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

Ενσωμάτωση αντικειμενοστραφών μέσων αποθήκευσης σε πλατφόρμα διαμοιρασμού δεδομένων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Χρήστος Ι. Ιωαννίδης

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Νοέμβριος 2015

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

Ενσωμάτωση αντικειμενοστραφών μέσων αποθήκευσης σε πλατφόρμα διαμοιρασμού δεδομένων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Χρήστος Ι. Ιωαννίδης

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή τη 12η Νοεμβρίου 2015.

.....	.....	.....
Νεκτάριος Κοζύρης Καθηγητής Ε.Μ.Π.	Νικόλαος Παπασπύρου Αναπληρωτής Καθηγητής Ε.Μ.Π.	Γεώργιος Γκούμας Λέκτορας Ε.Μ.Π.

Αθήνα, Νοέμβριος 2015

.....

Χρήστος Ιωαννίδης

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

Περίληψη

Επί του παρόντος, οι συμμέτοχοι στο πεδίο των Big Data αντιμετωπίζουν πολλές δυσκολίες όταν η συνεργασία μεταξύ τους είναι απαραίτητη. Απαιτούνται νέοι τρόποι συνεργασίας, με τους οποίους η συνεργασία θα είναι πολύ ευκολότερη και οι οποίοι θα λειτουργούν αδιάλειπτα. Καθώς το αντικειμενοστραφές μοντέλο προγραμματισμού είναι το πιο κυρίαρχο, παρατηρείται η διείσδυσή του και στο πεδίο των Big Data. Επιπλέον, και το υλικό (hardware) ακολουθεί αυτή την τάση του αντικειμενοστραφούς μοντέλου, καθώς παρουσιάζονται πλέον νέες τεχνολογίες αποθήκευσης οι οποίες πραγματοποιούν τη διαχείριση δεδομένων χρησιμοποιώντας την αφαιρετικότητα των αντικειμένων key-value. Επομένως, η προσπάθεια να συνδυαστούν το αντικειμενοστραφές λογισμικό και υλικό χαρακτηρίζεται παραπάνω από εύλογη. Αντικείμενο αυτής της διπλωματικής είναι η ενσωμάτωση των σκληρών δίσκων νέας γενιάς της Seagate, οι οποίοι καθιστούν εφικτή τη διαχείριση των δεδομένων χρησιμοποιώντας αντικείμενα key-value, στην πλατφόρμα διαμοιρασμού δεδομένων που έχει αναπτύξει η ερευνητική ομάδα Συστημάτων Αποθήκευσης (Storage Systems research group) στο Κέντρο Υπερυπολογιστικής της Βαρκελώνης (Barcelona Supercomputing Center).

Λέξεις κλειδιά

Big Data, Υπολογιστική Νέφους, Διαμοιρασμός δεδομένων, Διαμοιρασμός μοντέλων δεδομένων, Αντικειμενοστραφή συστήματα αποθήκευσης, Key-value storage

Abstract

Currently, Big Data actors tackle many difficulties when cooperation among them is needed. New ways of cooperation, which make it in a seamless way, are needed. As the object-oriented programming paradigm is the most dominant, the new approaches adopt it in Big Data field, too. Furthermore, hardware follows this object-based fashion, and new storage technologies enable data management using the key-value object abstraction. So, the attempt to pair object-based software and hardware is more than plausible. Objective of this master thesis is the integration of Seagate's new generation hard disk drives, which enable manipulation of data in key-value fashion, into the novel data-sharing platform that has been developed by the Storage Systems research group at Barcelona Supercomputing Center.

Keywords

Big Data, Cloud computing, Data sharing, Data model sharing, Object-based storage systems, Key-value storage

Ευχαριστίες

Όταν πλησιάζεις στην πραγματοποίηση των στόχων σου, οφείλεις να εκφράσεις τις ευχαριστίες σου σε αυτούς συνέδραμαν στην εκπλήρωσή τους.

Θα ήθελα να ευχαριστήσω όλα τα μέλη του Storage Systems Research Group στο Barcelona Supercomputing Center, οι οποίοι με βοήθησαν κατά την εκπόνηση αυτής της διπλωματικής. Συγκεκριμένα, θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου, Toni Cortes, που με εμπιστεύτηκε και με δέχτηκε στην ομάδα του ως επισκέπτη φοιτητή, παρόλο που, στην ουσία, δεν γνώριζε σχεδόν τίποτα για μένα.

Επίσης, θα ήθελα να ευχαριστήσω τα μέλη του Εργαστηρίου Υπολογιστικών Συστημάτων στο Εθνικό Μετσόβιο Πολυτεχνείο, οι οποίοι με υποστήριξαν και μου επίτρεψαν να εκπονήσω τη διπλωματική μου στο εξωτερικό ως φοιτητής ανταλλαγής.

Τέλος, οφείλω, ασφαλώς, πολλά ευχαριστώ στην οικογένειά μου και στους φίλους μου. Στην οικογένειά μου, γιατί δεν θα είχα την ευκαιρία για Πανεπιστημιακές σπουδές χωρίς την υποστήριξή τους. Και στους φίλους μου, για τις κοινές μνήμες κατά τη διάρκεια του περιπετειώδους ταξιδιού της φοιτητικής ζωής.

Contents

List of Figures

List of Tables

Κεφάλαιο 1

Συνοπτική Περιγραφή

Σκοπός του παρόντος κεφαλαίου είναι να παρουσιάσει περιληπτικά το περιεχόμενο αυτής της διπλωματικής εργασίας στην ελληνική γλώσσα. Όπως αναφέρεται και στον τίτλο της εργασίας, το αντικείμενο αυτής της διπλωματικής εργασίας είναι η "Ενσωμάτωση αντικειμενοστραφών μέσων αποθήκευσης σε πλατφόρμα διαμοιρασμού δεδομένων". Πιο συγκεκριμένα, αυτό που επιχειρείται είναι να ενσωματωθούν οι σκληροί δίσκοι νέας γενιάς της Seagate, οι Kinetic drives, στην πλατφόρμα διαμοιρασμού δεδομένων που έχει αναπτύξει το Storage Systems Research Group του Barcelona Supercomputing Center (BSC), το dataClay. Ιδιάζον χαρακτηριστικό των Kinetic drives είναι ότι καθιστούν εφικτή τη διαχείριση δεδομένων χρησιμοποιώντας αντικείμενα key-value.

Στο πρώτο στάδιο αυτού του κεφαλαίου, παρουσιάζεται το κίνητρο το οποίο οδήγησε στην ανάπτυξη του dataClay καθώς και τα βασικά χαρακτηριστικά του. Στη συνέχεια, παρουσιάζεται το κίνητρο για την ενσωμάτωση της τεχνολογίας Kinetic στο dataClay. Έπειτα, γίνεται παρουσίαση των πιο σημαντικών τεχνικών χαρακτηριστικών του dataClay και των Kinetic drives τα οποία χρειάστηκε να κατανοηθούν πριν γίνει η σύζευξη των δύο τεχνολογιών. Στη συνέχεια, παρουσιάζεται το συνθετικό μέρος αυτής της διπλωματικής εργασίας που δεν είναι άλλο από την ενσωμάτωση των Kinetic drives στο dataClay. Τέλος, παρουσιάζεται η αξιολόγηση της ανωτέρω εργασίας καθώς και τα συμπεράσματα τα οποία προέκυψαν.

1.1 Περιγραφή του dataClay

Αν επιχειρούσαμε να διατυπώσουμε σε μία πρόταση το κίνητρο για την ανάπτυξη του dataClay, θα ήταν ότι στην εποχή των Big Data οι ισχύουσες λύσεις για συνεργασία μεταξύ διαφόρων φορέων είναι ανεπαρκείς. Με τον όρο συνεργασία εννοείται είτε ο διαμοιρασμός δεδομένων είτε ο διαμοιρασμός μοντέλων δεδομένων (data models). Για να γίνει περισσότερο κατανοητή η ανεπάρκεια των σημερινών λύσεων, ας δούμε τα ελαττώματα τα οποία εμπεριέχουν.

Η πρώτη επιλογή για συνεργασία, η οποία είναι και η πιο ευέλικτη, είναι όλοι οι εμπλεκόμενοι (δηλαδή οι ιδιοκτήτες δεδομένων και οι συνεργάτες τους) να έχουν πλήρη πρόσβαση στην υποδομή των δεδομένων (π.χ. μία Βάση Δεδομένων). Ωστόσο, αυτή η λύση είναι εφικτή μόνο όταν υπάρχει ισχυρή σχέση εμπιστοσύνης μεταξύ των διαφόρων εμπλεκόμενων ή τα δεδομένα είναι

ανοιχτά/δημόσια. Σε κάθε περίπτωση, τα δεδομένα συνήθως είναι μόνο για ανάγνωση (read-only) και όλοι οι ενδιαφερόμενοι αναγκάζονται να δημιουργήσουν ένα δικό τους αντίγραφο των δεδομένων εάν απαιτούνται τροποποιήσεις, το οποίο συνεπάγεται την περαιτέρω χρήση πόρων χρόνου και χώρου.

Η δεύτερη επιλογή για συνεργασία είναι οι κάτοχοι των δεδομένων να παρέχουν πρόσβαση σε συγκεκριμένα σύνολα δεδομένων (datasets) στους συνεργάτες τους (τους οποίους θα τους καλούμε και ως καταναλωτές). Ναι μεν οι καταναλωτές έχουν πλέον την ευελιξία να διαχειριστούν τα δεδομένα όπως αυτοί επιθυμούν, από τη στιγμή που έχει δημιουργεί το αντίγραφο, αλλά αυτή η διαδικασία συνεπάγεται υπερβολική μεταφορά δεδομένων, καθώς επίσης και το γεγονός ότι οι κάτοχοι των δεδομένων χάνουν πλέον τον έλεγχό τους αφού δημιουργούνται πολλαπλά αντίγραφα.

Η τρίτη και τελευταία επιλογή για συνεργασία είναι να χρησιμοποιηθεί κάποιο data service (όπως το RESTful web service). Σε αυτήν την περίπτωση, ο ιδιοκτήτης (ή αλλιώς και κάτοχος) των δεδομένων μοιράζεται με τους ενδιαφερόμενους συνεργάτες την υπολογιστική υποδομή που και ο ίδιος χρησιμοποιεί, επιλέγοντας όχι μόνο τι θα μοιραστεί αλλά και πώς. Αυτή η επιλογή είναι πολύ περιοριστική στην περίπτωση που κάποιος καταναλωτής χρειαστεί μία επιπλέον λειτουργικότητα από αυτές που παρέχονται από τον κάτοχο. Σε αυτήν την περίπτωση, απαιτείται η συνεργασία μεταξύ κατόχου και καταναλωτή ούτως ώστε ο κάτοχος να ενσωματώσει την επιπλέον λειτουργικότητα που επιθυμεί ο καταναλωτής. Ως εκ τούτου, ένας εμπλεκόμενος δε μπορεί να εμπλουτίσει μία τέτοια υπηρεσία με τη δική του πνευματική ιδιοκτησία και απαιτείται να κρατήσει τις εφαρμογές του συμβατές με το API του παρόχου.

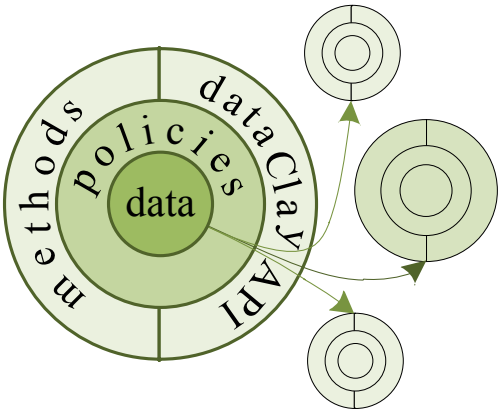
Επιπλέον, το dataClay αντιμετωπίζει μία ακόμη πρόκληση που συναντούν διαρκώς οι προγραμματιστές. Πιο συγκεκριμένα, επιτρέπει στους χρήστες του να μην απασχολούνται με το περιβάλλον στο οποίο βρίσκονται τα δεδομένα τους και να εστιάζουν στη λογική των εφαρμογών τους. Σήμερα, τα μοντέλα δεδομένων έχουν σχεδιαστεί με διαφορετικό τρόπο ανάλογα με το αν αυτά τα μοντέλα προσπελαύνονται σε ένα πτητικό περιβάλλον ή μη. Στα μη-πτητικά περιβάλλοντα, οι εφαρμογές έρχονται αντιμέτωπες είτε με συστήματα αρχείων είτε με βάσεις δεδομένων. Στον αντίποδα, στα πτητικά περιβάλλοντα, οι εφαρμογές αναγκάζονται να δεσμεύσουν ελεύθερη μνήμη προκειμένου να φορτώσουν τα δεδομένα και να τα διαχειριστούν. Δοθέντος αυτών των διαφορών, οι προγραμματιστές είναι αναγκασμένοι να αφιερώνουν χρόνο, μεν, για το σχεδιασμό μοντέλων δεδομένων ανάλογα με το περιβάλλον (πτητικό ή μη), δε, για τη μετάβαση από το ένα περιβάλλον στο άλλο.

Ενδεχομένως, ο αναγνώστης να βρίσκεται σε σύγχυση σχετικά με την έννοια του διαμοιρασμού μοντέλων δεδομένων. Ένα παράδειγμα μάλλον θα αποσαφηνίσει αυτήν την έννοια. Αν μοιραστούμε το schema μίας Βάσης Δεδομένων, δεν είναι τα δεδομένα που μοιράζονται αλλά η δομή (schema) του μοντέλου το οποίο σε συνδυασμό με τα δεδομένα αναπαριστά το προς μελέτη πρόβλημα. Ναι μεν στο συγκεκριμένο παράδειγμα επιτυγχάνεται η αποσύζευξη των δεδομένων από το μοντέλο τους κι έτσι το καθένα μπορεί να διαμοιραστεί ανεξάρτητα, ωστόσο παρατηρείται ότι αν επιθυμούμε επεξεργασία των δεδομένων, απαιτείται επιπλέον φόρτος εργασίας προκειμένου να ανακτήσουμε τα δεδομένα και να τα επεξεργαστούμε. Το dataClay πετυχαίνει και την αποσύζευξη των δεδομένων από τα μοντέλα δεδομένων, αλλά και την επεξεργασία των δεδομένων με έναν άμεσο τρόπο.

Με βάση τα ελαττώματα που παρουσιάζουν οι υπάρχουσες λύσεις για το διαμοιρασμό δεδομένων αλλά και τη διαχερίση τους με βάση το περιβάλλον που βρίσκονται, το dataClay αναλαμβάνει την

πρόκληση να αντιμετωπίσει όλα αυτά τα εμπόδια. Αρχικά, επιτυγχάνει τη συνεργασία μεταξύ των ενδιαφερόμενων φορέων καθιστώντας εφικτό το διαμοιρασμό των δεδομένων αλλά και των μοντέλων δεδομένων. Αυτό επιτυγχάνεται χωρίς να χάνει ούτε μία στιγμή ο κάτοχος των δεδομένων τον πλήρη έλεγχό τους. Επιπλέον, αποφεύγονται όλες οι άσκοπες μεταφορές δεδομένων αλλά δίνεται επιπλέον η δυνατότητα σε όλους τους ενδιαφερόμενους να εμπλουτίσουν τα μοντέλα δεδομένων με τη λειτουργικότητα που επιθυμούν χωρίς να απαιτείται η εμπλοκή των κατόχων. Τέλος, το dataClay αντιμετωπίζει τα δεδομένα με ένα διαφανή τρόπο, ανεξαρτήτως του περιβάλλοντος που βρίσκονται. Πιο συγκεκριμένα, έχοντας ως στόχο τη διευκόλυνση των προγραμματιστών, το dataClay παρέχει όλους τους μηχανισμούς που είναι απαραίτητοι ούτως ώστε τα δεδομένα που βρίσκονται σε ένα μη-πτητικό περιβάλλον να προσπεύλανονται με τον ίδιο τρόπο στην περίπτωση που βρίσκοταν στη μνήμη.

Προκειμένου να γίνει κατανοητή η λειτουργικότητα που παρέχει το dataClay στους χρήστες και η οποία έχει άμεση σχέση με το αντικείμενο αυτής της διπλωματικής, παρουσιάζεται ένα υποσύνολο των χαρακτηριστικών του. Σήμερα, το πιο δημοφιλές προγραμματιστικό μοντέλο είναι το αντικειμενοστραφές (OOP). Το dataClay χρησιμοποιεί τα αντικείμενα από το αντικειμενοστραφές προγραμματιστικό μοντέλο με την καθιερωμένη έννοια και προσθέτει δύο χαρακτηριστικά: 1) Τις πολιτικές (policies), χάρις στις οποίες είναι δυνατή η συνεργασία μεταξύ των ενδιαφερόμενων μερών, και 2) τη διεπαφή (API) που διευκολύνει την αποθήκευση και την ανάκτηση των αντικειμένων. Το αποτέλεσμα αυτής της σύζευξης ονομάζεται "Αυτάρκη αντικείμενα" (Self-contained objects - SCOs - Σχήμα 1.1). Αξίζει να σημειώσουμε ότι χάρις στο αντικειμενοστραφές μοντέλο επιτυγχάνεται το πολυπόθητο χαρακτηριστικό "Υπολογισμός κοντά στα δεδομένα" (Computation close to the data).



Σχημα 1.1: Ένας αυτάρκες αντικείμενο και η σχέση του με άλλα αυτάρκη αντικείμενα

Λογικά γίνεται πλέον αντιληπτό ότι ο διαμοιρασμός μοντέλων δεδομένων επιτυγχάνεται επιτρέποντας το διαμοιρασμό των OOP κλάσεων. Επομένως, και ο εμπλουτισμός των μοντέλων από τρίτα μέρη (3rd party enrichments) επιτυγχάνεται επιδρώντας πάνω στις κλάσεις που μοιράζονται. Πιο συγκεκριμένα, δίνεται η δυνατότητα στους χρήστες να προσθέτουν στις κλάσεις είτε νέα πεδία είτε νέες μεθόδους.

1.2 Αντικείμενο της διπλωματικής

Όπως έχει ήδη αναφερθεί, τα αντικείμενα προσπελαύνονται με τον ίδιο τρόπο είτε βρίσκονται σε πτητικό περιβάλλον είτε όχι. Με άλλα λόγια, είναι ασφαλές για τον προγραμματιστή να υλοποιεί την εφαρμογή του θεωρώντας ότι τα δεδομένα είναι πάντα διαθέσιμα στη μνήμη. Ωστόσο, όπως είναι προφανές δεν είναι δυνατόν τα δεδομένα να βρίσκονται στη μνήμη εσαεί. Απεναντίας, το dataClay φέρει τους μηχανισμούς οι οποίοι επιτρέπουν τη μετάβαση από το μη-πτητικό περιβάλλον στη μνήμη και το αντίστροφο.

