executed.

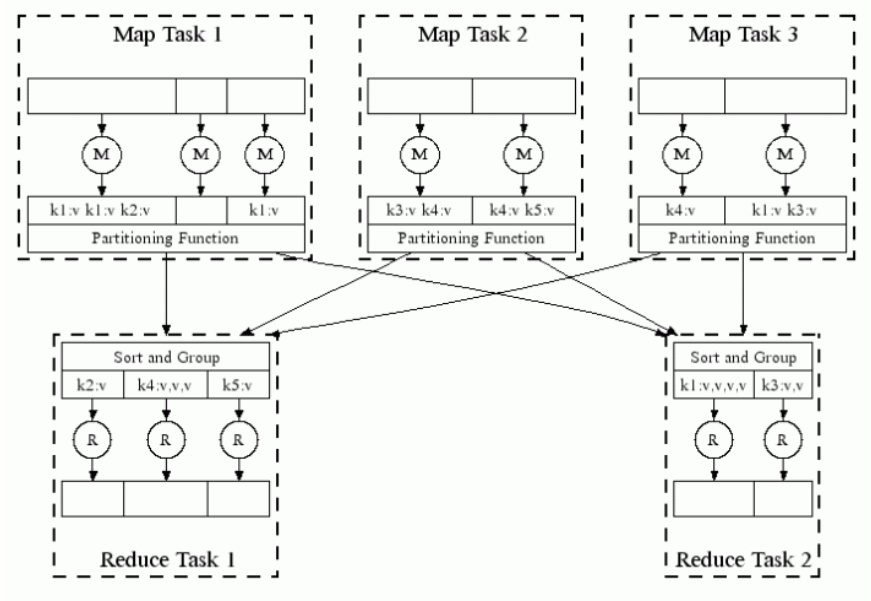

Figure 5-8. Parallel processing in MapReduce, from the Google paper [4]

5.3 Observations

The initial task to be achieved in order to create a system which can offer automated, on-the-fly elasticity for Hadoop MapReduce programs was to track down the various actors in the YARN-Hadoop platform and carefully study their interactions. We had to find a way to obtain metadata information (i.e. number of input splits, number of reducers to be run, type of MapReduce job) about a MapReduce job as soon as it is submitted to YARN but before the

execution starts fanning out to the `NameNodes`, because by then the application configuration and context files will have been created, and any cluster resource changes would not be detected by the job for execution. This information would then have to be fed to a decision making module which would process it and – taking into account the user-set restrictions, which also affect provisioning needs, as stated in the introductory section of this thesis – finally make a decision about whether to expand or not the cluster, and if yes by how many nodes, act on it and allow the execution to be continued.

In the presentation of YARN and MapReduce2.0 execution overviews above, it became obvious that YARN's `ResourceManager` and the application-specific `ApplicationMaster` (in the case of MapReduce, the `MRAppMaster`) are the two coordinators of the whole execution framework. The `ResourceManager` is the module aware of the total amount of the cluster resources (infrastructure context), while the `MRAppMaster` can have access to the MapReduce job metadata (job context). These two actors will prove to be of great significance in our implementation, as presented in the sixth chapter in which the architecture and full functionality of our system will be revealed, as well as the interactions between the various modules and the necessary modifications which the Hadoop framework has went under.

6 *TIRAMOLA*

TIRAMOLA [11] is a modular, cloud-enabled framework for monitoring and adaptively resizing noSQL clusters. The system was originally designed to incorporate a decision-making module which allowed for optimal cluster resize actions in order to maximize any quantifiable reward function provided together with life-long adaptation to workload or infrastructural changes.

Its main components are:

- A main class which orchestrates its various modules,
- A class which implements the interaction with the cloud provider – the *cloud management* module (e.g. offers functions which launch virtual machines - VMs, request list of active ones etc.)
- A class which implements the interaction with the noSQL cluster and uses the functions of the class above – the *cluster coordinator* module (e.g. offers functions which add/remove nodes to the cluster)
- A decision making module

When launched, the TIRAMOLA daemon does one of the following two things:

a. It starts a fresh cluster, automatically spawning new virtual machines and populating them with a precooked image of the software used and also copying predesigned configuration files in the VMs to set up the noSQL systems and start the services or

b. It assumes control of an existing cluster.

It then monitors the cluster, by reading metrics provided by an external source and periodically feeds those to the decision making module in an infinite loop. The latter decides whether the cluster is overprovisioned, underprovisioned or neither of the two according to a

selected user policy. Depending on the decision made, it can release one or more VMs which had been until now active nodes of the cluster, make a request to the cloud provider to launch new VMs (which are then automatically configured and added to the cluster), or simply do nothing.

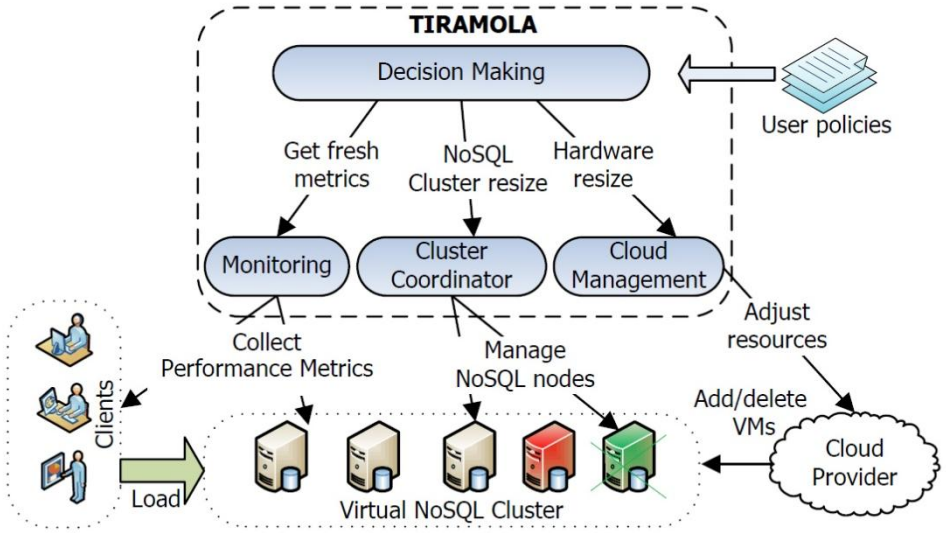

Figure 6-1. Original TIRAMOLA Architecture [11]

7 *The BBQ system*

The system we introduce in this diploma thesis is a modified version of Apache Hadoop which supports automated, on-the-fly cluster elasticity for the execution of MapReduce programs in a cloud environment, subject to user-defined restrictions. For this implementation, we have given the user the option to set an upper bound for execution time or a lower bound for financial cost, assuming a cloud provider charges the use of virtual machines per time units. These user-set constraints can directly affect the provisioning needs of the cluster. The cloud provisioning and cluster management module is a modified version of TIRAMOLA. As stated in section (**Error! Reference source not found.**), the modifications to the Hadoop code are transparent to the user. The modified Hadoop framework is designed to analyze incoming MapReduce jobs, collect metadata about them (i.e. number of input splits, number of reducers to be run, type of MapReduce program) and emit them to TIRAMOLA. TIRAMOLA is responsible for processing the metadata and the constraints the user has set and finally make a decision about whether to expand the working cluster by adding a number of nodes – and if yes, how many – or not; TIRAMOLA is then expected to act on the decision it reached and, when the acquired resources – if any – are available, notify the Hadoop framework to start the execution of the submitted job.

After the program submitted has been executed and if the cluster stays in an idle state for more than a specified time, TIRAMOLA starts releasing nodes – until it shrinks to a minimum cluster size.

Hadoop has also been optimized to improve the reduce phase cluster utilization, taking under consideration the fact that many jobs will be executed in an expanded cluster compared to the one they were submitted to. By default, MapReduce programs executed in Hadoop are set to use only one `ReduceTask`. Of course, a multi-node cluster using just one reducer would be waste of resources, as all worker nodes but one would remain idle during the reduce phase.

Typically the number of reducers is set by the user. That policy, however, implies that the user is an expert, fully aware of the specifics of

a. The cluster,

b. The program being executed and

c. The data being processed.

In our case, none of these conditions is true. So, instead of relying on the user to select the number of `ReduceTasks` to be run, our system binds the number of reduce tasks to be launched with the number of active nodes in the – now expanded – cluster by overriding the default and replacing it with a number of approximately twice the total capacity of the cluster in `ReduceTask` containers[5] [31]. This way we can ensure that there will be as few resources during the reduce phase as possible.

Last but not least, the decision making module of TIRAMOLA can be considered as a separate module, as it is pluggable and can be modified according to the user's needs. In order to test our system, we decided to work on an implementation of the work by Tian and Chen[12]. The two authors suggest using a simplified model and regression analysis to produce a cost function for the prediction of execution time of MapReduce programs.

In the following sections, we will present the architecture of our BBQ Hadoop system and explain what modifications were needed for its implementation to the original Hadoop/YARN and TIRAMOLA versions. We will also expose the interactions between the various modules and how these bring us the desired result.

7.1 BBQ architecture

As explained in the introductory section, a rough overview of the BBQ system would focus on three main modules:

- A modified – cloud aware – version of Hadoop
- A modified – job aware – version of TIRAMOLA
- A pluggable Decision Making module, based on which TIRAMOLA will make the provisioning decisions.

The TIRAMOLA Coordinator can be viewed an intermediate layer between the BBQ flavor Hadoop cluster and its own Decision Making module, as it stands in the center of all interactions, as seen below:

[5] For this we have assumed that our cloud based cluster consists of virtual machines of the same size in terms of resources. This is the typical case in a cloud based cluster.

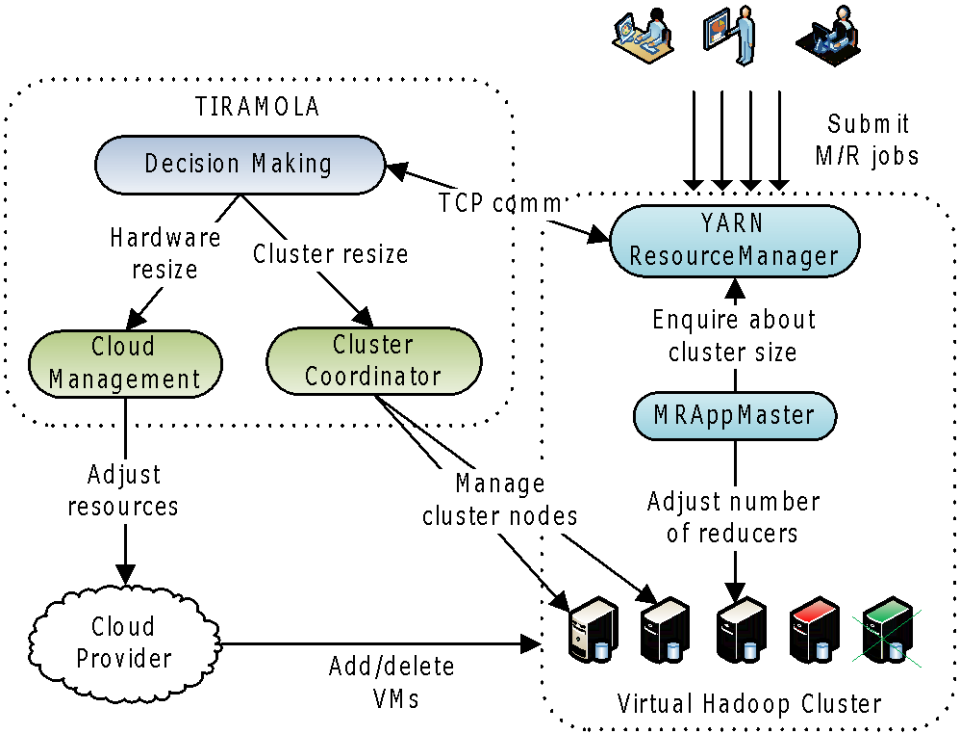

Figure 7-1. BBQ System Architecture

BBQ Hadoop can itself be put under the microscope and analyzed as a collection of actors, interacting with each other and with TIRAMOLA.

7.1.1 Module Interaction - Integration

In order for TIRAMOLA and Hadoop to eventually be integrated, we have had to design a communication protocol. We chose to implement a TCP socket connection through which the metadata required by TIRAMOLA would be shipped, in json format [32].

The transmitted message is of the form:

```
{maps: <number_of_MapTasks>, reducers:<number_of_ReduceTasks>}.
```

7.1.2 BBQ Hadoop

In section (5.3), we stated that finding a way to extract job-specific metadata was the first task in mind when designing the BBQ system. More specifically, the information that needs to be obtained before the job execution begins is the number of map and reduce tasks which are going to be launched. As stated earlier, the number of map tasks cannot be user-defined: it depends on the input – the data the user desires to process – and is equal to the number of the splits in which it is partitioned by the Hadoop framework. On the other hand, the number of

reduce tasks can be set by the user. For our implementation, we have chosen to give the user the option to let the BBQ flavored Hadoop to adjust the number of reducers to be executed by entering a negative value to the corresponding attribute while running the job through YARN.

Following through on our observations recorded in section (5.3), we examined the Hadoop MapReduce execution pipeline very closely and noticed that the number of input splits is initially calculated during the execution of the `Job.submit()`[6] method, which submits the program to the YARN platform. We also noticed that during the job submission process an `ApplicationSubmissionContext`[7] object is also created, in which various metadata about the program are stored. The `ApplicationSubmissionContext` object is then passed on to `RMAppManager`, a class which manages the list of applications for the YARN `ResourceManager`. Running a method called `submitApplication`, the `RMAppManager` processes the metadata stored in `ApplicationSubmissionContext` and then starts the Application.

Our first expansion of the original Hadoop source code took place here. Initially, we parsed the additional metadata required into `ApplicationSubmissionContext`. The code we modified resides in the `YARNRunner` class. We then modified the `RMAppManager.submitApplication()` method so that it extracts the aforementioned data from the `ApplicationSubmitionContext` object. We added a new class to the Hadoop source code, implementing a blocking TCP client, which also parses the metadata into json format, connects to the TCP server run by TIRAMOLA. The `RMAppManager.submitApplication()` uses an instance of this class to emit the metadata it has collected to TIRAMOLA, and waits for an answer from the server. When a message of success is received – meaning the cluster has been successfully expanded by a number of nodes or remained unchanged, then the socket unblocks and the execution continues. In the case where no successful expansion could be completed, then, again, the socket unblocks and the execution continues with the resources currently available.