Στην παρούσα φάση, έχουν αναπτυχθεί μηχανισμοί οι οποίοι αναλαμβάνουν την αντιστοίχιση των SCOs σε διάφορες βάσεις δεδομένων όπως σχεσιακές ή γραφικές κ.α. Ωστόσο, όπως είναι κατανοητό, η μετάβαση από το ένα περιβάλλον στο άλλο απαιτεί αρκετή επεξεργασία των δεδομένων καθώς τα δεδομένα μοντελοποιούνται με διαφορετικό τρόπο ανάλογα με το περιβάλλον. Παραδείγματος χάριν, όταν χρησιμοποιούνται σχεσιακές βάσεις δεδομένων για την αποθήκευση των SCOs, απαιτείται η μετάβαση από το αντικειμενοστραφές μοντέλο των SCOs στη μορφή πινάκων που χρησιμοποιούν οι σχεσιακές βάσεις. Θα ήταν πολύ προτιμότερο να υπήρχε η κατάλληλη υποδομή η οποία θα επιτρέπει την αποθήκευση των δεδομένων χρησιμοποιώντας το ίδιο μοντέλο με αυτό που χρησιμοποιείται το dataClay, δηλαδή αντικείμενα. Αποδεικνύεται ότι υπάρχει τέτοια τεχνολογία, τουλάχιστον σε φάση ανάπτυξης. Πρόκειται για την τεχνολογία Kinetic που έχει αναπτύξει η Seagate και έχει υλοποιηθεί σε σκληρούς δίσκους. Στην τεχνολογία Kinetic χρησιμοποιείται η ιδιαίτερα απλή αφαιρετικότητα των αντικειμένων key-value. Συνεπώς, αντικείμενο αυτής της διπλωματικής εργασίας είναι η σύζευξη των δύο τεχνολογιών, του dataClay και της Kinetic.

1.3 Περαιτέρω κατανόηση του dataClay

Πριν προχωρήσουμε στο αμιγώς συνθετικό μέρος αυτής της διπλωματικής εργασίας, είναι απαραίτητο, πρώτον, να κατανοήσουμε ουσιωδώς κάποια βασικά μέρη του dataClay, δεύτερον, να κατανοήσουμε σε κάθε λεπτομέρεια το custom μηχανισμό σειριοποίησης του dataClay, αλλά, και τρίτον, να αποκτήσουμε οικειότητα με το API των Kinetic drives.

1.3.1 Stub classes

Όταν γίνεται διαμοιρασμός μοντέλων δεδομένων, οι δικαιούχοι τους μπορούν να ανακτήσουν τις προς διαμοιρασμό κλάσεις είτε για να δημιουργήσουν νέες δικές τους εφαρμογές είτε για να εμπλουτίσουν τις υπάρχοντες κλάσεις με νέα πεδία ή μεθόδους. Συγκεκριμένα, αυτό που κάνουν οι καταναλωτές είναι να «κατεβάζουν» (download) ένα stub ανά κλάση. Αυτά τα stubs παράγονται από το dataClay φιλτράροντας και επιτρέποντας μόνο τα ορατά πεδία και μεθόδους σύμφωνα με τα συμβόλαια που έχουν συναφθεί. Επιπλέον, κάθε stub περιέχει ένα σύνολο μεθόδων (το dataClay API που περιγράφηκε προηγουμένως) το οποίο έχει κληρονομηθεί από μία κοινή κλάση (DataClayObject) την οποία όλα τα stubs επεκτείνουν (κάτι ανάλογο με την κλάση Object της Java).

1.3.2 Custom serialization

Όπως έχει αναφερθεί ήδη, το dataClay ενσωματώνει ένα custom μηχανισμό σειριοποίησης. Υπάρχει πληθώρα λόγων που οδήγησε σε αυτήν την απόφαση:

- Ο εγγενής μηχανισμός σειριοποίησης της Java απαιτεί να έχουμε την ίδια κλάση και στην πλευρά του server και του client. Το οποίο δεν ισχύει στην περίπτωσή μας, εξαιτίας του διαμοιρασμού μοντέλων δεδομένων που εφαρμόζουμε (βλέπε stubs κλάσεις).
- Η σειριοποίηση της Java δεν είναι αποδοτική λόγω του reflection. Ακόμη και αν υλοποιήσουμε το Externalizable interface, το αποφεύγει το reflection, η απόδοση εξακολουθεί να είναι ισχνή.
- Υπάρχουν κάποιες αναπαραστάσεις, όπως οι κυκλικές αναφορές ή αναφορές σε άλλα ήδη αποθηκευμένα αντικείμενα (που, συνεπώς, δεν χρειάζονται σειριοποίηση ξανά), οι οποίες είναι δύσκολο να αναπαρασταθούν στο Java RMI.
- Το να έχουμε το δικό μας μηχανισμό σειριοποίησης μας επιτρέπει να αποφύγουμε τη σειριοποίηση και την αποσειριοποίηση σε ενδιάμεσα στάδια.
- Τέλος, ο μηχανισμός σειριοποίησης της Java περιέχει δεδομένα και για τις κλάσεις μαζί με τα στιγμιότυπα (όπως οι τύποι, κ.ά.), το οποίο προκαλεί πλεονασμό πληροφορίας.

Όπως γίνεται κατανοητό, είναι φυσικό να επιχειρηθεί η ανάπτυξη ενός custom μηχανισμού σειριοποίησης. Σε αυτήν την ενότητα, παρουσιάζεται ο μηχανισμός σειριοποίησης του dataClay. Αν αναλύσουμε τον αλγόριθμο του custom μηχανισμού σειριοποίησης σε υψηλό επίπεδο, είναι τέσσερα τα βασικά χαρακτηριστικά του:

- Πρόκειται για μία αναδρομική διαδικασία: Κατά τη διάρκεια της σειριοποίησης ενός αντικειμένου, εάν ένα από τα πεδία του προς σειριοποίηση αντικειμένου περιέχει αναφορά προς άλλο μην αποθηκευμένο αντικείμενο, τότε σειριοποιείται και το προς αναφορά αντικειμένο, και αφού τελειώσει η σειριοποίηση του αντικειμένου-πεδίου συνεχίζεται η σειριοποίηση του αρχικού αντικειμένου.
- Κάθε σειριοποιημένο αντικείμενο περιέχει σειριοποιημένα, εκτός άλλων, και όλα τα πεδία όλων των κλάσεων τις οποίες κληρονομεί: Πιο συγκεκριμένα, μόλις ολοκληρωθεί η σειριοποίηση ενός αντικειμένου, τότε ξεκινάει η σειριοποίηση των υπερκλάσεών του. Αφού κάθε κλάση επεκτείνει το DataClayObject, κάθε αντικείμενο περιέχει πεδία τουλάχιστον μίας υπερκλάσης. Αυτό το γεγονός, δηλαδή ότι όλα οι κλάσεις επεκτείνουν το DataClayObject, διαμορφώνει και τη συνθήκη τερματισμού της σειριοποίησης. Μόλις σειριοποιηθούν τα πεδία του DataClayObject, η σειριοποίηση τερματίζεται.
- Κατά τη διάρκεια σειριοποίησης ενός αντικειμένου, εάν το dataClay βρει μία αναφορά προς ένα αντικείμενο που έχει ήδη σειριοποιηθεί, αυτό το αντικείμενο δεν θα σειριοποιηθεί ξανά. Υπάρχουν δύο λόγοι που οδηγούν σε αυτή την επιλογή: 1) Πλεονασμός επεξεργασίας και πληροφορίας αποφεύγεται, και 2) Οποιαδήποτε κυκλική αναφορά αποφεύγεται: Φανταστείτε εάν το dataClay κατέφευγε στην επεξεργασία μίας κυκλικής αναφοράς: Θα κατέληγε σε

μία ατέρμονη σειριοποίηση. Προκειμένου να αποφύγει τέτοιες δυσάρεστες καταστάσεις, το dataClay επισημειώνει (tags) κάθε αντικείμενο με έναν ακέραιο. Εάν ένα αντικείμενο ένα έχει σειριοποιηθεί προηγουμένως, το dataClay απλά προσθέτει το tag που αντιστοιχεί στο σειριοποιημένο αντικείμενο. Έπειτα συνεχίζει με τη σειριοποίηση του επόμενου πεδίου. Πώς, όμως, το dataClay αναγνωρίζει τα ήδη σειριοποιημένα αντικείμενα· Για αυτό το σκοπό, χρησιμοποιεί ένα map, του οποίου τα κλειδιά (keys) είναι τα hashcodes των αντικειμένων και οι τιμές είναι τα tags που έχουν χρησιμοποιηθεί για τα ήδη σειριοποιημένα αντικείμενα. Εάν βρεθεί ένα ζεύγος key-value στο map για ένα αντικείμενο, αυτό σημαίνει ότι το αντικείμενο έχει σειριοποιηθεί προηγουμένως. Σε αυτήν την περίπτωση, απλά προστίθεται το tag. Στον αντίποδα, όταν ένα αντικείμενο αντιμετωπίζεται για πρώτη φορά, αυτό επισημειώνεται με τον αμέσως επόμενο διαθέσιμο tag και αντίστοιχο key-value ζευγάρι προστίθεται στο map.

- Όταν ένα αντικείμενο σειριοποιείται, αυτό το αντικείμενο πιθανώς να περιέχει αναφορές προς άλλα αντικείμενα που έχουν ήδη αποθηκευτεί (με άλλα λόγια, να είναι persistent). Σε αυτήν την περίπτωση, δεν έχει νόημα ούτε να αποθηκεύσουμε ξανά (δηλαδή, να επανεγγράψουμε - overwrite) το ήδη αποθηκευμένο αντικείμενο, ούτε να το σειριοποιήσουμε. Αντί να επανασειριοποιήσουμε το persistent αντικείμενο, το dataClay γράφει το αναγνωριστικό αντικειμένου (object ID) στο μήνυμα σειριοποίησης του προς αποθήκευση αντικειμένου. Έπειτα, η σειριοποίηση των εναπομείνοντων πεδίων συνεχίζεται.

Ωστόσο, ποιο είναι το κριτήριο που κάνει ένα αντικείμενο persistent; Με άλλα λόγια, πώς αναγνωρίζει το dataClay τα αποθηκευμένα αντικείμενα; Ο κανόνας είναι: Εάν ένα αντικείμενο έχει ένα dataClay object ID, αυτό το αντικείμενο είναι persistent. Αλλιώς, δεν είναι. Ένα αντικείμενο συσχετίζεται με το αναγνωριστικό του (object ID) για ολόκληρη τη ζωή του (life cycle), μόλις αποθηκεύεται. Επομένως, ο έλεγχος του εάν ένα αντικείμενο είναι αποθηκευμένο ή όχι ισοδυναμεί με το εάν έχει αναγνωριστικό (object ID).

Είναι πολύ σημαντικό να ξεκαθαρίσουμε ότι ένα object ID δεν είναι ένα αναγνωριστικό της γλώσσας Java. Πρόκειται για ένα εσωτερικό αναγνωριστικό που χρησιμοποεί το dataClay. Ένας λόγος που τα αναγνωριστικά της Java (όπως το hashcode ενός αντικειμένου) δεν επαρκούν είναι ότι καθορίζουν τα αντικείμενα μόνο όσο αυτά βρίσκονται εντός του σωρού της Java. Αντιθέτως, το dataClay χρειάζεται να αναγνωρίζει τα αντικείμενά του ανεξαρτήτως του εάν αυτά βρίσκονται αποθηκευμένα ή βρίσκονται στη μνήμη. Επιπλέον, τα αντικείμενα του dataClay πιθανώς να μοιραστούν ανάμεσα σε διαφορετικούς clients και servers. Επομένως, χρειαζόμαστε ένα μοναδικό αναγνωριστικό για ολόκληρο το σύστημα. Για αυτό το λόγο το dataClay χρησιμοποιεί τα δικά του αναγνωριστικά (IDs), τα οποία είναι στην ουσία ένα πεδίο σε κάθε αντικείμενο. Πιο συγκεκριμένα, είναι ένα πεδίο στην κλάση DataClayObject. Αφού κάθε stub κλάση επεκτείνει την κλάση DataClayObject, τότε κάθε αντικείμενο έχει αυτό το πεδίο για το object ID.

Τέλος, το μόνο που απομένει να απαντηθεί είναι πώς και πότε επιτυγχάνεται αυτό το δέσιμο μεταξύ ενός object ID και του μόλις αποθηκευμένου αντικειμένου. Η απάντηση είναι ότι όταν ολοκληρώνεται η αποθήκευση ενός αντικειμένου στην υποδομή δεδομένων (π.χ. μία σχεσιακή βάση δεδομένων), το αναγνωριστικό (dataClay object ID) που έχει χρησιμοποιηθεί για αυτό το αντικείμενο επιστρέφεται ως αποτέλεσμα της καλούσας μεθόδου που έχει αναλάβει την αποθήκευση. Το δαταˆλαψ λαμβάνει αυτό το αναγνωριστικό και το θέτει στο αντίστοιχο αντικείμενο,

το οποίο εξακολουθεί να βρίσκεται και στη μνήμη. Μετέπειτα, οποιαδήποτε προσπάθεια για επανασειριοποίηση του αντικειμένου θα αποφευχθεί, αφού αυτό το αντικείμενο έχει ήδη ένα object ID.

1.4 Συνθετικό μέρος: Kinetic handler

Σε αυτήν την ενότητα περιγράφεται το συνθετικό μέρος αυτής της διπλωματικής εργασίας.

1.4.1 Kinetic key-value objects

Αυτή η ενότητα περιγράφει εν συντομία τα τεχνικά χαρακτηριστικά των αντικειμένων key-value τα οποία αποθηκεύονται σε ένα δίσκο Kinetic. Σύμφωνα με το API της τεχνολογίας Kinetic, ένα τέτοιο αντικείμενο καλείται Entry. Με άλλα λόγια, υπάρχει μία Java κλάση, που ονομάζεται Entry, η οποία αναπαριστά τα αντικείμενα key-value.

Κάθε στιγμιότυπο της κλάσης Entry αναγνωρίζεται από το μοναδικό του κλειδί (key), το οποίο σε όρους Java είναι ένα πεδίο byte array στην κλάση Entry, που ονομάζεται key. Το μέγιστο μέγεθος αυτού είναι 4 KB. Παρομοίως, κάθε αντικείμενο της κλάσης Entry έχει ένα άλλο πεδίο byte array για την αποθήκευση της τιμής (value) του αντικειμένου key-value, το οποίο φυσικώς ονομάζεται value. Το μέγιστο μέγεθος αυτού είναι 1 MB.

Επομένως, ένα αντικείμενο key-value σε ένα δίσκο Kinetic είναι ένα ζεύγος δύο πινάκων byte. Είναι αποκλειστική ευθύνη του προγραμματιστή να αποφασίσει τι θα αποθηκευτεί στα δύο πεδία, key και value.

Τέλος, είναι άξιο αναφοράς το γεγονός ότι όλα τα αντικείμενα key-value εντός ενός δίσκου Kinetic ταξινομούνται με βάση το κλειδί τους. Σύμφωνα με τις προδιαγραφές της τεχνολογίας Kinetic, ένα σχήμα/δομή (schema) για τα κλειδιά των αντικειμένων (object keys) το οποίο τοποθετεί τα αντικείμενα ακολουθιακά, μπορεί να βελτιστοποιήσει την απόδοση. Με άλλα λόγια, όταν προσπελαύνονται αντικείμενα key-value τα οποία βρίσκονται το ένα δίπλα στο άλλο, η απόδοση του δίσκου Kinetic βελτιστοποιείται.

1.4.2 Αναπαράσταση των κλάσεων στους δίσκους Kinetic

Όταν κάποιος χρησιμοποιεί μία αντικειμενοστραφή γλώσσα προγραμματισμού και σχεσιασές βάσεις δεδομένων, είναι ιδιαίτερα συχνό οι κλάσεις να αντιστοιχίζονται σε πίνακες της βάσης δεδομένων και κάθε αντικείμενο να αντιστοιχίζεται σε μία εγγραφή στον κατάλληλο πίνακα. Το dataClay ακολουθεί και αυτό την προαναφερθείσα πρακτική για το Postgres handler του, που αποθηκεύει SCOs σε μία σχεσιακή βάση Postgres. Προκειμένου να το πετύχει αυτό, όταν ένας χρήστης καταχωρεί στο dataClay μία νέα κλάση που έχει ορίσει ο ίδιος, η υποδομή δεδομένων (data infrastructure), η οποία, στην περίπτωσή μας, είναι μία σχεσιακή βάση δεδομένων πρέπει να προετοιμαστεί για μελλοντική αποθήκευση αντικειμένων. Για αυτό το σκοπό, δημιουργείται ένας πίνακας που αντιστοιχεί στην κλάση που μόλις έχει καταχωρηθεί στο dataClay και τα μελλοντικά

στιγμιότυπα αυτής της κλάσης θα αποθηκευτούν στη βάση δεδομένων ως εγγραφές αυτού του πίνακα.

Στον αντίποδα, η αρχιτεκτονική της τεχνολογίας Kinetic έχει υιοθετήσει την πολύ απλούστερη αφαιρετικότητα των αντικειμένων key-value. Μπορούμε να αποθηκεύσουμε μόνο δύο πίνακες από bytes (byte arrays), έναν για το κλειδί (key) και έναν για την τιμή (value). Τίποτα περισσότερο. Επομένως, το να αναζητήσουμε για μία δομημένη αναπαράσταση των δεδομένων όπως κάνουν οι πίνακες των σχεσιακών βάσεων δεδομένων ή τα συστήματα αρχείων δεν έχει πολύ νόημα στην περίπτωση της τεχνολογίας Kinetic. Αντ' αυτού θα δυσχέραινε μονάχα την απόδοση. Η πιο άμεση και, μάλλον, πιο αποτελεσματική λύση είναι να αποθηκεύουμε κάθε σειριοποιημένο αντικείμενο του dataClay (δηλαδή ένα SCO) ως ένα αντικείμενο key-value (δηλαδή αντικείμενο Entry), του οποίου η τιμή (value) θα είναι τα bytes του σειριοποιημένου αντικειμένου. Ως εκ τούτου, οι δίσκοι Kinetic δεν χρειάζεται να κάνουν κάτι για την προετοιμασία της υποδομής, όπως κάνουν οι σχεσιακές βάσεις δεδομένων δημιουργώντας πίνακες. Στην πραγματικότητα, στους δίσκους Kinetic κάθε αντικείμενο δεν σχετίζεται με τα υπόλοιπα, σε αντίθεση με τους πίνακες των σχεσιακών βάσεων δεδομένων που εμφωλεύουν όλα τα αντικείμενα της ίδιας κλάσης. Στους δίσκους Kinetic όλα τα αντικείμενα είναι ανεξάρτητα.