These changes conclude the expansion required to implement cluster elasticity. In Figure 7-2 we present an overview of the submission process. Highlighted in color are the classes or methods of interest which were modified. In Figure 7-4, we have similarly highlighted the `RMAppManager.submitApplication()` method which was altered.

[6] https://hadoop.apache.org/docs/stable2/api/org/apache/hadoop/mapreduce/Job.html
[7] https://hadoop.apache.org/docs/stable2/api/org/apache/hadoop/yarn/api/records/ApplicationSubmission Context.html

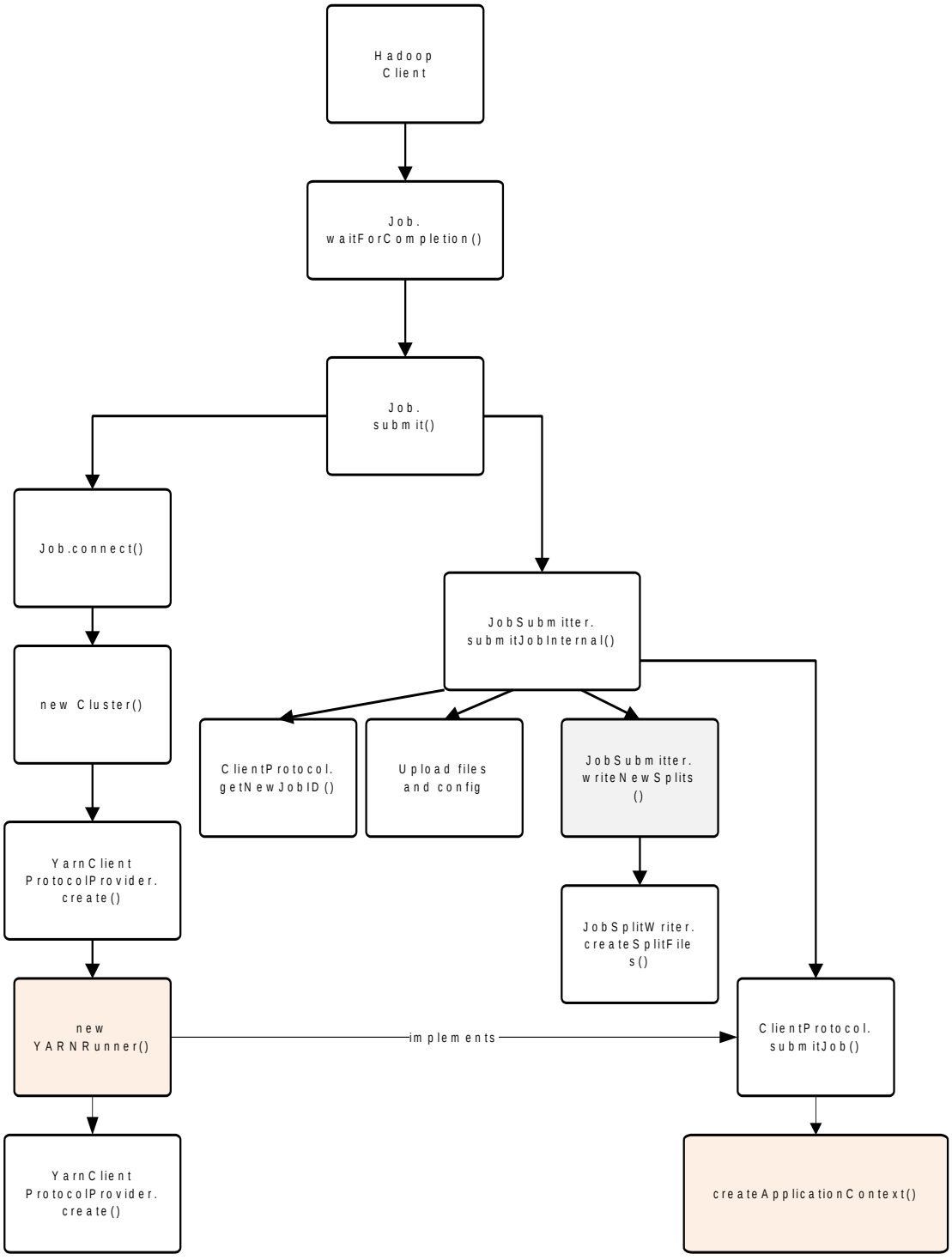

Figure 7-2. Job submission

In order implement the reducer phase cluster utilization optimization described above, we had to study closely the MapReduce framework of Hadoop. In Hadoop MapReduce, the number of `ReduceTasks` to be spawn needs to be known at the same time as the number of `MapTasks` is, since the latter need that information to complete the partitioning phase as seen in section (5.2.3.1). This means that any adjustments must be made before the execution of the `MapTasks` begins but obviously after the cluster size adjustments have been completed,

since our implementation binds the number of Reduce tasks to be launched to the number of nodes in the cluster. This presents a significant challenge, because the number of the reducers is by default static in the Hadoop MapReduce framework and is not supposed to be decided during execution time. To make sure that the correct number of `ReduceTasks` will be set, the relative information will have to be extracted from the `ResourceManager` – the only module of YARN which has full awareness of the cluster.