Από την άλλη πλευρά, οι πίνακες στις σχεσιακές βάσεις δεδομένων επιτυγχάνουν πολύ αποτελεσματικά την έννοια της ομαδοποίησης παρόμοιων ειδών: Κάθε εγγραφή σε έναν πίνακα αναπαριστά ένα αντικείμενο της ίδιας κλάσης. Για παράδειγμα, αναζητώντας όλα τα αντικείμενα μία κλάσης ισοδυναμεί μόνο με μία εντολή "SELECT * FROM". Ο handler του dataClay για τους σκληρούς δίσκους Kinetic πετυχαίνει και αυτός την έννοια της ομαδοποίησης. Αφού κάθε αντικείμενο Entry ταξινομείται με βάση το κλειδί του, η ομαδοποίηση αντικειμένων της ίδιας κλάσης επιτυγχάνεται εύκολα και στους δίσκους Kinetic. Εάν χρησιμοποιήσουμε το ίδιο πρόθεμα για το κλειδί των αντικειμένων που ανήκουν στην ίδια κλάση, τότε κάθε αντικείμενο Entry θα βρίσκεται το ένα δίπλα στο άλλο, εξαιτίας της ταξινόμησης των αντικειμένων Entry. Στην περίπτωσή μας, το φυσικότερο είναι να χρησιμοποιήσουμε ως πρόθεμα το αναγνωριστικό των κλάσεων (class ID) ως πρόθεμα για το key schema.

1.4.3 Single-object rule

Όπως έχει ήδη δηλωθεί προηγουμένως, ο handler για τους δίσκους Kinetic αποθηκεύει κάθε αντικείμενο ενός χρήστη του dataClay ως ένα αντικείμενο key-value Entry στο σκληρό δίσκο Kinetic. Αυτός είναι ένας κανόνας που ποτέ δεν πρέπει να παραβιάζεται ανεξαιρέτως κατεύθυνσης: Κάθε αντικείμενο πρέπει να αποθηκεύεται ως μία ολόκληρη οντότητα και ποτέ δεν κατατέμνεται. Από την άλλη πλευρά, ένα αντικείμενο key-value Entry στο δίσκο Kinetic δε μπορεί να περιέχει πληροφορίες (δηλαδή, bytes) για παραπάνω από ένα αντικείμενο. Επιπλέον, επιθυμούμε οι σχέσεις εξάρτησης ανάμεσα στα αντικείμενα που υπάρχουν λόγω των αναφορών μεταξύ τους να ανταναακλάται και στα αντικείμενα που αποθηκεύουμε στο δίσκο Kinetic. Πιθανότατα, αυτός είναι ο πιο σημαντικός κανόνας στο Kinetic handler. Από εδώ και στο εξής, θα αποκαλούμε αυτόν τον κανόνα "Single-object rule".

1.4.4 Παγίδα για παραβίαση του Single-object rule

Ας δούμε μέσω ενός παραδείγματος ότι αν δεν προσέξουμε, ο Single-object rule μπορεί να παραβιαστεί. Ας υποθέσουμε ότι επιθυμούμε την αποθήκευση ενός αντικειμένου, το οποίο θα το ονομάσουμε objectA. Επιπλέον, ας υποθέσουμε ότι το objectA περιέχει μία αναφορά προς ένα άλλο μη αποθηκευμένο αντικείμενο, το objectB. Αυτό υπονοεί ότι και τα αντικείμενα πρέπει να αποθηκευτούν στην υποδομή δεδομένων (στην περίπτωσή μας, το δίσκο Kinetic). Σύμφωνα με το μηχανισμό σειριοποίσης που περιγράφηκε προηγουμένως, το αντικείμενο objectB θα σειριοποιηθεί "μέσα" στο αντικείμενο objectA, αφού το αντικείμενο objectB δεν είναι ούτε αυτό αποθηκευμένο.

Εάν αποθηκεύσουμε το αντικείμενο objectA ακαριαία, χωρίς να απομονώσουμε το objectB, αυτό θα προκαλέσει παραβίαση του Single-object rule. Συγκεκριμένα, το αντικείμενο key-value Entry στο δίσκο Kinetic θα περιέχει πληροφορία (bytes) για παραπάνω από ένα αντικείμενο. Επομένως, το να αποθηκεύουμε αντικείμενα του dataClay στο δίσκο Kinetic δεν μπορεί να επιτευχθεί πραγματοποιώντας απλά την μεταφορά των σειριοποιημένων αντικειμένων (λειτουργία put). Πάντα, απαιτείται η επεξεργασία τους πρώτα.

1.4.5 Κίνητρο για το Single-object rule

Πριν προχωρήσουμε στην περιγραφή του τρόπου με τον οποίο εκπληρώνεται ο Single-object rule, είναι κρίσιμο να παρουσιάσουμε το κίνητρο για να έχουμε ένα τέτοιο κανόνα. Στην περίπτωσή μας, θα δούμε ότι ο κανόνας του «οικονόμου» είναι πραγματικά σημαντικός: Μια μικρή επιπλέον προσπάθεια που καταβάλλεται για συντήρηση ρουτίνας μπορεί να μας ανταποδώσει σε μακροπρόθεσμη βάση, προλαμβάνοντας μεγάλες καταστροφές.

Ο πρώτος λόγος για τον διαχωρισμό των αντικειμένων καθορίζεται από την ίδια τη συσκεή Kinetic. Όπως έχει αναφερθεί, κάθε αντικείμενο key-value Entry έχει περιορισμούς στο μέγεθος και του κλειδιού (key) και της τιμής (value). Συγκεκριμένα, το κλειδί μπορεί να είναι έως 4 KB και η τιμή έως 1 MB. Φανταστείτε ένα αντικείμενο το οποίο περιέχει μια μεγάλη συλλογή σε άλλα (μεγάλα) αντικείμενα. Προσπαθώντας να αποθηκεύσουμε το αρχικό αντικείμενο ως ένα ενιαίο οντότητα, ενδεχομένως να υπερβεί την χωρητικότητα ενός αντικειμένου Entry. Το οποίο, με τη σειρά του, θα προκαλέσει Exception.

Επιπλέον, διατηρώντας τα πράγματα οργανωμένα έχει νόημα και για την απόδοση του Kinetic handler. Ας υποθέσουμε ότι έχουμε ήδη αποθηκευμένο ένα αντικείμενο (ας το ονομάσουμε objectA), το οποίο περιέχει επίσης ένα άλλο αντικείμενο (ας το ονομάσουμε objectB). Επιπλέον, ας υποθέσουμε ότι με κάποιο τρόπο έχουμε επίγνωση αυτής της σχέσης Has-A μεταξύ objectA και objectB (το οποίο, παρεμπιπτόντως, είναι αρκετά δύσκολο κάνοντας άμεσα λειουργία put). Αν το objectB τροποποιηθεί, το dataClay θα επιχειρήσει να ενημερώσει το objectB στην υποδομή δεδομένων. Αλλά δεδομένου ότι το objectB είναι μέσα στο objectA (και ο handler το γνωρίζει), ο handler είναι υποχρεωμένος με την επιπλέον εργασία του να βρει ποια bytes είναι για το objectB και όχι για το objectA, και τελικά να κάνει την επιθυμητή λειτουργία ενημέρωσης.

Από την προηγούμενη παράγραφο και το παράδειγμά της, ελαφρώς υπονοήθηκε ότι δεν μπορούμε εύκολα να ξέρουμε τι έχει αποθηκευθεί όταν κάνουμε άμεσες λειτουργίες put. Με άλλα λόγια, είμαστε σε θέση να γνωρίζουμε το εξωτερικό αντικείμενο που αποθηκεύεται, αλλά όχι και αυτά

που περιλαμβάνονται στο εξωτερικό. Το να έχουμε αυτές τις πληροφορίες είναι ζωτικής σημασίας, όπως είδαμε προηγουμένως: Μόλις ένα αντικείμενο αποθηκεύεται, αυτό συσχετίζεται με ένα αναγνωριστικό (dataClay ID), το οποίο αποτελεί το κριτήριο για ένα αντικείμενο να είναι αποθηκευμένο. Εάν αυτό το βήμα δεν γίνει, είναι αρκετά πιθανό να έχουμε πολλαπλά αντίγραφα του ίδιου αντικειμένου στην υποδομή δεδομένων. Ας δούμε τα πιθανά προβλήματα ασυνέπειας με ένα παράδειγμα: το objectA (το οποίο περιέχει το objectB) αποθηκεύεται με άμεση λειτουργία put. Έτσι, δεν γνωρίζουμε ποια αντικείμενα αποθηκεύτηκαν εκτός από objectA. Ομοίως, ένα άλλο αντικείμενο (ας το ονομάσουμε objectC) το οποίο περιέχει επίσης μια αναφορά στο objectB αποθηκεύεται με άμεση λειτουργία put. Από τη στιγμή που δεν γνωρίζουμε ότι το objectB είναι ήδη αποθηκεύμενο (μέσα στο Entry του objectA), θα αποθηκευτεί ξανά (μέσα στο Entry του objectC). Στη συνέχεια, αν έχουμε τροποποιήσουμε το objectB σε οποιοδήποτε από τα δυο αντίγραφα, θα προκαλέσουμε ασυνέπεια στο άλλο.

Εν κατακλείδι, η σημασία του διαχωρισμού των αντικειμένων και της αποθήκευσής τους ξεχωριστά έχει πολύ νόημα. Αλλιώς, προβλήματα όπως η ασυνέπεια, η ισχνή απόδοση και η υπέρβαση των ορίων αποθήκευσης θα εγείρονται σαν ένα φαινόμενο ντόμινο. Ως εκ τούτου, η επιπλέον προσπάθεια για την τακτοποίηση των αντικειμένων όταν αυτά αποθηκεύονται για πρώτη φορά αξίζει με το παραπάνω.

1.4.6 Περιεχόμενο της τιμής (value) σε ένα key-value Entry

Σε αυτήν την ενότητα, παρουσιάζεται το pattern για την τιμή ενός key-value Entry. Είναι πολύ χρήσιμο να παρομοιάσουμε το δίσκο Kinetic με τον σωρό της Java. Στο σωρό, κάθε αντικείμενο περιέχει δεδομένα για τα πεδία του. Συγκεκριμένα, περιέχει πληροφορίες για κάθε πεδίο πρωταρχικού τύπου, καθώς επίσης και αναφορές σε άλλα αντικείμενα. Ομοίως, τα key-value entries στο δίσκο Kinetic πρέπει να συμπεριφέρονται σαν να ήταν αντικείμενα στο σωρό: Εάν ένα αντικείμενο έχει πεδία πρωταρχικού τύπου, τα πεδία αυτά σειριοποιούνται και αποθηκεύονται μέσα στο key-value Entry του αντικειμένου στο οποίο ανήκουν. Επιπλέον, τα key-value Entries παραπέμπουν σε άλλα key-value Entries στο δίσκο Kinetic και δεν περιέχουν άλλα Entries, όπως τα αντικείμενα δεν περιέχουν άλλα αντικείμενα στο heap.

Ωστόσο, τα σειριοποιημένα αντικείμενα πιθανώς να περιέχουν δεδομένα για παραπάνω από ένα αντικείμενο. Αυτό σημαίνει ότι οφείλουμε να επεξεργαζόμαστε τα δεδομένα αυτά, να διαχωρίζουμε τα περικλειόμενα αντικείμενα και να τα αποθηκεύουμε χωριστά. Αυτό απαιτεί μια πολύ προσεγμένη δουλειά αφού έχουμε να επεξεργαστούμε δεδομένα σε επίπεδο byte ως επί το πλείστον, αλλά και σε επίπεδο bit σε ειδικές περιπτώσεις.

Ευτυχώς, τόσο ο μηχανισμός σειριοποίησης όσο και οι δίσκοι Kinetic είναι της ίδιας φύσης. Και οι δύο καταλαβαίνουν bytes. Τίποτα άλλο. Τίποτα παραπάνω. Έτσι, αυτό που χρειαζόμαστε για να αποθηκεύσουμε στο δίσκο Kinetic καθορίζεται λίγο πολύ από το προηγούμενο στρώμα, το μηχανισμό σειριοποίησης. Δεν έχει και πολύ νόημα να προσπαθήσουμε για μια εντελώς διαφορετική απεικόνιση των αντικειμένων στο δίσκο Kinetic. Το μόνο που θα έκανε θα ήταν να επιβαρύνει την απόδοση, και στις δύο κατευθύνσεις: Για την αποθήκευση ενός αντικειμένου, θα έπρεπε να μεταφράσουμε το σειριοποιημένο αντικείμενο σε μία μορφή συμβατή για το Kinetic drive και ανακτώντας το αντικείμενο θα απαιτούσε το αντίστροφο. Αντ΄ αυτού, είναι αρκετά εύλογο να

κρατήσουμε το αποτέλεσμα του μηχανισμού σειριοποίησης και να το τροποποιήσουμε μόνο όταν αυτό είναι αναγκαίο.

Επιπλέον, ο μηχανισμός σειριοποίησης μας δίνει τη λύση για όταν θέλουμε να αναφερθούμε σε άλλα αντικείμενα: Κατά τη διάρκεια της σειριοποίησης ενός αντικειμένου, εάν βρεθεί μία αναφορά σε ήδη αποθηκευμένο αντικείμενο, το αναγνωριστικό (dataClay object ID) του αποθηκευμένου αντικειμένου προσαρτάται στο μήνυμα σειριοποίησης και η σειριοποίηση συνεχίζει στο επόμενο πεδίο. Ο Kinetic handler μιμείται αυτό το μοτίβο: Όταν ένα σειριοποιημένο αντικείμενο περιέχει bytes για άλλο σειριοποιημένο αντικείμενο, το ¨εσωτερικό' αντικείμενο αποθηκεύεται σε ένα ξεχωριστό key-value Entry και το ¨εξωτερικό' αντικείμενο αποθηκεύει μόνο το dataClay object ID του ¨εσωτερικού' αντικειμένου. Επιπλέον, ο Kinetic handler επιστρέφει τα object IDs όλων των πρόσφατα αποθηκευμένων αντικειμένων. Το dataClay λαμβάνει αυτό το αποτέλεσμα και εμπλουτίζει τις γνώσεις του σχετικά με τα αποθηκευμένα αντικείμενα. Χωρίς επεξεργασία του σειριοποιημένου αντικειμένου, θα ήταν αδύνατο να γνωρίζουμε ποια αντικείμενα αποθηκεύονται.

1.4.7 Επεξεργασία των σειριοποιημένων αντικειμένων

Στην προηγούμενη ενότητα, το περιεχόμενο του value των Entries καλύφθηκε διαισθητικά. Από τώρα και στο εξής, επικεντρωνόμαστε στο κομμάτι της επεξεργασίας του Kinetic handler. Κύριο καθήκον του Kinetic handler είναι να διακρίνει τα αντικείμενα που υπάρχουν μέσα σε ένα σειριοποιημένο αντικείμενο και να τα διαχωρίσει.

Κατά τη διάρκεια της επεξεργασίας των πεδίων ενός αντικειμένου, υπάρχουν δύο περιπτώσεις που μπορεί να συναντήσουμε. Η πρώτη και πιο εύκολη είναι να έχουμε ένα πεδίο πρωταρχικού τύπου. Σε αυτήν την περίπτωση, δεδομένου ότι γνωρίζουμε τον τύπο του πεδίου (χάρις σε κάποια μεταδεδομένα που διαθέτουμε), αντιγράφουμε από το σειριοποιημένο αντικείμενο στο key-value Entry το ακριβές ποσό των bytes που είναι αφιερωμένο για αυτό το πεδίο. Ευτυχώς, τα πεδία πρωταρχικού τύπου έχουν πάντα τιμή: Ακόμα και αν δεν έχουν αρχικοποιηθεί, έχουν την προεπιλεγμένη τιμή τους. Αξίζει να αναφέρουμε τη διαφορά μεταξύ των περιπτώσεων του Kinetic και των σχεσιακών βάσεων δεδομένων. Στο Kinetic, απλά πρέπει να αντιγράψουμε μερικά bytes και να συνεχίσουμε στο επόμενο πεδίο. Στον αντίποδα, ο Postgres handler απαιτεί την κατάλληλη διαμόρφωση μιας δήλωσης SQL, η οποία συνεπάγεται εκτεταμένο χειρισμό ενός string χρησιμοποιώντας αποστρόφους, παρενθέσεις, κ.λπ. Στο τέλος, αυτό οδηγεί σε ένα αρκετά μεγάλο και δύσκολο να κατανοηθεί κομμάτι του κώδικα, σκοπός του οποίου είναι ως επί το πλείστον ο σχηματισμός της δήλωσης SQL, αντί ο χειρισμός των δεδομένων από το σειριοποιημένο αντικείμενο.

Η δεύτερη περίπτωση είναι όταν έχουμε να κάνουμε με μια αναφορά. Η περίπτωση αυτή δεν είναι τόσο τετριμμένη όπως πριν και έχει πολλές υπο-περιπτώσεις. Όταν συναντάμε μια αναφορά, είναι πάντα αρκετά δυνατόν η αναφορά να είναι null. Ωστόσο, αυτή η πληροφορία δεν εξαρτάται από την μορφολογία της κλάσης που έχει ορίσει ο χρήστης. Αντ΄ αυτού, εξαρτάται σε μεγάλο βαθμό από το συγκεκριμένο στιγμιότυπο. Για αυτό το σκοπό κάθε σειριοποιημένο αντικείμενο έχει ένα bitmap, το notNullBitmap, το οποίο μας πληροφορεί για τα πεδία που είναι αναφορές σε άλλα πεδία: Ένα bit για κάθε αναφορά μας πληροφορεί αν η αναφορά είναι null ή όχι. Έτσι, το πρώτο βήμα, ενώ αντιμετωπίζουν αναφορά είναι να ελέγξετε το κατάλληλο bit στο notNullBitmap.

Εάν η αναφορά είναι null (σύμφωνα με το bitmap), δεν υπάρχουν bytes για την αναφορά στο σειριοποημένο αντικείμενο και ο handler συνεχίζει στο επόμενο πεδίο.

Από την άλλη πλευρά, υπάρχουν πολλές περιπτώσεις όταν η αναφορά δεν είναι null:

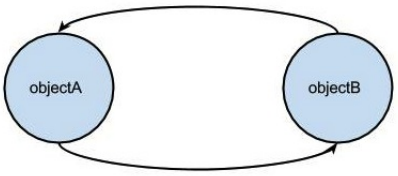
- **Περίπτωση 1 - Αναφορά σε ένα ήδη αποθηκευμένο αντικείμενο:** Εάν το πεδίο είναι αναφορά σε ένα αποθηκευμένο αντικείμενο, ο μηχανισμός σειριοποίησης έχει προσαρτηθεί μόνο το dataClay object ID του ήδη αποθηκευμένου αντικειμένου (μαζί με κάποια άλλα μεταδεδομένα όπως το tag).
- **Περίπτωση 2 - Νέα σειριοποιημένο αντικείμενο:** Αυτό συμβαίνει όταν το μήνυμα σειριοποίησης περιέχει bytes για παραπάνω από ένα αντικείμενο. Στην περίπτωση αυτή, το υπο-αντικείμενο πρέπει να διαχωριστεί από αρχικό σειριοποιημένο αντικείμενο, έπειτα να αντιγραφεί σε ένα άλλο key-value Entry και τότε να επανέλθει η επεξεργασίας του αρχικού αντικειμένου. Για το σκοπό αυτό, ο Kinetic handler λειτουργεί αναδρομικά, δεδομένου ότι και ο μηχανισμός σειριοποίησης ενεργεί κατά αναδρομικό τρόπο.
- **Περίπτωση 3 - Αναφορά σε ένα αντικείμενο που έχουμε επεξεργαστεί προηγουμένως:** Εάν η αναφορά παραπέμπει σε ένα αντικείμενο που έχει αντιμετωπιστεί στο παρελθόν, τότε μόνο το tag επισυνάπτεται για αυτό το αντικείμενο, αντί της εκ νέου σειριοποίησης του αντικειμένου.

Είναι πολύ σημαντικό να διακρίνουμε τη διαφορά ανάμεσα στην περίπτωση 1 και στην περίπτωση 3. Στην περίπτωση 3, ένα αντικείμενο θεωρείται ¨επεξεργασμένο' εάν ο handler έχει ήδη επεξεργαστεί αυτό το αντικείμενο στην ίδια κλήση της λειτουργίας αποθήκευσης. Με άλλα λόγια, ένα «επεξεργασμένο» αντικείμενο δεν αποθήκευτηκε πριν από την τρέχουσα κλήση της λειτουργίας αποθήκευσης. Από την άλλη πλευρά, ένα αντικείμενο θεωρείται αποθηκευμένο (περίπτωση 1), εάν έχει αποθηκευτεί σε προηγούμενη κλήση λειτουργίας αποθήκευσης.

Αλλά, πώς διακρίνονται οι τρεις αυτές περιπτώσεις· Ο Kinetic handler χρησιμοποιεί ένα map, που ονομάζεται alreadyEncounteredObjects, το οποίο έχει ως key το tag των ήδη επεξεργασμένων αντικειμένων και ως value το αναγνωριστικό (dataClay object ID) του κάθε αντικειμένου. Αν υπάρχει ένα tag στο map, αυτό σημαίνει ότι το αντικείμενο έχει υποστεί επεξεργασία στο παρελθόν (στην ίδια κλήση της λειτουργίας αποθήκευσης, όμως) και τα bytes μετά την ετικέτα είναι για το επόμενο πεδίο (περίπτωση 3). Σε αντίθετη περίπτωση, η αναφορά μπορεί να είναι είτε σε ένα αποθηκευμένο αντικείμενο (περίπτωση 1) ή σε ένα μη αποθηκευμένο (περίπτωση 2). Οι περιπτώσεις αυτές διακρίνονται από τα μεταδεδομένα που έχει κάθε σειριοποιημένο αντικείμενο.

Ο Kinetic handler πασχίζει να αποθηκεύσει τα αντικείμενα key-value σε μία (σχεδόν) κατανοητή μορφή για το dataClay. Με αυτόν τον τρόπο, η ανάκτηση ενός αντικειμένου από το δίσκο Kinetic θα απαιτήσει τη λιγότερη δυνατή επεξεργασία. Επομένως, τα tags σε κάθε key-value Entry θα πρέπει να ενεργούν ως μοναδικά αναγνωριστικά, όπως κάνουν στα σειριοποιημένα αντικείμενα. Ωστόσο, αν θέλουμε να εξαγάγουμε τα εσωτερικά αντικείμενα από ένα εξωτερικό, τα tags χάνουν την αναγνωριστική τους ιδιότητα.

Για παράδειγμα, ας υποθέσουμε ότι έχουμε τα αντικείμενα objectA και objectB, με τη μεταξύ τους συσχέτιση όπως απεικονίζεται στο Σχήμα 1.2.



Σχημα 1.2: Παράδειγμα: Σχέση μεταξύ δύο αντικειμένων.

Αν η λειτουργία αποθήκευσης κληθεί για το objectA, τότε και τα δύο αντικείμενα θα σειριοποιηθούν. Επιπλέον, το objectA θα επισημειωθεί με το tag 0, και το objectB με το tag 1. Η αναφορά του objectA στο objectB εμπίπτει στην περίπτωση 2, επειδή το objectB δεν το έχουμε αντιμετωπίσει στο παρελθόν, ούτε είναι αποθηκευμένο. Αντιθέτως, η αναφορά του objectB στο objectA εμπίπτει στην περίπτωση 3, επειδή το objectA το έχουμε ήδη συναντήσει. Έτσι, χρησιμοποιείται μόνο το tag 0 για την αναφορά αυτή.

Αν τα αντικείμενα objectA και objectB διαχωριστούν και αποθηκευτούν χωριστά, παύουν να παραπέμπουν το ένα στο άλλο πλέον. Για παράδειγμα, το tag με αριθμό 0 που περιέχει το objectB για την αναφορά του στο objectA δεν μπορεί να προσδιορίσει objectA. Για το λόγο αυτό, το object ID θα πρέπει να χρησιμοποιείται για την αναφορά σε άλλα αποθηκευμένα αντικείμενα, όπως κάνει ο μηχανισμός σειριοποίησης στην περίπτωση 1. Στη συνέχεια, κάθε μετέπειτα αναφορά σε ήδη επεξεργασμένο αντικείμενο πρέπει να περιέχει μόνο το tag. Για παράδειγμα, για μια δεύτερη αναφορά από το objectB στο objectA αρκεί να προσθέσουμε μόνο το tag του objectA, και όχι το object ID του. Έτσι, ο Kinetic handler θα πρέπει να εφαρμόσει αυτό τον εσωτερικό μηχανισμό επισημείωσης. Ένα map και ένα σύνολο (set) καθιστούν δυνατό αυτόν το μηχανισμό:

- alreadyEncounteredObjects (Map από tags σε object IDs): Αυτό το map περιγράφηκε και προηγουμένως: Είναι απαραίτητο για την αναγνώριση των αντικειμένων που έχουν επεξεργαστεί στο παρελθόν, στην ίδια κλήση της λειγουργίας αποθήκευσης. Ο αναγνώστης θα πρέπει να έχετε κατά νου ότι αυτό το map είναι μοναδικό και χρησιμοποιείται από όλες τις αναδρομικές κλήσεις της λειτουργίας αποθήκευσης.
- alreadyEncounteredTags (Σύνολο των tags): Κάθε φορά που καλείται η λειτουργία αποθήκευσης, ένα τέτοιο άδειο σύνολο αρχικοποιείται. Αυτό το σύνολο περιέχει όλα τα tags που έχει ήδη συναντήσει η τρέχουσα κλήση αποθήκευσης. Εάν ένα tag δεν υπάρχει στο σύνολο, το επισημειομένο αντικείμενο συναντάται για πρώτη φορά από την τρέχουσα κλήση της λειτουργίας αποθήκευσης, και το αναγνωριστικό αντικειμένου (dataClay object ID) θα πρέπει να επισυναφθεί, μαζί με το tag (το object ID μπορεί να βρεθεί από το map alreadyEncounteredObjects). Διαφορετικά, εάν το tag είναι στο σύνολο, το tag αρκεί για την αναφορά στο αντικείμενο.

Κάθε tag που είναι στο σύνολο alreadyEncounteredTags είναι και στο map alreadyEncounteredObjects. Αλλά, όχι το αντίστροφο.

Τέλος, ο αναγνώστης μπορεί να φανταστεί την πολυπλοκότητα της δήλωσης SQL για το χειρισμό των αναφορών στην περίπτωση του handler των σχεσιακών βάσεων δεδομένων.

1.4.8 Key schema

Έπειτα από την περιγραφή του τι αποθηκεύουμε για την τιμή ενός key-value Entry, το μόνο που απομένει να καλύψουμε είναι η δομή του κλειδιού για ένα Entry. Αυτή δεν εκπλήσει και ακολουθεί το αρκετά εύλογο μοτίβο: <**class ID**>_<**object ID**>

Το κίνητρο που οδήγησε σε αυτή τη δομή προέρχεται από τη διάταξη των key-value Entries με βάση το κλειδί τους. Αφού τα Entries είναι ταξινομημένα, τα αντικείμενα της ίδιας κλάσης ομαδοποιούνται, όπως παρομοίως κάνουν οι πίνακες των σχεσιακών βάσεων δεδομένων. Σύμφωνα με τις προδιαγραφές της τεχνολογίας Kinetic, η ανάκτηση των αντικειμένων που βρίσκονται κοντά το ένα στο άλλο μπορεί να βελτιστοποιηθεί. Σύντομα, θα δούμε συγκεκριμένες περιπτώσεις όπου εκμεταλλευόμαστε αυτό το χαρακτηριστικό.

1.4.9 Εμπλουτισμός κλάσεων

Στην αρχή αυτού του κεφαφαίου, αναφέρθηκε εμφατικά ότι στις τρέχουσες προσεγγίσεις για την ανταλλαγή δεδομένων, οι συνεργάτες είναι πάντα περιορισμένοι στην λειτουργικότητα που παρέχεται από τον κάτοχο των δεδομένων. Η μόνη λύση για αυτούς είναι να δημιουργήσουμε ένα αντίγραφο των δεδομένων και τότε να το χειριστούν με βάση την επιθυμία τους. Ωστόσο, η ουσιαστική συνεργασία και η ανταλλαγή δεδομένων παύουν να υπάρχουν με τη δημιουργία ενός αντιγράφου.

Αντ΄ αυτού, το dataClay επιτρέπει και στις δύο πλευρές να λειτουργούν απρόσκοπτα. Ακριβέστερα, ο εμπλουτισμούς των κλάσεων είναι δυνατός με τρεις τρόπους: 1) Την προσθήκη νέων πεδίων σε μια κλάση, 2) την προσθήκη νέων μεθόδων σε μια κλάση, 3) την προσθήκη μιας νέας υλοποίησης για μια υπάρχουσα μέθοδο σε μια κλάση. Προφανώς, ο Kinetic handler ασχολείται μόνο με την πρώτη περίπτωση, επειδή η δεύτερη και τρίτη περίπτωση επηρεάζουν τη συμπεριφορά των κλάσεων που δεν έχει τίποτα να κάνει με την υποδομή των δεδομένων. Αντ΄ αυτού, οι αλλαγές στα πεδία μίας κλάσης επηρεάζουν την κατάσταση των στιγμιοτύπων του. Και, όντως η υποδομή των δεδομένων (στην περίπτωσή μας, οι δίσκοι Kinetic) αποθηκεύουν μόνο την κατάσταση των αντικειμένων. Αντιθέτως, οι αλλαγές στις μεθόδους αντανακλώνται στα αρχεία των κλάσεων.

Στην περίπτωση των σχεσιακών βάσεων δεδομένων, η αναπαράσταση μιας κλάσης επιτυγχάνεται χρησιμοποιώντας ένα αντίστοιχο πίνακα, όπου τα πεδία της τάξης αντιστοιχίζονται με τις στήλες του πίνακα. Έτσι, εμπλουτίζοντας μια τάξη με την προσθήκη νέων πεδίων αντικατοπτρίζεται στη βάση δεδομένων, με την προσθήκη νέων στηλών στον αντίστοιχο πίνακα. Αντίθετα, στην περίπτωση των Kinetic drives οι κλάσεις δεν αναπαριστόνται με κάποιο τρόπο. Κάθε στιγμιότυπο είναι ανεξάρτητο από το άλλο σε έναν δίσκο Kinetic, ακόμη και αν βρίσκονται το ένα δίπλα στο άλλο. Ωστόσο, οι τροποποιήσεις στη δομή μίας κλάσης επηρεάζουν όλα τα στιγμιότυπά του, ακαριαία. Ακόμη περισσότερο, η τροποποίηση μίας κλάσης επηρεάζει και τα αντικείμενα των κλάσεων που επεκτείνουν την εμπλουτισμένη κλάση. Έτσι, αν θέλαμε να ενημερώσουμε όλα τα αντικείμενα που επηρεάζονται από τον εμπλουτισμό μίας κλάσης, είναι αδύνατον, επειδή δεν

μπορούμε να βρούμε εύκολα τα αντικείμενα άλλων κλάσεων που επεκτείνουν την εμπλουτισμένη κλάση.

Για αυτό το λόγο, δεν είμαστε σε θέση να κάνουμε κάτι με λογική πολυπλοκότητα, τη στιγμή του εμπλουτισμού. Το μόνο που μπορούμε να κάνουμε τη στιγμή του εμπλουτισμού είναι να ενημερώσουμε τη γνώση που έχουμε για τη δομή της εμπλουτισμένης κλάσης. Αυτή η ενημερωμένη έκδοση δεν επηρεάζει κανένα από τα "dirty" αντικείμενα. Ωστόσο, αυτή η νέα γνώση μας επιτρέπει να αναγνωρίσουμε τα "dirty" αντικείμενα. Η ενημέρωση ενός "dirty" αντικειμένου συμβαίνει κατά τη στιγμή της ανάκτησης.

Πριν προχωρήσουμε στη λύση που έχουμε υιοθετήσει, είναι σημαντικό να διευκρινίσουμε τι αναμένει το dataClay από την υποδομή των δεδομένων όταν αιτείται ένα αντικείμενο του οποίου είτε η κλάση ή κάποια από τις υπερκλάσεις του έχουν εμπλουτιστεί. Αναμένει ένα σειριοποιημένο αντικείμενο το οποίο έχει πληροφορίες (bytes) για όλα τα πεδία της κλάσης, τόσο παλαιά όσο και νέα. Πιο συγκεκριμένα, η μέθοδος αποσειριοποίσης μίας εμπλουτισμένης κλάσης έχει ενημερωθεί έτσι ώστε να αποσειριοποιεί «εμπλουτισμένα» αντικείμενα. Έτσι, όταν ζητάμε ένα "dirty" αντικείμενο, πρέπει να προσθέσουμε τις προεπιλεγμένες τιμές στα νέα πεδία: Εάν το νέο πεδίο είναι πρωταρχικού τύπου, παίρνει την προκαθορισμένη τιμή. Διαφορετικά, το νέο πεδίο αποτελεί σημείο αναφοράς και δεν θα προσθέσουμε επιπλέον bytes, αφού οι αναφορές null σε σειριοποιημένα αντικείμενα δεν έχουν bytes. Επιπλέον, το notNullBitmap πρέπει να περιέχει bits για κάθε αναφορά, ακόμη και τις νέες.

Πώς αναγνωρίζεται ένα “dirty” αντικείμενο από το Kinetic handler; Προκειμένου να αναγνωρίσει ένα επηρεασμένο αντικείμενο ο Kinetic handler, μερικά μεταδεδομένα προστίθεται στο value καθενός key-value Entry, κατά την αποθήκευσή του. Γνωρίζουμε ότι όταν ένα αντικείμενο αποθηκεύεται η κλάση δεν είναι εμπλουτισμένη, αφού χρησιμοποιούμε την τελευταία της εκδοχή. Για αυτό το λόγο μαρκάρουμε κάθε αντικείμενο με τον αριθμό των πεδίων για κάθε μία από τις κλάσεις του (δηλαδή την κλάση του και τις υπερκλάσεις). Στη συνέχεια, όταν ένα αντικείμενο ανακτάται, αυτά τα αποθηκευμένα μεταδεδομένα συγκρίνονται με την τρέχουσα γνώση που έχουμε για την ίδια κλάση. Αν εντοπιστεί έστω και μία διαφορά αυτό σημαίνει ότι το αντικείμενο είναι “dirty”.

Τέλος, πώς αναγνωρίζουμε τα νέα πεδία όταν πρέπει να ενημερώσουμε ένα dirty αντικείμενο; Η λύση έρχεται από το ίδιο το dataClay. Κάθε νέο πεδίο προστίθεται στο τέλος των ήδη υπαρχόντων πεδίων. Η αλλαγή αυτή αντικατοπτρίζεται επίσης στη γνώση που έχουμε για τη μορφολογία μίας κλάσης. Έτσι, δεδομένου ότι γνωρίζουμε τον αριθμό των ήδη αποθηκευμένων πεδίων, τα οποία ταυτίζονται με τα πρώτα πεδία που γνωρίζουμε από τη γνώση για τη μορφολογία των κλάσεων, μπορούμε επίσης να διακρίνουμε τα νέα πεδία. Επιπλέον, τα πεδία δεν αφαιρούνται ποτέ από τις κλάσεις που έχουν οριστεί από τους χρήστες. Ως εκ τούτου, όταν ένα πεδίο προστίθεται σε μία κλάση, η θέση του στην κλάση σε σχέση με τα άλλα πεδία δεν θα αλλάξει ποτέ.

1.4.10 Ανάκτηση των αντικειμένων από το Kinetic drive

Πλέον το μόνο που απομένει να καλυφθεί είναι ο αλγόριθμος που ακολουθείται για την ανάκτηση αντικειμένων από το δίσκο Kinetic. Όταν επιθυμούμε να λάβουμε ένα αντικείμενο από το δίσκο Kinetic, το πρώτο βήμα που κάνουμε είναι να ανακτήσουμε το αντίστοιχο Entry από το δίσκο.