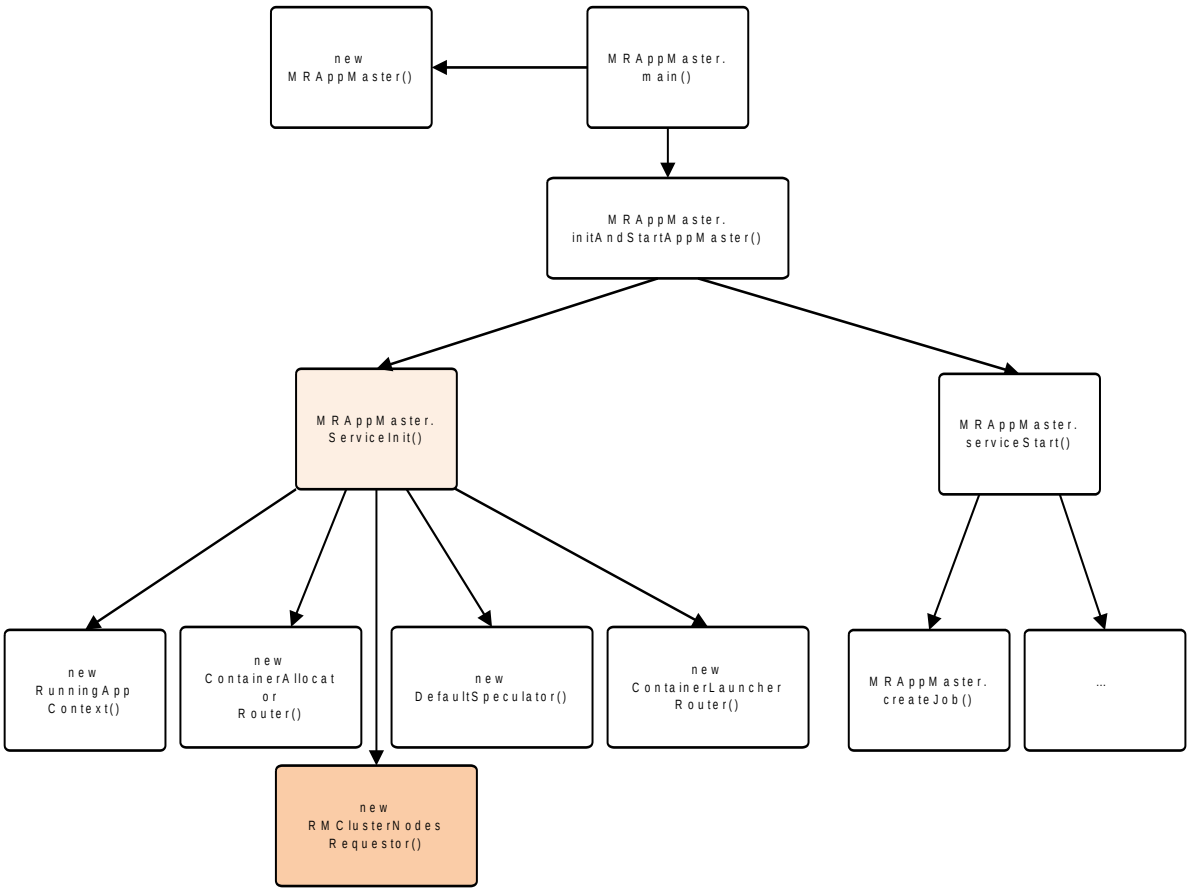

Figure 7-3. MRAppMaster

As explained in paragraph (5.2.1), when a new YARN Application is launched, the `ResourceManager` reserves a container for its `ApplicationMaster` – or `MRAppMaster` for MapReduce jobs – which is responsible for the communication between the MapReduce framework and the YARN Server. During the initialization stage a MapReduce Application, an HDFS staging directory where the job configuration files and other temporary data are stored is created. Every new task that spawns, even the `MRAppMaster`, creates a local copy of that staging directory in order to have quick access to the job configuration[8]. Since the `ApplicationMaster` occupies the first container leased and is launched before any other

[8]https://support.pivotal.io/hc/en-us/articles/201925118-How-to-find-and-review-logs-for-yarn-mapreduce-jobs - as described in Pivotal's support webpage. Pivotal is an Enterprise Hadoop developper

Tasks start running, we expanded its initialization function – `serviceInit()` – by allowing it to reach into the HDFS staging directory and overwrite the setting that corresponds to the number of `ReducerTasks` to be launched. This way, all the `MapTask` instances that will spawn later, will have access to the modified configuration file.

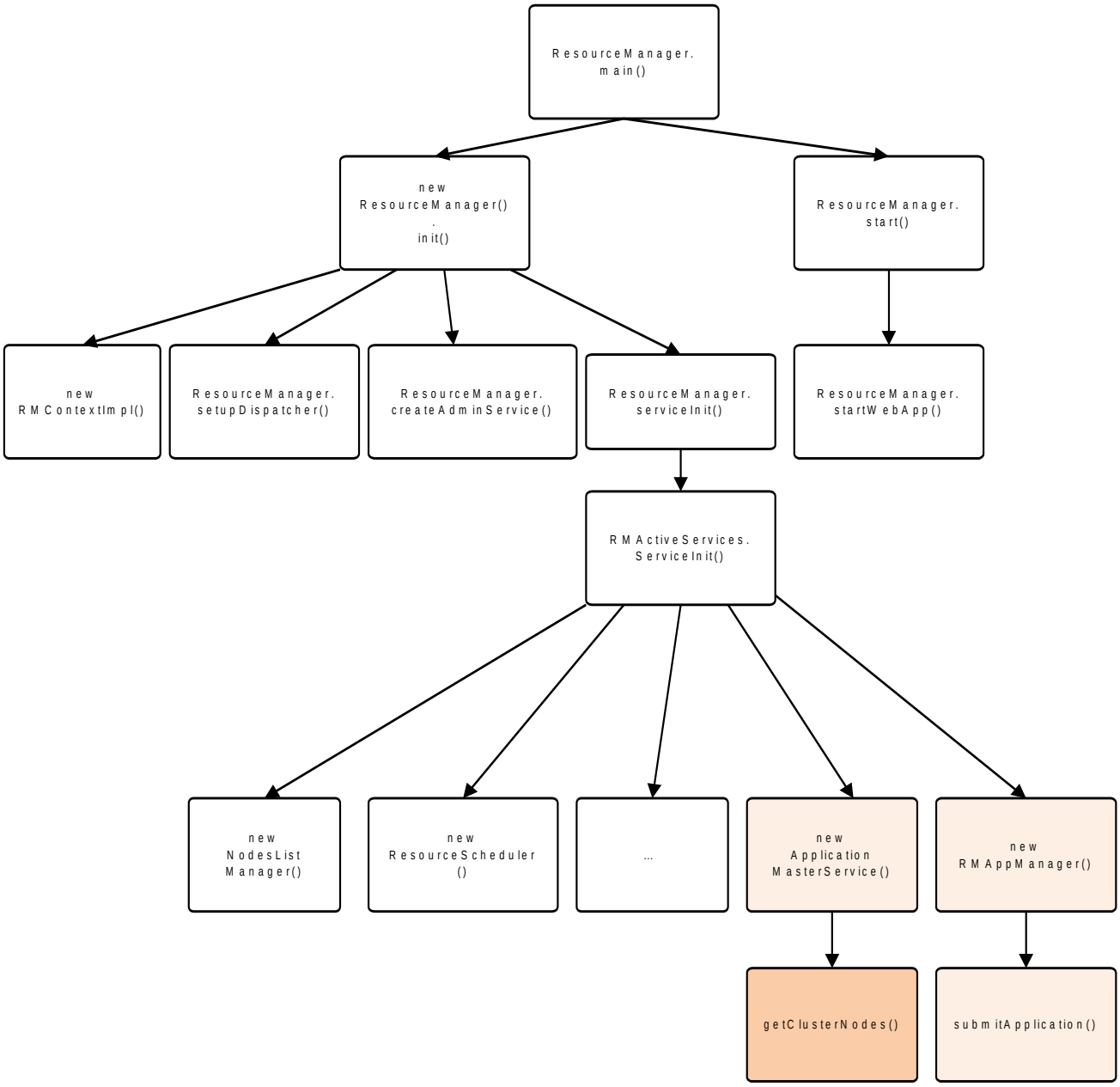

Figure 7-4 ResourceManager

In order to complete this second expansion of the Hadoop platform, though, we also needed to create a way to make the `MRAppMaster` cluster aware so that it can calculate the optimized number of `ReduceTasks` to be launched and replace with it the default value. The core of this optimization is the dynamic binding of the number of `ReduceTasks` to the number of

active nodes in the cluster. As highlighted in section (5.2.1), `ApplicationMasters` are completely ignorant of the cluster infrastructure. The YARN module which possesses this kind of information is the `ResourceManager`.

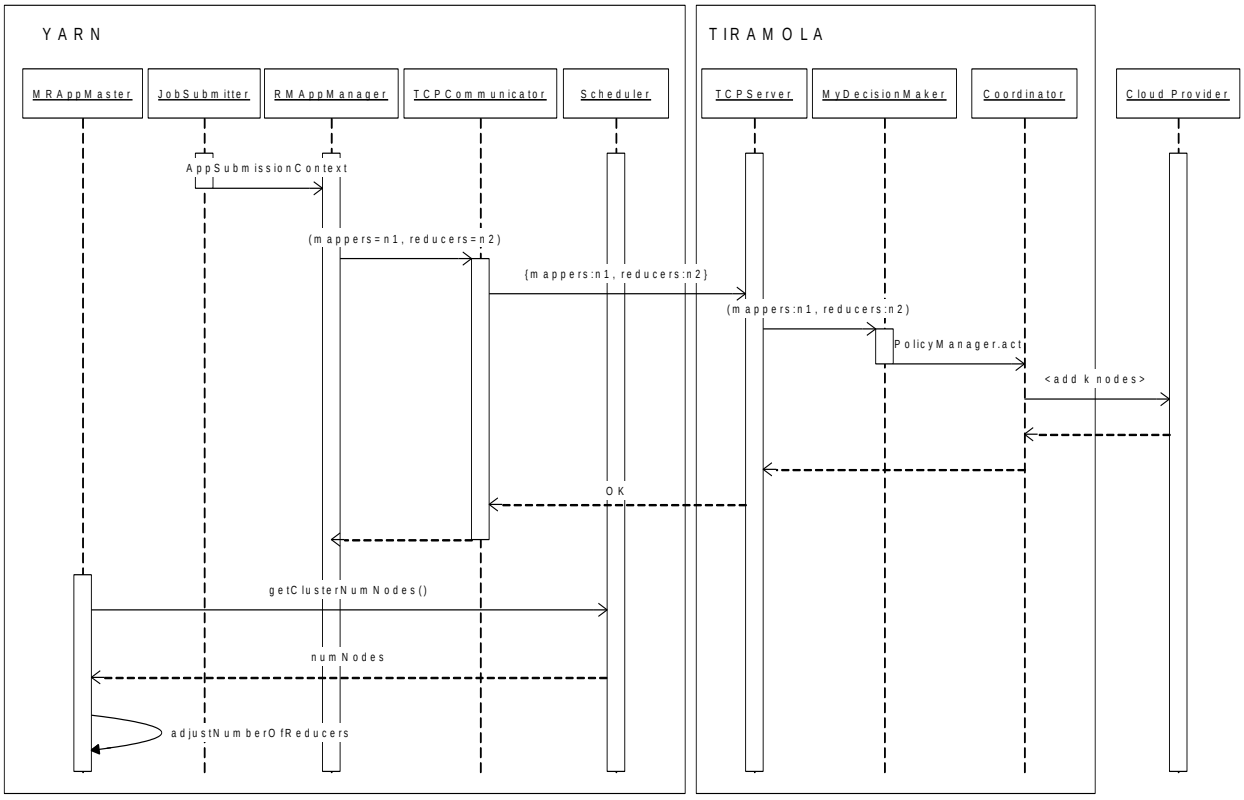

Figure 7-5. Resource calculation and adjustment Sequence Diagram

To render the `MRAppMaster` cluster aware, we expanded the communication protocol between the `MRAppMaster` and the `ResourceManager's ApplicationMasterService` by implementing a class named `RMClusterNodesRequestor`, which submits a request about the number of active nodes in the cluster to the `ResourceManager` and returns that number as a result as well as a number of Abstract classes required by the protocol design. This enquiry also takes place in the `MRAppMaster` initialization method, `serviceInit()` and precedes what was described in the previous paragraph. In Figure 7-3 we can see the architectural overview of `MRAppMaster`. `RMClusterNodesRequestor()` has been implemented from scratch, while the `AppMaster.serviceInit()` method has been modified to include the new class. The `getClusterNodes()` method, which extracts the information required from the `ResourceManager`, has been added to the `ApplicationMasterService` class as seen in Figure 7-4.

In Figure 7-5, we present an overview of all actors within the BBQ system and their interactions during the execution of a MapReduce job on a Hadoop YARN cluster.

7.2 Cost function and resource calculation algorithm

The resource calculation algorithm has been based on an implemented of a cost function built as presented in the work of Fengguang Tian, Keke Chen[12]. The authors suggest that the execution time of a MapReduce1.0 program can be modeled by a cost function of three variables, *M* – the size of the input, calculated in input splits, *m* – the number of map slots and *R* – the total number of reduce tasks.

$$T(M, m, R)$$

Translating the variables above into Hadoop2.0+ vocabulary, *m* would be the capacity of the cluster in map task containers. Since we have selected to modify the number of `ReduceTasks` as we previously described in (7) and since cloud clusters typically consist of identical – in terms of resources – nodes, *R* can also be derived from the number of active nodes in the cluster, *n.* The capacity of the cluster in `MapTasks`, *m,* is also a product of *n*, it follows that:

$$T(M, m, R) = T(M, n)$$

In the virtual machine (VM) based cloud infrastructure (e.g., Amazon EC2), the cost of cloud resources is calculated based on the number of VM instances used in time units (typically in hours). According to the capacity of a virtual machine (CPU cores, memory, disk and network bandwidth), a virtual node can only run a fixed number of Map/Reduce containers. For simplicity, we have chosen to assign map and reduce task containers the same resources – which means that the total capacity of the cluster in MapReduce Task containers is *kn* where *k* the number of containers per node.

If the price of renting one VM instance for an hour is *u*, the total financial cost is determined by the result *unT(M,n).* Therefore, given a financial budget φ, the problem of finding the best resource allocation to minimize the job time can be formulated as

minimize T(M,n)
subject to unT(M,n) $\leq \varphi$,
$m > 0$, and $R > 0$.

If the constraint is about the time deadline τ for finishing the job, the problem of minimizing the financial cost can be formulated as

minimize unT(M,n)
subject to T(M,n) $\leq \tau$,*m > 0,* and *R > 0.*

In the paper, the execution time of a MapReduce program can be derived by the following model. To calculate the desired parameters we can use the method of linear regression

$$T(M,m,R) = \beta_0 + \beta_1 \frac{M}{m} + \beta_2 \frac{M}{R} + \beta_3 \frac{M}{R} log\left(\frac{M}{R}\right) + \beta_4 M + \beta_5 \boldsymbol{R} + \boldsymbol{\varepsilon}$$

Modifying the function above to suit our needs, we get the cost function:

$$T(M,n) = \beta_0 + \beta_6 \frac{M}{n} + \beta_3 \frac{M}{n} log\left(\frac{M}{R}\right) + \beta_4 M + \beta_5 \boldsymbol{n} + \boldsymbol{\varepsilon},$$

$$\boldsymbol{where\ n = the\ (updated) number\ of\ nodes\ in\ the\ cluster}.$$

In order to calculate the values of β_i for a specific MapReduce program we ran multiple jobs changing the two variables involved in every experiment (*input size* and *number of nodes* in the cluster) and recorded the execution times.

Once the cost function of a program has been built we can use it to solve the optimization problems such as calculating the optimal amount of resources that can minimize the financial cost with a time deadline or minimize the time under certain financial budget.

The experiments conducted are presented in detail in the following chapter.

7.3 BBQ Execution Overview

To wrap up our presentation of the functionality of the BBQ system, we present an execution overview of a MapReduce job.

When a new job arrives for execution, the system performs the following steps:

TIRAMOLA's main class now runs a blocking TCP server, waiting for incoming connections from YARN's `RMAppManager`.

YARN's `JobSubmitter` class has also been modified. This is the class that sets up the configuration for the job that is about to be executed and then passes it to YARN. It is responsible for reading the input data and creating an `ApplicationSubmissionContext` instance. Through it, the metadata required can now be passed to the YARN Resource Manager.