Έπειτα ακολουθεί ο έλεγχος αν είναι αυτό το αντικείμενο είναι "dirty". Με άλλα λόγια, ελέγχουμε εάν η κλάση στην οποία ανήκει το αντικείμενο έχει εμπλουτιστεί με νέα πεδία. Στην περίπτωση που δεν παρατηρείται κάποια αλλαγή για την κλάση του αντικειμένου, αλλά και όλες τις υπερκλάσεις που επεκτείνει αυτή, το αντικείμενο δεν χρειάζεται κάποια περαιτέρω επεξεργασία. Είναι έτοιμο να μεταβιβαστεί στο dataClay. Στον αντίποδα, αν παρατηρηθεί ότι έστω μία από τις κλάσεις του αντικειμένου έχει τροποποιηθεί, τότε μεσολαβούν δύο βήματα πριν περαστεί το τελικό αντικείμενο στο dataClay. Το πρώτο βήμα είναι να προστεθούν οι default τιμές για τα νέα πεδία της κλάσης. Το δεύτερο βήμα είναι αποθηκευτεί εκ νέου το αντικείμενο Entry στο δίσκο Kinetic, το οποίο θα περιέχει πληροφορίες για όλα τα πεδία, νέα και παλιά. Αφού τελειώσει η αποθήκευσή του, το ίδιο αντικείμενο επιστρέφεται και στο dataClay.

Παρατηρούμε ότι ενώ το τελικό αποτέλεσμα είναι διαθέσιμο με το πέρας του πρώτου βήματος, περιμένουμε την περάτωση και του δεύτερου βήματος πριν διαβισβαστεί το τελικό αντικείμενο στο dataClay. Η αλήθεια είναι ότι αυτή η υλοποίηση επιδέχεται βελτιστοποίηση. Πιο συγκεκριμένα, το ανανεωμένο αντικείμενο μπορεί να μεταβιβαστεί στο dataClay αμέσως μετά το πρώτο βήμα και το δεύτερο βήμα να εκτελεστεί με ασύγχρονο τρόπο. Ο μόνος λόγος που δεν έχει υλοποιηθεί η βελτιστοποιημένη ανάκτηση δεδομένων ήταν η έλλειψη χρόνου.

1.5 Αξιολόγηση

Εν τέλει, ποιο είναι το συμπέρασμα μετά την ολοκλήρωση αυτής της έρευνας; Ακόμη και αν τα αποτελέσματα που προκύπτουν από τις δοκιμές, όπως παρουσιάζονται στο πέμπτο κεφάλαιο, δεν είναι τα βέλτιστα, η ενσωμάτωση της τεχνολογίας Kinetic της Seagate στο dataClay φαίνεται πολύ ελπιδοφόρα. Πρώτα απ΄ όλα, η τεχνολογία Kinetic πληροί το επιθυμητό χαρακτηριστικό για byte addressability, δηλαδή αναφορά στα δεδομένα σε επίπεδο byte. Με άλλα λόγια, ο μηχανισμός σειριοποίησης και η υποδομή δεδομένων, δηλαδή, οι δίσκοι Kinetic, μιλούν την ίδια γλώσσα που είναι bytes. Αλλά, επιπλέον και πιο ενδιαφέροντα, υπάρχει τεράστιο περιθώριο για βελτίωση και για τις δύο τεχνολογίες και, επίσης, για την ενσωμάτωσή τους. Περαιτέρω λεπτομέρειες ακολουθούν.

Η τεχνολογία Kinetic της Seagate εξακολουθεί να έχει μεγάλη πρόοδο να διαγράψει (όπως και το dataClay). Παρόλο που η τεχνολογία Kinetic έχει υιοθετήσει την απλή αφαίρεση των αντικειμένων key-value, η αλήθεια στο παρασκήνιο είναι λίγο πιο περίπλοκη. Στην πραγματικότητα, η αποθήκευση αντικειμένων key-value δεν έχει επιτευχθεί σε επίπεδο hardware, ακόμα. Αντ΄ αυτού, ένα ενσωματωμένο σύστημα που τρέχει υπό το λειτουργικό σύστημα Linux είναι υπεύθυνο για την αποθήκευση των αντικειμένων key-value σε μια βάση δεδομένων LevelDB. Με άλλα λόγια, η υποδομή key-value είναι ορισμένη στο λογισμικό προς το παρόν. Αυτό το μεσαίο στρώμα μονάχα επιβαρύνει την απόδοση του Kinetic handler. Όταν η υποδομή key-value υλοποιηθεί σε επίπεδο υλικού, η λειτουργία από και προς το Kinetic drive αναμένεται να βελτιστοποιηθεί.

Περαιτέρω βελτιστοποίηση μπορεί να επιτευχθεί με την αλλαγή της τεχνολογίας του ίδιου του δίσκου. Αυτή τη στιγμή, οι Kinetic drives εφαρμόζονται σε κλασικούς Σκληρούς Δίσκους. Εάν εφαρμοστεί η τεχνολογία Kinetic σε δίσκους στερεάς κατάστασης (Solid State Drives) θα παρουσιαστεί κατά πάσα πιθανότητα περαιτέρω βελτίωση στις επιδόσεις.

Μεγάλο περιθώριο για βελτιστοποίηση υπάρχει και στην πλευρά του dataClay. Ο μηχανισμός σειριοποίσης έχει σχεδιαστεί κατά τέτοιο τρόπο που τα αποθηκευμένα δεδομένα θα φέρουν την

σημασιολογία τους. Για παράδειγμα, τα SCOs που έχουν αποθηκευτεί σε μια σχεσιακή βάση δεδομένων φέρουν σημασιολογική πληροφορία. Ο κύριος λόγος της επιλογής αυτού του σχεδιασμού ήταν η δυνατότητα για τις εφαρμογές να έχουν πρόσβαση άμεσα στα δεδομένα στην υποδομή των δεδομένων. Ωστόσο, μια τέτοια ανάγκη για εφαρμογές που επιθυμούν την άμεση πρόσβαση στην υποδομή δεδομένων δεν έχει προκύψει μέχρι στιγμής. Επιπλέον, η πρόσβαση σε δεδομένα με τον τρόπο αυτό διασπά το επιθυμητό χαρακτηριστικό του υπολογισμού κοντά στα δεδομένα. Έτσι, ο ισχύων μηχανισμός σειριοποίησης φαίνεται να είναι παρωχημένος. Πράγματι, ο μηχανισμός σειριοποίησης υποχρεώνει επί του παρόντος τους handlers να κάνουν πολλή δουλειά που θα μπορούσε να αποφευχθεί. Πιο συγκεκριμένα, η επεξεργασία λαμβάνει χώρα αυτή τη στιγμή και στις δύο πλευρές, και στο μηχανισμό σειριοποίησης και στους handlers. Αυτό είχε νόημα μόνο στην περίπτωση που αποθηκεύουμε σημασιολογικώς πλούσια δεδομένα. Επειδή δεν υπάρχει μια τέτοια ανάγκη, όπως εξηγήθηκε προηγουμένως, είναι απαραίτητος ένας πιο αποτελεσματικός μηχανισμός σειριοποίησης. Για το σκοπό αυτό, το Storage Systems research group του BSC έχει ξεκινήσει την ανάπτυξη ενός νέου μηχανισμού σειριοποίησης που θα αποφεύγει τη διπλή επεξεργασία και θα κάνει την εργασία των handlers πολύ ευκολότερη. Συγκεκριμένα, δεν θα υπάρχει πλέον καμία ανάγκη για επεξεργασία των σειριοποιημένων αντικειμένων στο επίπεδο των handlers. Οι handler θα είναι μόνο υπεύθυνοι για το πέρασμα των σειριοποιημένων αντικειμένων στην υποδομή δεδομένων και το αντίστροφο.

Chapter 2

Introduction and Motivation

Since the widespread use of the term "Big data" from 2011 and on [1], big data has evolved into a promising actor for radical changes in every activity. Big data has tremendous potential to transform businesses and to power revolutionary customer experiences. Insights from big data can enable companies to make better decisions — deepening customer engagement, optimizing operations, preventing threats and fraud, and capitalizing on new sources of revenue. All these insights were hidden previously due to the high cost of processing that data.

One core benefit of dealing with big data is its use for analysis purposes. In comparison to the older statistical approach of sampling, processing of every single item of data in reasonable time is now feasible and it leads to safer conclusions. Having big amounts of data can beat out even the best model. On the other side, big data can also drive into new products or services, which can change dramatically our everyday life. For example, Facebook has been able to craft a highly personalized user experience and create a new kind of advertising business, by combining a large number of signals from a user's actions and those of their friends.

On the other side, new challenges have been arisen by the emergence of big data. According to Edd Dumbill [2],"Big data is data that exceeds the processing capacity of conventional database systems." The *volume* of data produced is unprecedented and what seems big today will probably be considered normal in the not-so-distant future. Estimations on the size of the digital universe mention a growth from 130 exabytes in 2005 to 40000 exabytes in 2020 [3]. This is translated into a growth by a factor of 300. Furthermore, data is also produced extremely fast. It is created in such a high *velocity* that a new Moore-like law has arisen: The total amount of data will be doubled every two years [3]. In addition, data is arriving from multiple sources: Social networks, mobile devices, financial market data, traffic flow sensors, anything, in general, can create data. This *variety* of sources is also reflected on the multiple forms of data: structured (e.g. databases), semi-structured (e.g. XML, JSON) and unstructured (e.g. text, video, sound, images etc) data consist the ocean of information.

Both academia and industry are putting much effort in order to tackle the multiple challenges that big data has brought. To name some of them, scalability, performance and heterogeneity are challenges that scientists and engineers are called to deal with. A short description of them follows.

Managing big and fastly increasing loads of data has been an issue for many decades. Until the recent past, Moore's law was the resolver of this problem. The increasing processing capacity was high enough so that it was able to catch up the increasing data volume. But since the CPU speeds have been limited due to the power constraints, the new approach of dealing the *scalability* problem has been resolved by changing the dimension of processing: Processors are built with increasing numbers of cores. Though, parallelism is like passing the problem from hardware to software. And software is now called to deal with the scalability challenge.

Bottlenecks on *performance* are also introduced because of the big data emergence. For example, the need for rapid real-time value from data results in performance challenges as the amount of data that moves into the system increases. First, there is the challenge of whether there is enough I/O and network bandwidth when data is pushed to storage. Second, since the only way to accomplish such a huge workload is by distributing it in several nodes, this leads to the need for a quite sophisticated network design, where possible failures must be predicted.

Both challenges described above are more or less of technical nature. Nevertheless, problems are also arisen when somebody analyzes data. Such a problem is data *heterogeneity*. When humans consume information, a great deal of heterogeneity is comfortably tolerated. In fact, the nuance and richness of natural language can provide valuable depth. However, machine analysis algorithms expect homogeneous data, and are poor at understanding nuances. In consequence, data must be carefully structured as a first step in (or prior to) data analysis.

Taking into consideration the big data benefits and challenges mentioned above, the motivation for conducting a master thesis on big data seems reasonable. This thesis, though, examines some of the challenges that have not been described above. It makes a fair attempt to see how collaboration between data owner and their partners can be facilitated. It also tries to provide programmers all the tools that will make their work easier and spend their time on the logic of their program rather the data storing. Last but not least, performance is never sacrificed, and the intention is always to keep it high. Further details follow.

Throughout this master thesis, two developing technologies are being examined. The first one is dataClay, a platform for data sharing which has been developed by Storage Systems Research Group at Barcelona Supercomputing Center. dataClay has been built having in mind data sharing as a fundamental feature. The second emerging technology is the Seagate Kinetic Open Storage platform (sometimes called just Kinetic from now and on). Kinetic technology is revolutionizing the way we know data storage and is trying to face many of the challenges that big data has raised. Across the rest of this chapter, the motivation of developing the aforementioned technologies is presented which also happens to be the motivation for using them in this master thesis.

2.1 dataClay

What seems neglected by big data research community so far is the effort to make easier the collaboration among all the actors. dataClay has been built having data sharing as a key feature. This research focuses on how data can be shared in an easy and effective way.

Moreover, manipulating data on current platforms is highly dependent on the layer where it relies. In dataClay approach, data is handled in a transparent way which permits programmers to focus on the logic of their applications rather than coping with the data transfer. Last but not least, performance is what really matters. Recent trends compel moving the computation close to the data rather than the reverse way. dataClay faces this challenge by its design.

2.1.1 Data sharing

Nowadays, somebody can basically share intellectual property, such as data models, specific data, code, etc. in three ways: a) by sharing the data infrastructure, b) by choosing the datasets that can be copied or downloaded, or c) by offering restricted data services.

Sharing the data infrastructure provides full access to everybody. However, even though it is a very flexible approach, its core requirement is the firm trust among all the stakeholders. This probably happens when data is open/public. But, in this case, data is probably read-only. If stakeholders want to modify it, the need for creating a copy into their workstation rises. Which leads to extra time and space needed for this operation.

In the second case, data providers can only decide about the consumers to be granted the authority to copy or download specific datasets. This option supplies to the consumers the flexibility to process the data in their infrastructure according to their needs. However, downloading data implies too much data movement, especially in cases that the consumer's infrastructure cannot store the whole dataset. Furthermore, data owners lose the control of their data, because it leaves their infrastructure when it is copied.

Finally, when using a data service (such as RESTful web services), the data can be accessed in the provider's infrastructure and the owner maintains a strict control by deciding not only what but also how this data is being shared. Despite the fact that such an approach prevents the data movements and the data owner remains the only manager of the data, it restricts the data consumers to use only the functionality provided by the data owners. A modification to the given functionality is only possible with the involvement of the data provider.

Listening to the above needs and misbehaviors, dataClay is designed in such a way that ensures data provider is the one who controls the data but, also, gives to third parties the potential to enrich them, either by adding functionality or granting them modifications privileges. Last but not least, dataClay avoids any redundant data transfer.

2.1.2 Persistent vs. non-persistent data models

Today, data models are designed in a different way depending on whether they are treated within a persistent (non-volatile) environment or within a non-persistent (volatile) one.

Common cases of persistent storage include file systems and databases. Accessing data from files demands I/O operations to be done by a developer. On the other hand, querying data from databases is needed when someone deals with them, which also impose extra effort from the developer.

In the case of non-persistent storage, data relies on memory. The applications themselves (usually) allocate free memory for storing data and, since then, the data is processed through references, pointers, iterators etc.

Given the differences between volatile and non-volatile data models, developers are compelled to devote too much effort, first, design the two different data models and the mapping between them, and then to implement the whole data flow for their applications with transitions between persistent and non-persistent data.

dataClay provides all the mechanisms that are needed in order to handle persistent data as non-persistent, thus facilitating application development and simplifying the design of data models.

2.1.3 Computing close to the data

In non-big data cases, data processing involves the data loading from the persistent layer into the memory and then its processing from CPUs. However, in big data era this approach is, at least, inefficient, if not unfeasible. Data is produced in such a high pace that data transfer close to the CPU is slower. Modern trends impose the movement of computation close to the data, and not the reverse. Popular solutions that implement this include Apache Hadoop and Active Storage. dataClay fulfills this need by its design: data are joined with code thanks to its key technology: self-contained objects (SCOs). SCOs are like regular OOP objects, which are enriched with some components which enable the two aforementioned long desired features, the *seamless data sharing* and the *abstraction of data from their environment*. SCOs are described detailly in the following chapter.

2.2 Seagate Kinetic Open Storage platform

Nowadays, we see an explosion of data that has been created of mobile, social, video applications, Internet of Things, connected devices, cloud computing and big data. These applications rely on data that is primarily unstructured (or semi-structured), and easy and inexpensive to create. The new status of data also drives the evolution of the storage infrastructure.

Today's storage architecture was designed decades ago, for a very different use case, not for a globally distributed, large scale cloud architecture and environment. In order for the industry to achieve the growth demanded to support the new storage demands, layers of inefficiency from legacy architectures must be removed and a new approach optimized for scale-out application and data center needs must be introduced.

The Seagate Kinetic Open Storage platform and its developer tools make this radical change. It takes traditional hard drives and adds two key elements: 1) Object Storage Protocol and 2) Ethernet connection. The combination of these two elements lets the entire storage architecture become more efficient.

The Kinetic technology is detailed in the next chapter. However, at that moment, what the reader only needs to understand about Kinetic is that it is a new class of key-value Ethernet

connected drives. Instead of dealing with file semantics or a file system for finding where data resides on the device, Kinetic offers a simple key-value abstraction for working with objects (information). This abstraction is driven by the needs of modern applications: They just need simple object semantics (e.g., write the whole thing, read the whole thing, delete the whole thing etc).

The second key element of Kinetic is the use of Ethernet protocol. In traditional storage architectures, data which starts from an application passes through several layers of hardware and software in order to reach its destination, the storage device. Kinetic eliminates these multiple layers in the path between application and storage devices and uses Ethernet instead. So, information is just an IP address away. This enables applications to target storage devices directly and take advantage of storage features.

2.3 Objective of the master thesis

One of the novelties introduced with dataClay is the abstraction of data from the layer they rely on. They can be either in memory or in a persistent environment. From programmer's point of view, data can be accessed in the same way regardless the environment they rely on. On the other side, although the data is accessed like being in memory, it cannot reside in it forever, obviously. It is dataClay's task to offer the functionality which enables the abstraction feature.

dataClay is a purely object-based platform. As it was mentioned in the section "Computing close to the data", it deals with SCOs. Which, actually, are regular OOP objects. Thus, information is enclosed inside objects. Working with data is actually working with objects.

In dataClay, when data needs saving, the corresponding objects must be saved. For this purpose, dataClay has been using several data infrastructures, so far. For example, Postgres relational databases, Cassandra distributed databases, and others. All these cases involved the mapping of dataClay objects to the internal structure of the underlying data infrastructure. For example, OOP objects must be mapped into rows in the corresponding table in a relational database. On the contrary, Kinetic technology has adopted the much more simple key-value object abstraction. Thus, it seems to be in advantageous position in comparison to the other data infrastructure, regarding dataClay. Nevertheless, integration of Kinetic technology into dataClay is not as easy as it seems, even though dataClay and Kinetic talk the "language of objects".

When dealing with objects, the easiest and most portable way to save them is to serialize them. However, default serialization of OOP languages is not capable to support dataClay's features, like data sharing. Thus, dataClay has developed its own serialization mechanism. In order to accomplish the integration of Kinetic technology into dataClay, solid understanding of the underlying technologies and of their mechanisms is needed. For this reason, Chapter 2 describes both technologies from high level. Then, dataClay's custom serialization mechanism is presented in Chapter 3 in detail. Kinetic handler, which accomplishes the pairing of these two technologies, is described in Chapter 4. Next, evaluation of Kinetic handler follows in fifth chapter.

Chapter 3

Related Technology

The purpose of this chapter is to present dataClay and Seagate Kinetic Open Storage platform in detail. In addition, the underlying technical mechanisms of some key features of both technologies are presented.