YARN's `RMAppManager` class has been modified so that it runs a blocking TCP client. This is the class that reads the information provided by `JobSubmitter`. This class also has access

to metrics of the Hadoop cluster. Every time a new job is submitted for execution, the `RMAppManager` reads the input data size and the TCP client sends a message including the metadata needed by the decision making module, serialized in json format, to TIRAMOLA's server. Then it waits for TIRAMOLA's response.

TIRAMOLA's Coordinator receives the message and passes the information from the `RMAppManager` on to the `takeDecision` method from TIRAMOLA's `DecisionMaker` class as an argument. `takeDecision` uses the algorithm presented in (7.2) to decide the number of virtual machines to be deployed (if there is such need) – in order to respect the user's restrictions, initializes and launches these additional VMs.

When the required number of nodes has finally been added to the cluster, the Coordinator sends a message to `RMAppManager's` client, unblocking the map-reduce job, which starts running on the expanded cluster. The application is now being executed. During the initialization of the `MRAppMaster`, the `ApplicationMaster` for MapReduce applications, the Hadoop framework will now request information from the YARN Scheduler about the size of the cluster in order to automatically adjust the number of the reducer instances to be spawn. This number should not be significantly smaller than the total capacity of the cluster in Containers, as that could cause a very small part of the now expanded cluster to be utilized. Then, it will reach in the HDFS staging directory of the job and accordingly modify the job.xml file, where the job configuration is stored. The execution pipeline will continue as usual until the MapReduce job is completed.

Since the additional nodes are deployed in order to enhance the cluster's computing power, they are not needed when there are no MapReduce jobs running. Therefore, if Coordinator's server does not receive an incoming connection for a specified amount of time, a timeout exception is thrown. The exception causes `DecisionMaker` to launch `takeDecision`, only to release a node this time - unless the number of nodes in the cluster already is the minimum allowed.

8 Experimental Results

8.1 Experimental Setup

8.1.1 Hardware Setup and Software Used

The experiments are conducted in the CSlab private OpenStack [33] cloud. Each node had four processors, 4GB memory, and a 300GB hard drive. The virtual machines run on Ubuntu 14.04LTS operating system. The version 2.6.0 of Hadoop is installed in the cluster, running over Oracle Java 1.8.0.2. One node serves as the master node and the other as the slave nodes. TIRAMOLA runs on a separate machine, with 1 processor, 2GB memory and a 10GB hard drive. The version of TIRAMOLA used is written in Python3.4. The single master node runs theResourceManager, while each slave node run NodeManagers.

8.1.2 Hadoop Memory Configuration

Memory allocation has been made according to Hortonworks' suggestions[29].

YARN and MapReduce settings were set as follows:

Variable	Definition	Value
`yarn.scheduler.minimum-allocation-mb`	Minimum memory allocation for a single container	512
`yarn.scheduler.maximum-allocation-mb`	Maximum memory allocation for a single container	3072
`mapreduce.map.memory.mb`	Memory allocation for map task containers	1024

`mapreduce.reduce.memory.mb`	Memory allocation for reduce containers	1024

The settings above allow every machine to run at most 3 MapReduce containers at the same time. As mentioned in (7), we have selected to run a number of approximately

$$1.8 * (\text{max\_number\_of\_active\_containers}) = 1.8 * 3 * \text{number\_of\_nodes}$$

reduce tasks.

8.1.3 Datasets Used

To run the tests we used 3 sample datasets from Wikipedia dumps. With the following sizes:

- 4.3GB
- 23.8GB
- 50.2GB

8.1.4 Benchmark Used

The tests were run using the WordCount benchmark. WordCount is a sample MapReduce program in the Hadoop package. The Map function splits the input text into words and the result is locally aggregated by word with a Combiner; the Reduce function sums up the local aggregation results <word, count> by words and output the final word counts.

8.2 Experiments

8.2.1 Model Construction for the WordCount Benchmark

Initially, a number of jobs were run using various permutations of (M, n), M being the total number of splits of the input datasets and n the number of active nodes on the cluster. The jobs were timed and the results are displayed in the table below:

Exp No	Input Data Size (GB)	Input Data Size (HDFS blocks)	Number of active nodes	Time (minutes)
1	4.3	17	2	13,15
2	4.3	17	3	8,45
3	24	93	3	56,63

4	24	93	4	30,53
5	49	196	4	87,05

Using linear regression to analyze our data, we manually construct a model for the execution time as a function of input data size (M) and the active number of nodes (n) in the cluster.

The model created is

$$T(M,n) = \frac{1}{10}\left(6.729\frac{M}{n} + 4.691\frac{M}{n}(ln\left(0.186\frac{M}{n}\right) + 45.685)\right)$$

The time cost function for WordCount shows us that the expected execution time is a function of the fraction of the size of the input over the number of active nodes in the cluster.

In the following table we can compare the actual execution times of our experiments against the model's estimation:

Exp No	**Input Data Size (HDFS blocks)**	**Number of active nodes**	**Actual Time (minutes)**	**Model Prediction Time (minutes)**
1	17	2	13,15	12,097
2	17	3	8,45	8,5095
3	93	3	56,63	50,840
4	93	4	30,53	36,134
5	196	4	87,05	88,230

The graph below shows that the model does indeed match the real time measurements to a satisfying degree – also, using R^2 as a measure of goodness of fit [34], we get a very acceptable measure of 0,98%.

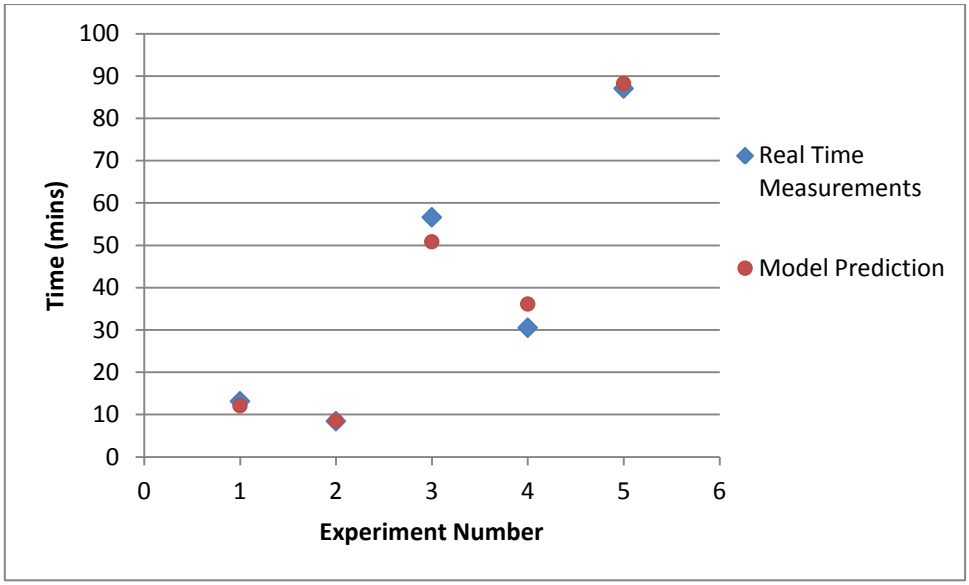

Figure 8-1. Model building.

We then proceed to test our results and constructed model by executing some new WordCount operations on different datasets.