3.1 dataClay

In the previous chapter, it was mentioned that self-contained objects (SCOs) are the key elements of dataClay. They are presented in detail in the following section. Furthemore, the possible ways a user can enrich a class with are presented. In the last section, details on how dataClay can be used are presented and, along with them, some of the underlying key mechanisms are explained.

3.1.1 Self-contained objects

What drives the introduction of self-contained objects (SCOs) is the widespread use of the Object Oriented Paradigm (OOP). OOP objects are made out of two key features. Firstly, they have *state* via their fields. Secondly, they also have *behavior*, via the methods they are equipped with. dataClay takes advantage of this *data-computation proximity* derived by the OOP design and adds some new features to the traditional objects. This blending results into the concept of SCOs (Figure 3.1).

SCOs are like regular objects in the sense that they are instances of a certain set of data models (the traditional OOP classes) and applications use them as in the common OOP. One thing added on top of the regular objects is the *policies* that enable the long desired feature of efficient data sharing. *What* is shared, or *with whom*, or for *how long*, etc. is controlled by the policies that the data provider defines.

Furthermore, SCOs are also provided with a user-friendly *dataClay API* that enables the programmers to store and retrieve them handily. In order to create new SCOs the user calls a single method that makes the whole object and its relationships with other objects persistent

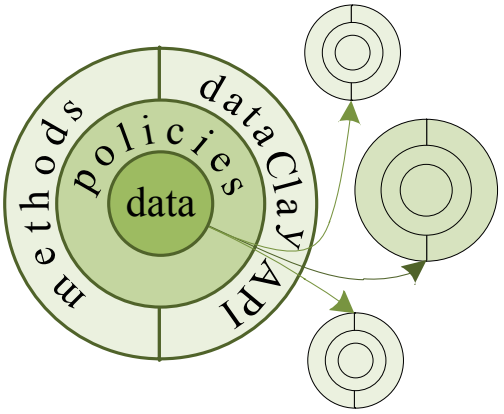


FIGURE 3.1: A self-contained object and its relationship with other.

in a transparent way. Then, SCOs can be retrieved by using three mechanisms: a) by using a reference from another SCO, b) by tagging them with an alias and then querying by this tag, or c) by performing *"query by example"* that is resolved searching SCOs that match with a certain dummy SCO used as a prototype.

Now, it is easy to see, from a different perspective, how SCOs fulfill the three motivational factors that led to the development of dataClay. Data sharing is possible due to the policies that every SCO bears. Dealing with data (either persistent or not) in a transparent way is enabled due to the dataClay API that every SCO inherits. Last but not least, computation close to the data is achieved due to the nature of traditional OOP objects: fields and methods reside *together* in an object.

Dealing with SCOs is actually dealing with OOP objects. Once a SCO is retrieved, the user can manipulate it like a regular OOP object by using its fields or calling the methods that its class has defined. In this way, dataClay saves lots of efforts to the programmers avoiding the transitions between current persistent and non-persistent data models since now they can focus on a single combined model with a SCO oriented basis.

3.1.2 3rd party enrichment

As it was described in the previous chapter, sharing data has not been an easy and effective procedure so far. Current solutions imply too much data movement and multiple copies of data. But most crucially, consumers are restricted by the functionality provided by the owner. dataClay deals with these issues and offers both data owners and 3rd parties the potential to enhance existing data models with intellectual property. Since dataClay follows the object-oriented paradigm, data models are represented by classes. Thus, enrichment of a class is done by modifying its elements, namely the fields or/and the methods. A class can be enriched in three ways: a) by adding new fields, b) by adding new methods, and c) by adding new implementations to existing methods.

Even though adding new fields to a class affects its structure, this enrichment does not have any negative impact on the rest of the collaborators. In the same way, adding new methods offers one more functionality, which does not exclude the one provided by the data model owner. Thus, the new methods can be executed on the existing SCOs as well as the original methods. In fact, the new methods become part of the original class as the original ones. Last but not least, dataClay offers users the potential to add a new implementation to an existing method. Nevertheless, this modification does not affect the rest of the users who have access on the same class. Instead, when a user calls a method which has been modified previously by someone else, the expected implementation (for him) will be executed. Soon, the underlying mechanism which enables class enrichments will be described.

3.1.3 dataClay details

Across this section, the typical workflow of data model sharing will be explained and some key underlying mechanisms will be explained too. The example will be described in Java, since the implementation part of this master thesis has been done in this programming language. However, dataClay supports Python too.

Data model sharing

When data providers aim to share their data models with 3rd parties, they begin by registering the corresponding Java classes in dataClay. The data provider specifies the location of the class files and dataClay proceeds with the registration process.

Given that a class name is not a universal resource identifier, dataClay uses namespaces as the entities to organize class names as if they were part of a particular package or application. These namespaces are also registered by the data model provider.

Once providers have registered their classes, they can sign **model contracts** that grant 3rd parties to access them during a certain period of time. At that moment, the provider defines: a set of interfaces, one per class, which include the fields and methods that will be exposed through the contract, as well as the expiration date of the contract. Therefore, the providers retain control of what they share, with whom, and for how long.

In order to manage the entire process, shown in Figure 3.2, dataClay offers an API via a client library tool that provides the functionalities to register namespaces, classes, interfaces and contracts.

Stubs

Beneficiaries of model contracts can retrieve the included classes to use them either to compile their applications or to generate new enrichments. In particular, and also by means of the client library tool, consumers download one **stub** per class (a bytecode class file representing the original class). These stubs are generated by dataClay considering the *visibility scope* derived from the contracts, i.e. the union of visible methods and fields according to the

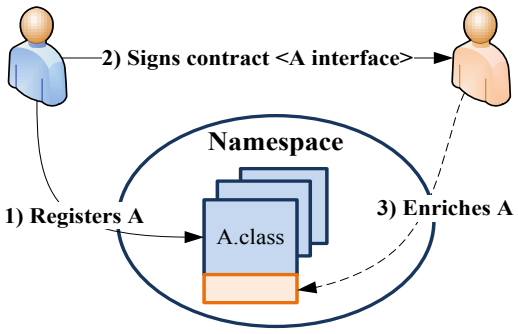


FIGURE 3.2: Sharing data models.

interfaces included; plus a set of specific methods (**dataClay API**) that are inherited from a common class (*DataClayObject*) that all stubs extend (analogous to *Object* class in Java):

- *makePersistent*: stores the object as a new SCO in the system.
- *deleteObject*: deletes the referenced SCO from the system.
- *getByAlias*: retrieves a SCO reference of the same class by its alias.
- *getAlike*: retrieves the references of those SCOs whose values match with those of the current instance acting as a prototype.

All the methods in a stub, except those provided by the dataClay API, have two different parts controlled by an if-clause: *local execution* and *remote execution*. The former, contains the bytecode of one of the accessible implementations of the method (considering the visibility scope from the contracts) and it is necessary while the object remains local, that is, until it is not made persistent with the *makePersistent* method. On the contrary, when the object is already a SCO, it is called the remote execution behaving like a RPC (the following section *Remote execution* describes it).

The stub also comprises (hardcoded) the information related to the model contracts used to generate it — since, as we said, a stub might be generated from the union of several contracts potentially containing different interfaces of the corresponding class.

Enrichments

Regarding the enrichments, introduced in the previous section, the process is analogous to registering original classes, since at the end any enrichment is an extra piece for a data model which can be defined within a class. In particular, when 3rd parties aim to add new value to an existing class (for which they have authority via a model contract) registered in dataClay, they use the Java extension mechanism (i.e. Java *extends* token) to define the corresponding enrichments. That is, the class containing the enrichment extends from the stub corresponding to the class being enriched, so that the enrichment class is allowed to use the original fields of

the class and the original methods as if it was a child class. This is useful not only to compile the enrichment class and look for errors if any, but also to register the enrichment by using an analogous process as a regular class registration.

In the data infrastructure, the enrichment is deployed by updating the original class to include the new functionalities and fields, thus enabling to share these new parts through immediate subsequent new model contracts containing them (with interfaces including the new defined fields and methods). That is, although the original class and SCOs are extended, the existing interfaces and model contracts are not affected. Thus, the existing applications using older stubs are not compromised, with the new parts of the data model being simply out of their visibility scope.

The enrichments of fields and methods are simple, in the sense that they add new value to existing data models but the resulting class is like a regular one. On the contrary, the enrichment of implementations of existing methods is a bit more complex because it entails handling multiple implementations for any single method.

For this reason, model contracts do not only comprise the interfaces that define the visibility scope of every included class, but also the visible implementations of the corresponding methods. Then, stubs are generated considering also this information so that dataClay can select which implementation to be executed when processing a method execution request.

Datasets

In order to facilitate the organization of the SCOs and to easily define how they are shared, dataClay offers the concept of **dataset** that enables data owners to enclose a set of SCOs.

Once the data owners register their datasets (a process analogous to registering a namespace for classes), they can provide **data contracts** to consumers granting them access to corresponding datasets. In short, a data contract offers a specific dataset so that the beneficiary can access the SCOs associated with such a dataset.

Besides, the data contract is limited with an expiration date and also defines whether the contracting party has the privilege to create new SCOs on the dataset or not. Therefore, the data owners keep control on the datasets they share, how, with whom and for how long.

It is worth noting that data contracts comprise a new use-case and are different than model contracts described previously. The former, grant access to the SCOs of the offered datasets; the latter, establish the visibility scope from a specific set of classes. In other words, data contracts enable *data sharing* (that is, SCOs), while model contracts enable *data model sharing* (that is, classes).

Remote execution

At this point it is worth to introduce some components of our system: the **Logic Module** (LM), and the **Data Service** (DS). Logic Module keeps track of all the management information such as namespaces, classes, interfaces, contracts, etc. On the other hand, Data Service

is in charge of processing method execution requests by managing the **persistence layer** where SCOs are actually stored. To this end, once a class is registered via the Logic Module it is then deployed to Data Service which also prepares the data infrastructure to store future SCOs that instantiate the class (more details in the following chapters).

Once a SCO is stored in the data infrastructure the stub behaves as the interface to access it remotely. To this end, dataClay's approach comprises a TCP communication binary-protocol and its own serialization mechanism both to create new SCOs and to pass the necessary arguments when executing a method on specific SCO.

The workflow for the execution of a method remotely, as it is show in the Figure 3.3, is: When a client application invokes a method of a certain SCO via the corresponding stub, the arguments (if any) are transparently serialized and transferred in the request, the request is then analyzed by Logic Module that checks if it comes from a valid contract that grants access to that class, and finally the Data Service processes the request by: retrieving the SCO from the data infrastructure, executing the requested method with the given parameters, and returning the result (if any) by using the same serialization mechanism and communication protocol as for the requests.

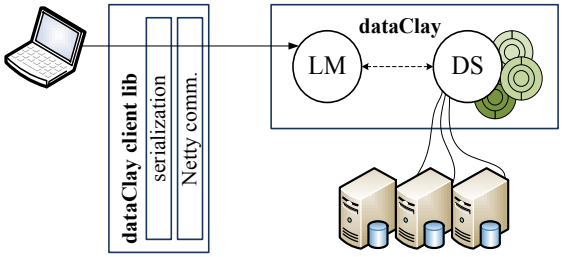


FIGURE 3.3: Remote execution in dataClay.

Communication with the Persistence layer

Even though dataClay offers the very elegant feature of working with SCOs without concerning where they reside (either on memory or a non-volatile environment), dataClay has to deal with this issue. It is dataClay's exclusive task to store and retrieve data (that is, objects) from the data infrastructure they rely on.

dataClay is ready to used several data infrastructures like Postgres, Cassandra, Neo4j and others as backends. Since the nature of the data store varies, the interaction between dataClay and any of these infrastructures varies too. For example, representation of data on relational databases is very different than on graph databases. For this reason, dataClay has a handler for each of the data infrastructure that it supports.

The objective for each handler is to map objects from the non-persistent to the persistent environment, and vice versa. More specifically, every handler has to prepare the data infrastructure for future storing of objects, which solely depends on the nature of the data

infrastructure. Afterwards, objects are stored, retrieved, modified, deleted to/from the data infrastructure according to the representation the handler uses on the persistence layer. More precisely, a handler receives a serialized object, manipulates this information (that is, bytes) and store the data on the persistence layer, according to the representation that it has chosen. The opposite happens when we want to retrieve an object from the persistence layer.

Objective of this master thesis is the development of the appropriate handler that enables the use of Kinetic technology into dataClay. The following chapters describe in detail several aspects for the development of this handler. Before moving to them, though, it is worth to gain some insight into Kinetic technology regardless dataClay. This is done in the rest of this chapter.

3.2 Seagate Kinetic Open Storage platform

As it was mentioned in the first chapter, Kinetic Open Storage is a drive architecture in which the drive is a key/value server with Ethernet connectivity. The purpose of this section is to provide a detailed description of Kinetic Open Storage platform. First, the motivation for developing this Ethernet key/value storage device is presented. Then, some Kinetic features that are relevant to this thesis are introduced. Last, the software and the hardware resources are exposed. Before proceeding further, it should be mentioned that the biggest part of this section derives from Kinetic website [4].

3.2.1 Kinetic architecture

The Seagate Kinetic Open Storage platform represents an opportunity to substantially address the inefficiencies of traditional datacenters whose legacy architectures are not well-adapted to highly distributed and capacity-optimized workloads of exploding unstructured data and applications.

Current datacenters are characterized by multiple layers of software and hardware stacked together in order to enable a data path between two poorly compatible systems: An object-oriented application layer and a hardware layer (spanning HDDs, SSDs, and tape) based on block-storage. The transit path from application to storage requires multiple layers of manipulation from databases, down through POSIX interfaces, file systems, volume managers and drivers. Information passes over Ethernet, through Fiber Channel, into RAID controllers, SAS expanders and SATA host bus adapters. A stack might look something like in Figure 3.4.

Beyond the obvious inefficiency of having to move through multiple layers, this model relies on a dated assumption about the operation of local storage devices: in the 1970's storage was organized close to, and based on, the physical attributes of a device. This has all changed, but the software stack has not evolved.

The majority of today's mass scale object applications do not need either file semantics (e.g. change the middle of a file, append to the end of a file, refer to a file by a name in a tree of names) or a file system to determine and maintain the best strategy for space management on

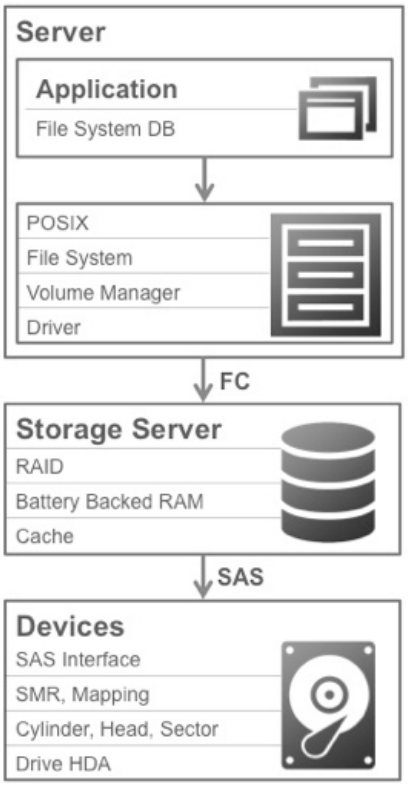


FIGURE 3.4: Traditional storage stack in a datacenter.

a device. Modern applications need only object semantics (e.g. write the whole thing, read the whole thing, delete the whole thing, refer to it by a handle chosen by the client and cluster manager), and should not need to worry about where data resides on a given device.

In order to manage this complexity, an entire ecosystem of storage server technology providers (both hardware and software) has risen up purely to abstract it from both the device and the application layers. Not only is this inefficient, it also introduces additional barriers between the two realms that can impede surfacing of storage features and functionality.

What if we could start over and re-structure the stack from the bottom up? What would it look like if object-oriented applications could speak directly to, and in the language of the storage device?

It would look like the Kinetic Open Storage platform. Kinetic is:

- A new class of key/value Ethernet drives + an open API and series of libraries.
- Designed to provide the simplest semantic abstraction and enable the broadest set of applications through an easy-to-use, minimalist API.
- An efficient platform to maximize innovation and value both within and above the storage device.

Together, these pieces enable applications to target storage devices directly and take best advantage of storage features. Drives talk in keys and values, as opposed to blocks. They do

'get', 'put', and 'delete' operations. They allow applications to distribute objects and manage clusters, while letting the drive efficiently manage functionality such as:

- Managing key (object) ordering
- Quality of service
- Policy-based "drive-to-drive" data migration
- Handling of partial device failures and other management
- Data at rest security

In contrast to the traditional stack described above, the Kinetic storage stack might look like this in Figure 3.5:

FIGURE 3.5: A storage stack using Kinetic technogy.

The new model has a number of significant implications:

- The superfluous layers of legacy software and hardware are removed.
- Need for the traditional storage server tier is obviated.
- Storage can truly be disaggregated from compute.
- Racks can be more dense.

- Fans are minimized.
- Data traffic leverages the existing datacenter transit fabric, Ethernet.
- Datacenter operational management is simplified, and both cost- and risk- reduced.

Scale-out is simplified, cost-effective, and unconstrained by legacy architectures and infrastructure. Information is now just an IP address away.

3.2.2 Kinetic Open Storage Value proposition

The Kinetic Open Storage platform is architected to enable simple, flexible storage performance and scaling. It delivers optimal total cost of ownership (TCO) for datacenter storage providing savings both in capital outlays and operational expenses

Performance

Kinetic drives are native key/value stores. This shifts the burden of maintaining the space mapping of a device from a file system to the drive itself. Applications need only put and get objects; they no longer need to guess at LBA layout or prescribe data location. This shift largely eliminates a very significant amount of drive I/O that moves no data but rather represents metadata- and file system-related overhead.

There is also incremental benefit here for scaling: as both device manufacturers and cloud datacenter operators ramp device capacity as aggressively as possible, the increased I/O efficiency - and resulting net I/O utilization - enables more balanced scaling of I/O and capacity, in addition to absolute performance on a given device and across a Kinetic cluster.

Scale

The Kinetic platform is uniquely optimized for explosive-growth, scale-out datacenters. The Kinetic architecture with its disaggregation of storage from compute enables cloud datacenter operators to simply add storage as need for capacity grows. Additionally, the combined impact of Ethernet connectivity and the key/value API command structure enables incremental capacity to be scaled in a highly distributed manner with the replication of data directed from drive to drive, with minimal incremental system cost.

Simplicity, Ease of Use/Adoption

Kinetic drives are provided with a comprehensive user space library that allows applications to access the device directly. This library provides the complete interface to access the data and to manage the drive. It bypasses the normal operating system storage stack and lets the application to talk directly to the drive as if it were talking to another service in the datacenter.