8.2.2 Model Evaluation

For the evaluation of our mode, we give our BBQ cluster an ***expected execution time*** for various values of input data size and initial number of nodes, with *no restriction to financial cost*. The BBQ system is asked to execute the requested program within the set time limits, which means that it will have to automatically manage its resources in order to achieve the execution times requested.

Using similar datasets, we test our model:

Exp No	**Input Data Size (HDFS blocks) – M**	**Number of starting active nodes - n**	**Number of nodes added**	**Actual Time (minutes)**	**Maximum Requested Time (minutes)**
6	27	2	+0	19.21	30
7	43	3	+1	18.17	20
8	43	2	+2	19.83	20
9	71	1	+2	43.15	40

We notice that the results are acceptable, despite some expected overhead caused by the spawning and initialization of the new nodes. In one case the overhead causes the execution to exceed the maximum requested time but only for a few minutes (<10% of the total actual time). This can be avoided by adding a constant factor to the Time cost function presented above.

8.2.3 Observations

There are a couple issues which are raised concerning the use of BBQ. First of all, there is the overhead caused by the spawning and initialization of the new nodes, as mentioned above. According to Mao et al. [35], VM startup time can be considered of trivial cost and should not cause substantial problems. The authors state that launching VMs in a cloud environment should not be a bottleneck, as long as the precooked images of their hard drives are not extremely large. In our case these images only include an installation of Ubuntu and Hadoop.

A second issue could be the one of data locality. Newly spawn nodes do not possess local copies of the data which they will be asked to process. Hadoop2.0 tries to make use of data locality whenever possible, so this situation could cause an amount of considerable delay. According to Ananthanarayanan et al. [36], data locality – in terms of disk locality – tends to be trivial in datacenter environments, as disk operations remain slow while, on the contrary, data are able to move around the cluster faster than ever. This means that the value we gain by adding processing power to our cluster can completely overshadow the data transfer overhead.

9 *Conclusions*

We have successfully proved the fact that Hadoop MapReduce can run on an elastic infrastructure with necessary changes and additions to the initial Hadoop source code. We managed to automate cloud provisioning which now happens on-the-fly, without the user having to worry about the setup of the infrastructure or the architecture of it. The experimental results have confirmed that such a service could be useful and profitable if offered to users.

The system we are introducing, taking advantage of the facts that

a. the Hadoop MapReduce framework splits the input data before it starts executing a program and
b. the YARN Resource Manager is the only module aware of the cluster and the single blocking point of failure for the framework,

collects and transmits metrics specific to the application which is executed to the TIRAMOLA cluster provisioning module.

Taking into consideration that the time for the launch of a large number of virtual machines is not significantly larger than that of a single one, according to Mao et al [35], BBQ simple way to boost execution time by utilizing a cloud's resources in a responsible way. The simplicity of BBQ's design and architecture could make it an ideal platform for the execution of independent MapReduce Jobs – in comparison to job workflows which other systems may support. To establish the utility value of the BBQ platform we will present a real use case example in the following section.

9.1 Fit for Lambda architecture – a use case example

In this example we will present the BBQ system as a part of a lambda architecture model. Lambda architecture [37] is a data-processing architecture designed to deal with massive quantities of structured or unstructured data by taking advantage of both batch- and stream-processing methods. This approach to architecture attempts to balance latency, throughput, and fault-tolerance by using batch processing to provide comprehensive and accurate views of historical data, while simultaneously using real-time stream processing to provide views of online data. The two view outputs can be joined before presentation providing the user with a complete picture of the data in store. The rise of lambda architecture is correlated with the growth of big data, real-time analytics, and the drive to mitigate the latencies of MapReduce.

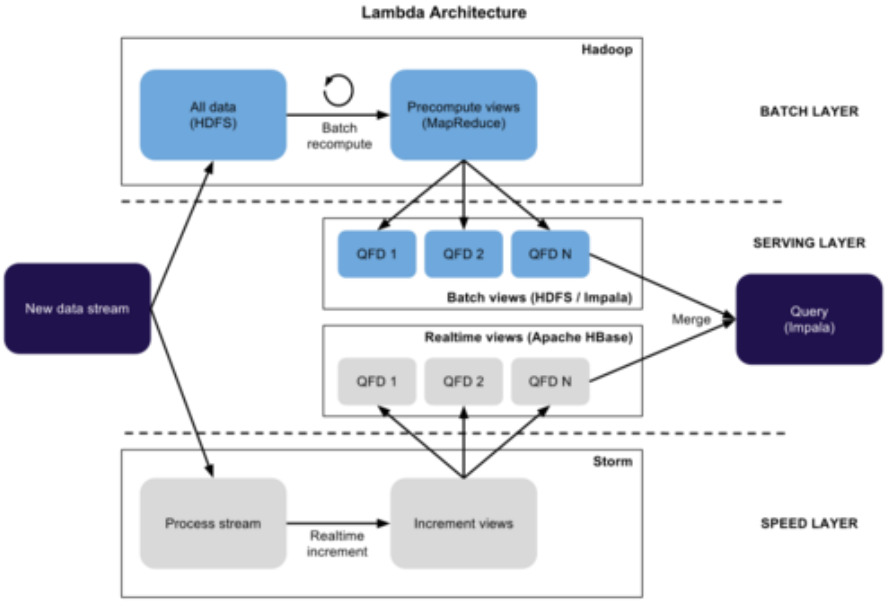

Figure 9-1. Lambda architecture.

In Figure 9-1. Lambda architecture., we can see an overview of a Lambda architecture based system and some of the technologies which can be used for its implementation. In general the Lambda Architecture is composed of three layers: the batch layer, the serving layer and the speed layer.

The batch layer contains the immutable, constantly growing, append-only master dataset stored on a distributed file system like HDFS [38]. With batch processing (MapReduce) batch views are computed from this raw dataset. Hadoop is a perfect fit for the concept of the batch layer.

The task of the serving layer is to load and expose the batch views in a datastore so that they can be queried. This serving layer datastore does not require random writes – but must

support batch updates and random reads – and can therefore be extraordinarily simple (candidates could be ElephantDB [39], Impala [40] or Voldemort [41]).

The speed layer deals only with new data not accounted for in the batch processing. It compensates for the high latency updates of the serving layer by being implemented on top of stream processing systems (Storm [42], S4 [43], Spark [44]) and random read/write datastores to compute the realtime views (HBase [8]). These views remain valid until the data have been handed over and processed by the batch and serving layer. To get a complete result, the batch and realtime views must be queried and the results merged together.

Now, suppose an organization using an implementation of Lambda architecture decides to select BBQ Hadoop as the platform on which the batch layer is built. Since batch operations are expensive and only take place in regular time windows, they could use a cloud storage service implementing an HDFS file system to save their data and then, by only keeping a server running all the Hadoop and TIRAMOLA master services alive – i.e. `ResourceManager`, `Coordinator` – they could efficiently produce the batch views required, allowing the system to automatically spawn and add processing nodes to the cluster whenever such an operation is required to be executed, while keeping its execution time under control. In fact, the dedicated BBQ master server can be hosted in a local machine, thus further reducing the running costs of the organization.