This process utilizes a typical application remote procedure call (RPC). This Kinetic platform currently provides libraries for Java, C++, C, Python, and Erlang, and other languages will be provided over time.

The Kinetic API allows applications to interact with the drive as if it were a typical key/value service on the network; it allows applications to put data in the form of keys and values to the drive and to get this data back by specifying just the key. As one would expect, keys and their values can be deleted. Additionally, the keys are ordered (lexicographically) so that searching of the keys within ranges and finding the next and previous keys are possible. The schematic in Figure 3.6 shows the basic architecture.

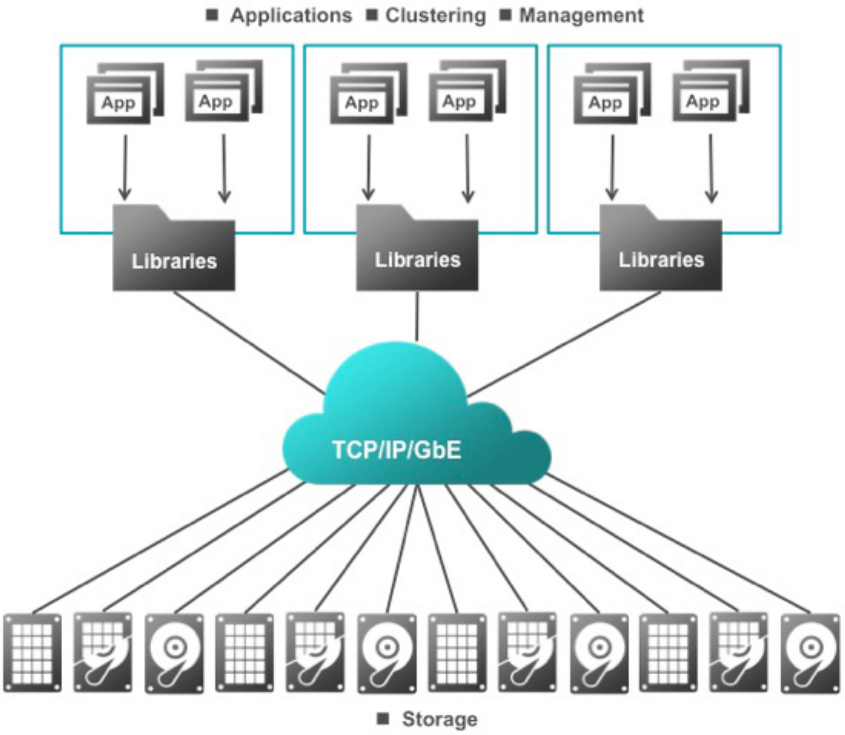


FIGURE 3.6: Architecture of basic application using Kinetic technology.

There are also extensive drive management commands that allow the drive to report its health and to manage who is allowed to communicate with the drive.

The Kinetic platform allows implementation of new datacenter architectures. This is due to the fact that Kinetic drives interface directly with the applications, thereby eliminating an entire tier of hardware. This technological advantage allows much denser storage racks, which impacts total cost of ownership in a number of different areas. Elaboration of these advantages is out of scope for this master thesis. Interested readers are prompted to Kinetic website for further information.

3.2.3 Kinetic features

Across this section, the most important features of Kinetic technology are described. First is described the notion of object storage. Then, elaboration on the key schema follows. Last, an inner Kinetic mechanism for concurrency issues is presented.

Simple object storage

Object based storage organizes data into flexible-sized data containers, with the approach of addressing and manipulating discrete units of storage called objects. Objects are not organized in hierarchy, such that one object cannot be placed inside another. Since every object is at the same level, this is considered a flat address space known as a storage pool. The key semantics for object storage are PUT, GET, and DELETE.

Object storage differs from legacy disk storage, where legacy disk storage used block-oriented interfaces that reads and writes fixed sizes of blocks of data. The object contains uninterpreted sequence of bytes (data) and sets of attributes to describe the object (metadata). The metadata are used to assign unique identifiers that allows a server or end user to retrieve the object without needing to know the location of the data, which is extremely useful for automating and streamlining data storage for cloud computing. The key functions of object storage are:

- Create objects
- Delete objects
- Write bytes to and from individual objects
- Read bytes to and from individual objects
- Set attributes on objects
- Get attributes on objects

The advantages of using Object Storage are:

- **Data Mobility** - The ability to reference objects by IDs rather than file names provides more freedom for migration of data, and eliminating the constraints of underlying hardware.
- **Scalability of Namespace** - The namespace does not have any size limitations and completely independent of the file and operating system.
- **Performance Scalability** - The ability to read and write directly to the objects simultaneously with no limitations.
- **Simplified Integration and Development** - The enhanced feature that provides easier coupling of applications and storage.

- **Storage Efficiency** - Objects only uses the space that they need, without having to pre-allocate storage for the storage container

Kinetic Drives implement key/value object storage for the advantages stated above. The basic semantics used for simple object storage are get, put, and delete.

In order to write the object onto the drive, the put operation is performed with the client requesting to write a created object using the Kinetic API by sending the object's keys and values through the network to the Kinetic drive, as shown in Figure 3.7.

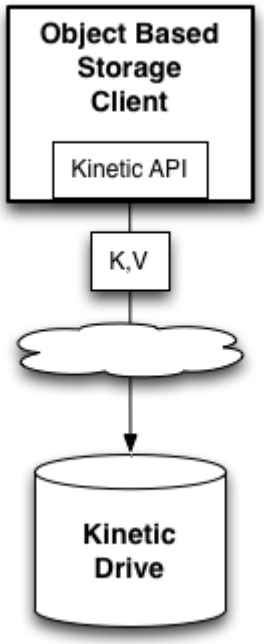


FIGURE 3.7: Put Operation using Kinetic API.

When requesting to read a desired object, the client sends the object's keys to the Kinetic drive through the network. Once the Kinetic drive receives the keys, the keys and values will be returned to the client through the network, as shown in Figure 3.8.

Similarly, the delete function is performed with the client sending the objects' keys to the Kinetic drive, in which the drive removes the keys and the corresponding values from the Kinetic drive.

Key schemas

In key-value object storage systems including Kinetic Open Storage, a key is a unique identifier for the object. Key-value object stores typically support a large key size such that not all possible values of keys can be stored. For instance, first generation Kinetic drives support keys of up to 4K bytes. This means that there are over 10^9864 possible values of keys. (In comparison, the Logical Block Address (LBA) of the largest block storage disk drives is about

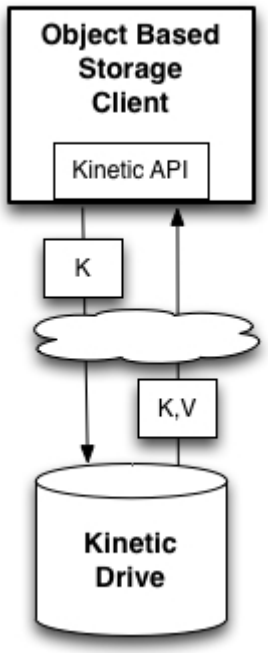


FIGURE 3.8: Get Operation using Kinetic API.

34 bits or 10^10 possible addresses, and the number of atoms in the observable universe is estimated to be about 10^80.) The LBA space of a block device is dense; all addresses from zero to the maximum LBA are used. The key space in a object store is typically sparse, but may be dense in a small part of the possible key range. The key is used to specify the key-value object being accessed. Put operations must specify the key for the key-value object to be added to the object store. Get operations must specify the key of the key-value object for which the value is to be retrieved from the object store. Delete operations must specify the key of the key-value object to be removed from the object store. If the key already exists in the object store during put operations, then the existing key-value object is replaced.

The key for a key-value object is totally at the discretion of the client or cluster manager. Any key that is within the limits of the drive is allowed. The key is an opaque value to a Kinetic device, with only key ordering as a quality that is understood by the device. The set of key-value objects in a Kinetic device are ordered based on the value of the key.

While keys are opaque to the drive, various considerations are likely to drive the formation of keys and the key schema.The major considerations include the following:

- Cluster Load Balancing
- Key Collisions and Key Space Segregation
- Key Ordering
- Semantic Meaning to Clients or Cluster Managers

Among the four considerations, only the third is explained further, since the rest are out of scope for this master thesis. Interested readers are kindly prompted to the Kinetic website for further information. As it is presented in the fourth chapter, key ordering is important for our case too, and it has been taken into consideration.

Key Ordering

Various use cases include access to a set of key-value objects. A key schema that provides for the most common sets to be sequential in key space can optimize performance as well as simplifying the implementations.

Throughput can be optimized for use cases where a set of keys will be accessed together if the set of keys are in sequential order. The following can benefit from key schemas that make keys in the set of key-value objects sequential.

- Large objects sharded into multiple smaller key-value objects stored on the same device
- Objects that are appended by an application by adding key-value objects to the same device
- Columns for a multi-dimensional database

If the key schema includes a sub-field that enumerates shards, addenda or columns and this sub-field is in the lowest order part of the key then access to the multiple key-value objects that make up the larger object may be optimized. For a database, a key schema that has a column identifier in the less significant part of the key would optimize accesses to a given column for all rows.

Searches can be optimized by minimizing the number of get key list calls needed for use cases where a list of a set of keys are needed. The following can benefit from key schemas that make keys in the set of key-value objects sequential.

- Object collections
- Versioned key-value objects

If the key schema includes a sub-field that enumerates the collection to which a key-value object belongs and this sub-field is in the lower part of the key then get key list methods can be issued to discover all of the keys belonging to the collection. Similarly, if the key schema includes a sub-field that enumerates the version of an object and this sub-field is in the lower part of the key then get key list methods can be issued to discover all of the versions for an object, and in particular can directly discover the newest or oldest version of an object. This can be further leveraged for snapshots.

Multiple clients with shared key/value objects

Kinetic also offers an inner mechanism for dealing with concurrency issues. Its key/value entries are accompanied by version identifiers. Moreover, every put operation can specify the existing version id as well as the resulting version id. This provides an elegant solution for shared data with safe updates. The mechanism is explained further with an example that follows.

If an object is shared by multiple clients, then occasionally two clients will attempt to update the same object at about the same time (see Table 3.1).

Client A	**Client B**
	get(key1)
get(key1)	modify object
modify object	put(key1)
put(key1)	

TABLE 3.1: Object modification by multiple users

In the case of Table 3.1, the data that Client B just wrote will be silently obliterated by Client A.

To help protect against this kind of accidental data destruction, each object has a 'version' identifier associated with it. The version id is returned to the host with the get command, and is sent to the device with the put command.

When a put command is sent to the device, the client can include version id that it previously read and specify that the put shall succeed only if the current version id matches. Prior to accepting the new object data, the device checks the current version id for the object, and if it does not match, the device rejects the new put command.

In the example of Table 3.2, the first put operation that Client A makes is rejected, because the version of the already-stored key/value entry that he wants to override does not match the one that he expects. This means that someone else has modified the key/value entry. This causes Kinetic to raise a VersionMismatchException exception. The programmer is called to handle this exception. In the example above, the modification for Client A is repeated on the recently modified by Client B object.

The modifications that the two clients need to do are different. The clients are doing two separate transformations. But if the transformations are commutative then they do not need to be strictly ordered.

Such operations scale, and the simple yet powerful Kinetic version mechanism directly supports high performance shared data.

Client A	**Client B**	**Device**
	get(key1)	returns version 1
get(key1)	modify version 1, resulting in a desired version 2	return version 1 for client A
modify version 1, resulting in a desired version 2		
	put(key1, old version = 1, new version = 2)	put is accepted
put(key1, old version = 1, new version = 2)		put is rejected
get(key1)		return version 2 for client A
modify version 2, resulting in version 3		
put(key1, old version = 2, new version = 3)		put is accepted
	get(key1)	return version 3

TABLE 3.2: Versioned object modification by multiple users

3.2.4 Software resources

API Overview

The Kinetic Library includes two categories: Admin and Client. Here, only client library is presented, since it is more meaningful for the purpose of this master thesis.

The Client API provides the Kinetic client application interface to communicate directly with the Kinetic service, in two types of operations.

- **Synchronous** – Operations guaranteed to be successfully performed on server if call returns with no Exceptions
- **Asynchronous** – Operation guaranteed to be successfully performed on server if CallbackHandler (passes instance of the implementation) receives successful CallbackResults (obtains operation requests)

Client API provides more than the basic key/value object-based storage methods, with the key methods shown below:

- **put** – Put the specified entry to the persistent store
- **get** – Get the entry associated with the specified key
- **delete** – Delete the entry that is associated with the key specified in the entry.

- **getNext** – Gets the entry associated with a key that is after the specified key in the sequence
- **getPrevious** – Gets the entry associated with a key that is before the specified key in the sequence
- **getKeyRange** – Get a list of keys in the sequence based on the specified key range
- **getMetadata** – Get entry metadata for the specified key.

3.2.5 Hardware resources

The first generation Kinetic drive is a 4TB, 5900 rpm, 3.5" hard disk drive (HDD). Compared to its conventional sister drive, the Kinetic drive implements the Kinetic API that enables key-value object storage. The Kinetic drive replaces the Serial Advance Technology Attachment (SATA) or Serial Attached SCSI (SAS) interface connections with two 1–Gbps SGMII Ethernet ports, which enables direct network attached connectivity. The Ethernet interface allows communication between drives and direct communication to the datacenter, eliminating the need for Storage Servers for datacenter storage racks.

Throughout the conduction of this master thesis, the implementation code was tested using a Kinetic prototype device which Seagate had kindly offered to Storage Systems Research Group at BSC. This device is the 4-Bay Development Chassis, which is presented in the Figure 3.9.

FIGURE 3.9: The 4-Bay Development Chassis.

The purpose of the 4–bay development chassis is to provide a low cost, easy–to–use desktop device to use as a test and software application development device. The chassis consists of four drive bays, each with two (2) SGMII Ethernet ports, and a "backplane" PCB into which the SGMII drives plug. The backplane provides all SGMII signal routing through an Ethernet switch or switches covering all eight (8) SGMII drive ports. The backplane also provides routing for all electrical power required by the drives and supplied by the included power supply. The system is managed by manual on/off power control switches for each disc drive in the enclosure and externally managed through Ethernet and I^2C.

Simulator

Last but not least, Kinetic offers a simulator API that can be used instead of the 4-bay development chassis. The Simulator API provides a simulator boot-strap class used for applications to start new instances of the Simulator to act as a drive.

Chapter 4

Persistence Layer: Serialization mechanism

As it was mentioned in the previous chapters, one of the core advantages of dataClay is the SCOs' abstraction from the layer they rely on. Either on memory or not, the programmer deals with SCOs in the same way, like being loaded on memory. In that way, programmers do not need to cope with storing, reading, updating, deleting data. They just focus on the logic of their application.

On the other hand, data are not loaded on memory for ever, obviously. Data (that is, SCOs) are stored in non-volatile environments. dataClay supports internally all the needed mechanisms that enable this functionality. More specifically, dataClay has its own serialization mechanism and also has handlers that support data management on several platforms, like PostgreSQL Relational DataBase Management System, Apache Cassandra and others. The objective of this master thesis is the development of another handler, which connects dataClay with Seagate Kinetic Open Storage platform. However, profound understanding of the serialization mechanism is needed before moving into the implementation of the new handler.

Across this and the following chapter, dataClay's persistence layer is examined from two perspectives. This chapter examines how things are organized before reaching the data infrastructure. In other words, how dataClay's custom serialization mechanism works. On the other side, the following chapter examines how Kinetic handler deals with the output that the previous layer (that is, the serialization mechanism) produces.

The first section of this chapter describes one essential procedure for dataClay, the bytecode analysis, which is quite important for persistence layer and beyond. The second section presents the motivation for implementing a custom serialization mechanism for dataClay and how it is achieved. Next, how dataClay represents user classes and their fields in the Data Service is presented. What follows is the handling of null references. Then, treatment of already stored or already serialized objects follows. After all these introductory parts, the content of a serialization message is presented. Before reaching to the end of the chapter, special cases of the serialization mechanism are presented. These cases include array, collections and maps.

4.1 Bytecode analysis

In the previous chapter, it was slightly mentioned that dataClay supports its own serialization mechanism. Nevertheless, it has not been revealed yet who or what mechanism implements the serialization and deserialization methods and at which moment this process is accomplished. The answer is that dataClay itself is in charge of this process and this is achieved during the *bytecode analysis*, which is detailed below.

During class registration, the dataClay client library tool analyses the bytecode of the classes registered in the system. The fact is that this process is multipurpose. It is important to a) prepare the persistence layer where SCOs are actually stored, b) generate the bytecode required for the serialization mechanism, c) check dependencies among classes, and d) identify the methods that modify the state of the SCO.

In order to prepare the persistence layer to store the SCOs, the handlers of data storages need the information about the morphology of the class to prepare the required structures in the underlying storage. For that purpose, the bytecode of the class is analyzed to extract its fields and types.

As we know, stubs are the result of a user class which is "filtered" with a model contract. The analysis of the class structure is also useful for the serialization mechanism, thus allowing generating the bytecode of the *makePersistent* method that is in charge of serializing the stub instance to generate the persistent SCO. In the same way, the bytecode analysis is useful for the generation of the class methods that require argument passing and/or return some value, since knowing the types of the arguments is necessary to generate the corresponding bytecode in order to serialize efficiently the parameters in the corresponding execution requests.

Regarding the class dependency analysis, it is useful firstly to enable the client library to transparently register the required classes needed for the main one; and secondly, to notice the requirements of a class when it is offered to a 3rd party via a contract.

Last but not least, analyzing the methods enables dataClay to know the arguments and return types, which is necessary for the serialization mechanism; and to keep track of those methods that modify the state of the SCO, (i.e. those setting new values to any of its fields), which is useful to know when to propagate updates and making them persistent transparently from the point of view of the application.

4.2 Motivation for implementing custom serialization mechanism

It has been mentioned several times that dataClay has its custom serialization mechanism. There are several reasons that led to this decision:

- Java default serialization requires having the same class at both client and server sides. Which is not true in our case, because of the model contracts that we apply.

- Java standard serialization is also slow, because of reflection. Even if we implement the Externalizable interface, which avoids reflection, the performance is quite poor, too.
- There are some representations, like cycle references or the references to other persistent objects (and they do not need serialization) that are difficult to be represented in Java RMI.
- Also, having our own serialization mechanism allows us to avoid deserialization and serialization in intermediate layers like Logic Module.
- Java serialization also sends class information together with the instance (types and so on), which causes redundancy of information.