9.2 Future work

The system that was designed and implemented for the needs of this diploma thesis can be used as a basis for future expansion. In this thesis we have assumed that our cloud-provided clusters only have an active job to execute. BBQ can however be further expanded to take into account cluster load and scheduling policies. Relevant job metadata can be extracted from YARN and be passed on to TIRAMOLA. This could prove to a very interesting topic for future research, as existing solutions tend to treat Hadoop clusters as one-purpose entities, an idea which comes in contrast with the powerful impact the integration of YARN has made. BBQ Hadoop has been designed as a permanent infrastructure layer for big data processing, which means there should be cases where large numbers of jobs are submitted to the cluster. It is interesting, in this case to examine scheduling and load management challenges which the option of expanding an existing cluster creates.

10 *Works Cited*

[1] T. Hey, S. Tansley, and K. Tolle, "The Fourth Paradigm: Data-Intensive Scientific Discovery, 2009," *Microsoft Res.*, 2009.

[2] IBM, "IBM - What is big data?" [Online]. Available: http://www-01.ibm.com/software/data/bigdata/what-is-big-data.html. [Accessed: 12-May-2015].

[3] Apache Hadoop, "Apache Hadoop." [Online]. Available: https://hadoop.apache.org/. [Accessed: 12-May-2015].

[4] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[5] Apache Mahout, "Apache Mahout: Scalable machine learning and data mining." [Online]. Available: http://mahout.apache.org/. [Accessed: 12-May-2015].

[6] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1626–1629, 2009.

[7] Apache Pig, "Apache Pig." [Online]. Available: https://pig.apache.org/. [Accessed: 12-May-2015].

[8] Apache HBase, "Apache HBase." [Online]. Available: http://hbase.apache.org/. [Accessed: 12-May-2015].

[9] Amazon, "AWS | Amazon Elastic MapReduce (EMR) | Hadoop MapReduce in the Cloud." [Online]. Available: http://aws.amazon.com/elasticmapreduce/. [Accessed: 12-May-2015].

[10] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, and others, "Apache Hadoop Yarn: Yet Another Resource Negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*, 2013, p. 5.

[11] I. Konstantinou, E. Angelou, D. Tsoumakos, C. Boumpouka, N. Koziris, and S. Sioutas, "Tiramola: elastic nosql provisioning through a cloud management platform," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012, pp. 725–728.

[12] F. Tian and K. Chen, "Towards optimal resource provisioning for running mapreduce programs in public clouds," in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, 2011, pp. 155–162.

[13] F. Gabbay and A. Mendelson, "Can program profiling support value prediction?," in *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on*, 1997, pp. 270–280.

[14] P. Mell and T. Grance, "The NIST definition of cloud computing," 2011.

[15] N. R. Herbst, S. Kounev, and R. Reussner, “Elasticity in Cloud Computing: What It Is, and What It Is Not.,” in *ICAC*, 2013, pp. 23–27.
[16] M. Cox and D. Ellsworth, “Application-controlled demand paging for out-of-core visualization,” in *Proceedings of the 8th conference on Visualization’97*, 1997, p. 235–ff.
[17] R. Bryant, R. H. Katz, and E. D. Lazowska, *Big-data computing: creating revolutionary breakthroughs in commerce, science and society*. December, 2008.
[18] D. Laney, “3D data management: Controlling data volume, velocity and variety,” *META Group Res. Note*, vol. 6, 2001.
[19] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The Google file system,” in *ACM SIGOPS operating systems review*, 2003, vol. 37, pp. 29–43.
[20] Amazon, “AWS | Amazon Elastic Compute Cloud (EC2) - Scalable Cloud Hosting.” [Online]. Available: http://aws.amazon.com/ec2/. [Accessed: 12-May-2015].
[21] Amazon, “AWS | Amazon Simple Storage Service (S3) - Online Cloud Storage for Data & Files.” [Online]. Available: http://aws.amazon.com/s3/. [Accessed: 12-May-2015].
[22] C. W. Olofson and D. Vesset, *Worldwide Hadoop-MapReduce ecosystem software 2012–2016 forecast*. May, 2012.
[23] Amazon, “Resize a Running Cluster - Amazon Elastic MapReduce.” [Online]. Available: http://docs.aws.amazon.com/ElasticMapReduce/latest/DeveloperGuide/emr-manage-resize.html. [Accessed: 14-May-2015].
[24] K. Kambatla, A. Pathak, and H. Pucha, *Towards optimizing hadoop provisioning in the cloud*. HotCloud, 2009.
[25] S. Babu, “Towards automatic optimization of MapReduce programs,” in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 137–142.
[26] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu, “Starfish: A Self-tuning System for Big Data Analytics.,” in *CIDR*, 2011, vol. 11, pp. 261–272.
[27] H. Herodotou, F. Dong, and S. Babu, “No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics,” in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, 2011, p. 18.
[28] A. Verma, L. Cherkasova, and R. H. Campbell, “Resource provisioning framework for mapreduce jobs with performance goals,” in *Middleware 2011*, Springer, 2011, pp. 165–186.
[29] Hortonworks, “Determine YARN and MapReduce Memory Configuration Settings - Hortonworks Data Platform.” [Online]. Available: http://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.0.9.1/bk_installing_manually_book/content/rpm-chap1-11.html. [Accessed: 12-May-2015].
[30] E. Coppa, “Hadoop Internals.” [Online]. Available: http://ercoppa.github.io/HadoopInternals/. [Accessed: 24-May-2015].
[31] Apache, “HowManyMapsAndReduces - Hadoop Wiki.” [Online]. Available: http://wiki.apache.org/hadoop/HowManyMapsAndReduces?action=recall&rev=7. [Accessed: 19-May-2015].
[32] T. Bray, “The JavaScript Object Notation (JSON) Data Interchange Format,” 2014.
[33] OpenStack, “OpenStack Open Source Cloud Computing Software.” [Online]. Available: https://www.openstack.org/. [Accessed: 24-May-2015].
[34] A. C. Cameron and F. A. Windmeijer, “An R-squared measure of goodness of fit for some common nonlinear regression models,” *J. Econom.*, vol. 77, no. 2, pp. 329–342, 1997.
[35] M. Mao and M. Humphrey, “A performance study on the vm startup time in the cloud,” in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, 2012, pp. 423–430.
[36] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, “Disk-locality in datacenter computing considered irrelevant,” 2011.
[37] “Lambda Architecture » λ lambda-architecture.net.” [Online]. Available: http://lambda-architecture.net/. [Accessed: 02-Jun-2015].
[38] D. Borthakur, “HDFS architecture guide,” *Hadoop Apache Proj.*, p. 53, 2008.

[39] “nathanmarz/elephantdb,” *GitHub*. [Online]. Available: https://github.com/nathanmarz/elephantdb. [Accessed: 02-Jun-2015].
[40] Cloudera, “Impala.” [Online]. Available: http://impala.io/. [Accessed: 02-Jun-2015].
[41] “Voldemort.” [Online]. Available: http://www.project-voldemort.com/voldemort/. [Accessed: 02-Jun-2015].
[42] Apache, “Storm, distributed and fault-tolerant realtime computation.” [Online]. Available: https://storm.apache.org/. [Accessed: 02-Jun-2015].
[43] Apache, “S4: Distributed Stream Computing Platform.” [Online]. Available: http://incubator.apache.org/s4/. [Accessed: 02-Jun-2015].
[44] Apache, “Apache Spark™ - Lightning-Fast Cluster Computing.” [Online]. Available: https://spark.apache.org/. [Accessed: 02-Jun-2015].