Since we want to avoid both reflection and the *externalizable* interface, dataClay implements its own serialization method in a different way. Given that dataClay generates the bytecode of stub classes, the serialization code can be hardcoded in the *makePersistent* method and for the parameters and return values of methods, thus avoiding reflection every time an object is serialized. Soon, the structure of a serialized object is presented.

4.3 Representation of classes in the Data Service

Throughout this section, the representation of a user class in the Data Service is represented.

As it has been said earlier, when a user registers a class, the Data Service (DS) is in charge of preparing the data infrastructure for future storing of objects that instantiate that class. For this purpose, a special class, named *DBClass*, exists. The objective of this class is to represent the user classes. It contains an array which has the same length as the number of the user class fields. Every element of this array represents one of the user class, in the same order as in the class.

In this paragraph, the workflow which starts with the registration of a class in dataClay until the preparation of the data infrastructure is presented. As it has been said, when a user registers a class, the bytecode analysis of the class follows and then its registration is accomplished. During this process, the Logic Module requests from another smaller component, the *Class Manager*, to instantiate a DBClass object for the user class representation. The Class Manager is capable of creating the appropriate instance of DBClass for the user class, since it stores all the information of the class resulting from the bytecode analysis that was held earlier. When the DBClass object is prepared, it is passed from the Logic Module to the Data Service. Then, the Data Service has to prepare the persistence layer (that is, the data infrastructure) for future storing of instances of that user class. The preparation of the data infrastructure depends exclusively on the data infrastructure nature. For example, relational databases require different handling than NoSQL databases. Thus, Kinetic infrastructure also needs its special treatment. For this reason, Kinetic handling will be explained in the next chapter.

Last but not least, it is worth to mention that DBClass objects are also necessary for the handlers that are in charge of storing/updating/reading/deleting data objects. Handlers deal

only with serialized messages (that is, bytes). So, these bytes do not bear any semantics of the data and no information for the morphology of the class they represent. However, having this information is definitely needed. For example, we need to know the amount of bytes we have to read for a field: An integer has 4 bytes, while a double 8 bytes. Thanks to DBClass, bytes from serialized objects can get meaning.

4.4 Representation of class fields in the Data Service

In a similar way to user classes, fields of user classes are represented in the Data Service by instances of a special dataClay class, called *DBField*. Without diving too deeply into the technical details, DBField models the user class fields, namely their name and their type. The type of a field can be either any primitive type or reference to other object. This is achieved thanks to a boolean called *isNullable*. If it is true, the user field is a reference. Otherwise, it is a user field of primitive type. This boolean is set during the registration of a class. Afterwards, *isNullable* is used by the handlers, since dealing with references is very different than dealing with primitives.

4.5 Meaning of Not-Nulls-Bitmap

As it is obvious, objects can reference to other objects through their reference fields. However, these reference fields may not point to other objects. In such a case, their value is *null* and there is no need to serialize anything for null references. The serialization mechanism takes this fact into consideration and appends to every serialized object (that contains references) a variable number of bytes that bear this information.

More precisely, a bitmap is appended in every serialized object (which has references) and it serves the purpose of knowing which references are null or not. This bitmap is called *notNullBitmap*. It has the same number of bits as the number of reference fields of the serialized object, one per reference. The order of the bits is the same as the order of the references that they have been defined in the class. If a bit is set (that is, true), it means that the corresponding reference is not null and bytes for the referenced object exist in the serialized object. On the opposite, if one bit is not set (that is, false), it means that the serialized object has no bytes dedicated for the null reference.

Last but not least, it is worth to mention that it depends on the data infrastructure how null references are mapped on the persistence layer. For example, relational databases can map a null reference into a null value in the corresponding table, row, and column. In the case of graph databases though, if we suppose that objects are mapped to nodes, null references can be mapped by not putting an edge in the graph. So, null treatment depends on the data infrastructure. The Kinetic case is explained in the next chapter.

4.6 Handling of already stored objects

When an object is serialized, this object may contain references to other already stored objects (in other words, persistent). In this case, it does not make sense neither storing again (that is, overwritting) the already stored objects, nor serializing them. Instead of re-serializing the persistent objects, dataClay writes their object ID into the serialization message of the unstored object. Afterwards, it continues with the serialization of the remaining fields.

However, what is the criterion that makes an object persistent? In other words, how does dataClay recognize persistent objects? The rule is: If an object has a dataClay object ID, it is persistent. Otherwise it is not. An object gets tied with its object ID for its whole life cycle, only when it is stored. Thus, checking the persistency of an object is actually checking its object ID.

It is very important to make clear that this object ID is not a Java identifier. It is an internal dataClay unique identifier. One reason is that Java identifiers (like the hashcode of an object) identify objects only as long as they reside into the Java heap. On the contrary, dataClay needs to identify its objects regardless they reside in the persistent layer or in memory. Secondly, dataClay objects can be shared between different clients and servers. Thus, we need unique identifiers for the whole system. That's why dataClay uses its own IDs, which are actually a field of every object. More precisely, it is a field of DataClayObject. Since every stub class extends DataClayObject, every object has this field.

In the end, how is this bonding between an object ID and the freshly-persistent object achieved? When *makePersistent* operation finishes for an object, its dataClay object ID is returned. Data Service receives this object ID and sets it to the corresponding object, which still resides in the memory. Thereafter, any attempt for reserialization of the object will be prevented, since it has already a dataClay object ID.

4.7 Handling of already serialized objects

During the serialization of an object, if dataClay finds a reference to an object that has been already serialized, this object will not be serialized again. There are two reasons for doing this: 1) Redundancy of both processing and information is avoided, and 2) Any possible cycle-reference is avoided: Imagine dataClay heading up in processing a cycle-reference: This would lead to an infinite serialization.

In order to avoid such cases, dataClay tags every object with an integer. If an object has been already serialized before, dataClay just appends the tag that corresponds to that object. Then it continues with the serialization of the next field.

How does dataClay recognize the already serialized objects? It uses a map, whose keys are the object hashcode and values are the tags of the already serialized objects. If a key-value can be found for an object in the map, it means that it has been already serialized. In such a case, only its tag is appended. On the other side, when an object is encountered for the first time, it is tagged with the next available tag and its key-value pair is added to the map.

4.8 Serialization message

Several smaller parts of the serialization mechanism have been introduced so far. Their synthesization composes the serialization mechanism. There are some key facts that make the understanding of the serialization mechanism of dataClay easier:

- It is a recursive procedure: While processing with the serialization of an object, if some of its fields refer to other non persistent objects, then they are serialized, and later the serialization remainder of the initial object resumes.
- When the serialization of an object is finished, its superclass is serialized too. Since every class extends DataClayObject, every class serializes at least another superclass. Moreover, serialization of DataClayObject comprises also the halt criterion of the serialization.

The pseudocode of Algorithm 1 describes the algorithm of the serialization mechanism:

Algorithm 1 The algorithm of the serialization mechanism

```
1: procedure SerializeObject()
2:     append tag
3:     if tag already used then
4:         return
5:     end if
6:     append classID of the object
7:     append isPersistent boolean
8:     if isPersistent = true then
9:         append objectID of the object
10:        return
11:    end if
12:    append notNullBitmap
13:    for all field do
14:        if primitivefield then
15:            append its value
16:        else
17:            if field not null then
18:                field.SerializeObject()
19:            end if
20:        end if
21:    end for
22:    if superclass = DataClayObject then
23:        SerializeDataClayObject()
24:        return
25:    else
26:        goto step 6
27:    end if
28: end procedure
```

4.8.1 Further explanation of the algorithm

In order to facilitate the understanding of the algorithm, some steps are explained further.

- The step 2 was described in the section "Handling of the already serialized objects". As it is obvious, step 3 is always false when SerializeObject() is called for first time. In other words, when an object is encountered for first time, its tag is also used for first time.
- The steps 7–9 were described in the section "Handling of the already stored objects".
- The objective of notNullBitmap was described above in the corresponding section. However, it is worth to mention how step 12 really works. As it is obvious, it is quite hard to know whether the fields of the objects are null or not, before processing them. In other words, the job of step 12 seems impossible before step 13. What is actually done is a bit more complex: At step 12, only the space for the notNullsBitmap is allocated. Instead, the notNullsBitmap is formed during the step 13–21, when every field is processed. Then, between step 21 and step 22, the final notNullBitmap is written in the space that was allocated at step 12.
- The step 23 has not been detailed thoroughly enough, on purpose. During that step, the serialization mechanism appends some metadata of fixed size (in bytes) for the DataClayObject. Since it is a trivial procedure, we can neglect its technical details. The only worthy thing to mention is that after this step the termination step of SerializeObject() comes.
- As it has been mentioned before, the serialization mechanism is a recursive procedure. So, it is quite possible having information for more than one objects after SerializeObject() finishes. Actually, this can happen at two different steps: Either the object itself has references to other objects (step 18), or some of its superclass fields are references to other objects (step 26). This fact is very important and the reader should keep it in mind for the next chapter.

4.9 Wrappers

Even though the algorithm above covers several use cases, it does not cover other common ones. For example, it can handle any user object with either primitive or reference fields, but it misses functionality for other common cases like arrays, collections and maps. The reason is that these richer structures are not defined by some user. Instead, there are part of the Java language. Thus, there is neither registration of arrays/collections/maps, nor stub classes for them. Instead, their Java regular classes are used. Since no stubs are produced for these classes, there is no serialization method (by dataClay) for these structures either. So, if we want to store an object which contains an array (for example), this array will be wrapped into a special class during the serialization of the outer object. Then the wrapped array will be serialized, and this will finally be stored in the data infrastructure.

As it has been said, during the registration of a class, its bytecode is analyzed and the serialization and deserialization methods are implemented. If a reference to any of these types is found, then these fields are wrapped and the serialization method of the wrapper is used instead, which is already implemented. It is further detailed below.

4.9.1 Array wrappers

When a reference to an array is found during the serialization of an object, then this reference is wrapped into the appropriate wrapper. Particularly, there are 8 wrappers for the 8 primitive types and one wrapper for arrays of any object. The latter case covers even multidimensional arrays or arrays of Java default wrappers (like Integer etc.). In the case of an array of primitive type (Algorithm 2), its serialization is quite simple.

Algorithm 2 The algorithm for the serialization of arrays with primitive types

```
1: procedure SerializePrimitiveArray()
2:     append tag
3:     if tag already used then
4:         return
5:     end if
6:     append classID of the object
7:     append isPersistent boolean
8:     if isPersistent = true then
9:         append objectID of the object
10:    end if
11:    append array length
12:    append serialized array
13:    SerializeDataClayObject()
14:    return
15: end procedure
```

There are a few things worth to mention:

- At step 6 the class ID of the corresponding wrapper is serialized. For example, if an array of shorts is serialized, the class ID of the wrapper for arrays of shorts is going to be appended. This information is very useful for handlers.
- At step 11 the length of the array is appended, because it is very useful for the deserialization. If this integer is not included, then deserialization cannot know how many bytes to read for the array.
- At step 13, only the DataClayObject is serialized, since it is the only class that the wrappers extend.
- As someone might have noticed, there is no notNullBitmap. The reason is that an array will not be wrapped and serialized if the reference to the array is null. Instead, the object that contains the reference to the array will mark as false the corresponding bit in its notNullBitmap. On the handler's side, since the bit will be false, the deserialization will continue to the next field.

In the case of an array with references to other objects, the serialization is bit more complex (Algorithm 3).

Algorithm 3 The algorithm for the serialization of references

```
1: procedure SerializeObjectsArray()
2:     append tag
3:     if tag already used then
4:         return
5:     end if
6:     append classID of the object
7:     append isPersistent boolean
8:     if isPersistent = true then
9:         append objectID of the object
10:    end if
11:    append classID of components
12:    append array dimension
13:    append array length
14:    if length > 0 then
15:        append size of notNullBitmap
16:        append notNullBitmap
17:        for all array elements do
18:            if element not null then
19:                element.SerializeObject()
20:            end if
21:        end for
22:    end if
23:    SerializeDataClayObject()
24:    return
25: end procedure
```

Regarding Algorithm 3, some comments follow:

- At step 11, the class ID of the elements is appended. This information is needed for the array instantiation during the deserialization. At that moment, the array must have some type.
- At step 14, if the length of the array is 0, there is no need to append any extra information except for the superclass. Furthermore, this metadata is very important because it denotes, more or less, the amount of bytes someone can expect in the remainder of the serialized array.
- At step 15, the size of the notNullBitmap is appended. This is done because there is no other way to compute this size during the serialization. If it were not supplied, deserialization cannot be aware of how many bytes to read for notNullsBitmap. Instead, in the case of SerializeObject() such an information is not stored since the size of the bitmap can be calculated thanks to the DBClass instance.
- At step 19, the serialization method of the referenced object is executed, which is known due to the bytecode analysis.

4.9.2 Collection wrapper

After looking the array wrappers and their details, collection wrapper does not surprise. Its serialization pseudocode can be found in Algorithm 4.

Algorithm 4 The algorithm for the serialization of collections

```
1: procedure SerializeCollection()
2:     append tag
3:     if tag already used then
4:         return
5:     end if
6:     append classID of the object
7:     append isPersistent boolean
8:     if isPersistent = true then
9:         append objectID of the object
10:    end if
11:    append name of the collection
12:    append size of collection
13:    if size > 0 then
14:        append size of notNullBitmap
15:        append notNullBitmap
16:        for all collection members do
17:            if member not null then
18:                member.SerializeObject()
19:            end if
20:        end for
21:    end if
22:    SerializeDataClayObject()
23:    return
24: end procedure
```

The only thing we should mention here is that at step 11, the name of the collection is written in a string (for example, "ArrayList"). Having this information is needed for the instantiation of the collection during the deserialization of the object.

4.9.3 Map wrapper

Map wrapper does, more or less, twice the process of a collection wrapper. Its serialization pseudocode can be found at Algorithm 5.

As someone can notice from the Algorithm 5, steps 12 through 21 are for the keys of the map and steps 23 through 31 are for the values.

Algorithm 5 The algorithm for the serialization of maps

```
1: procedure SerializeMap()
2:     append tag
3:     if tag already used then
4:         return
5:     end if
6:     append classID of the object
7:     append isPersistent boolean
8:     if isPersistent = true then
9:         append objectID of the object
10:    end if
11:    append name of the map
12:    append size of map
13:    if size > 0 then
14:        append size of notNullBitmap of keys
15:        append notNullBitmap of keys
16:        for all keys do
17:            if key not null then
18:                key.SerializeObject()
19:            end if
20:        end for
21:    end if
22:    append size of map
23:    if size > 0 then
24:        append size of notNullBitmap of values
25:        append notNullBitmap of values
26:        for all value do
27:            if value not null then
28:                value.SerializeObject()
29:            end if
30:        end for
31:    end if
32:    SerializeDataClayObject()
33:    return
34: end procedure
```

Chapter 5

Persistence Layer: Kinetic handler

Up to this point, very few things have been revealed for the implementation part of this master thesis. The objective of the previous chapters was to provide the reader with the required background to reach this chapter (and understanding all these concepts was also part of my master thesis, given their novelty and lack of documentation). The vast majority of this chapter describes the design choices made for accomplishing the integration of the Kinetic technology into dataClay. The most important criterion for making these choices was always the high performance. Another objective of this chapter is to expose the differences between semantically rich data infrastructures, like relational databases, and byte enabled ones. For this reason, Kinetic handler is compared with Postgres handler several times throughout this chapter.

5.1 Establishment of connection with Kinetic 4-bay development chassis

Throughout the implementation part of this master thesis, Kinetic 4-bay development chassis was used extensively. In this section establishment of connection to the 4-bay development chassis is described. The 4-bay developer kit architecture is simple and only needs a single switch to tie the Ethernet ports together. The Figure 5.1 exemplifies the architecture and gives one way of connecting a system together.

In this case, a cable is connected to the local area network (LAN) assuming that the LAN has an existing IPv4 DHCP server. The other port can be connected to a computer. Since the switch bridges all the 10 ports together (the 8 drive ports [4 drive x 2 ports each] and the 2 external ports) the computer is able to get the IP address from the LAN as it would normally do. The drives will then do DHCP for all 8 ethernet ports and begin to multicast their configuration using UDP to 239.1.2.3 port 8123.

For the purpose of this master thesis, setting up a DHCP server in the personal computer is more plausible than using the LAN. Next are presented the steps followed for the configuration of the DHCP server under the Linux operating system:

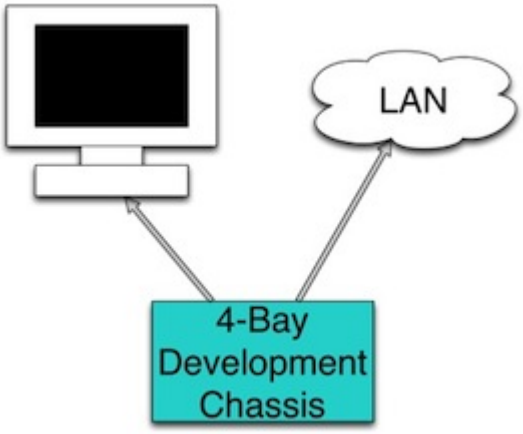


FIGURE 5.1: Establishment of connection with the Kinetic 4-bay development chassis.

1. Installation of the DHCP network service:
   sudo apt-get install isc-dhcp-server

2. Then, the configuration file /etc/default/isc-dhcp-server was edited in order to specify the interface that the DHCP should listen to. The interface eth0 was added.

3. Editing of the file /etc/dhcp/dhcpd.conf is needed, too. By doing so, the address range that the DHCP will multicast is specified. In our case, the range 10.5.5.10 to 10.5.5.20 was selected. The following lines were added in the configuration file:
   subnet 10.5.5.0 netmask 255.255.255.224 {
   range 10.5.5.10 10.5.5.20;
   option routers 10.5.5.1;
   }

4. Last, assigning a static IP to the interface that we use for dhcp (eth0 in our case) is needed. This was done by using the graphical network manager that Ubuntu comes with. An instance of this step is captured in the Figure 5.2
   Moreover, through the *Routes...* option (as in Figure 5.2), this connection was set to look only for resources on its network. By doing so, navigating to the Internet through this connection is prevented.

The 4 steps above do not need to be done more than once. However, the DHCP network service has to be started every time we want to connect to the 4-bay development chassis. This is achieved with the following command: sudo service isc-dhcp-server start

Seagate has developed several script tools in Python programming language. One of them, called discover.py, recognizes which Kinetic drives are connected and which IP address is assigned to them. One of these IP addresses can be used for the configuration of the client which is responsible for connecting the application (in our case, the dataClay platform) with the Kinetic drive.

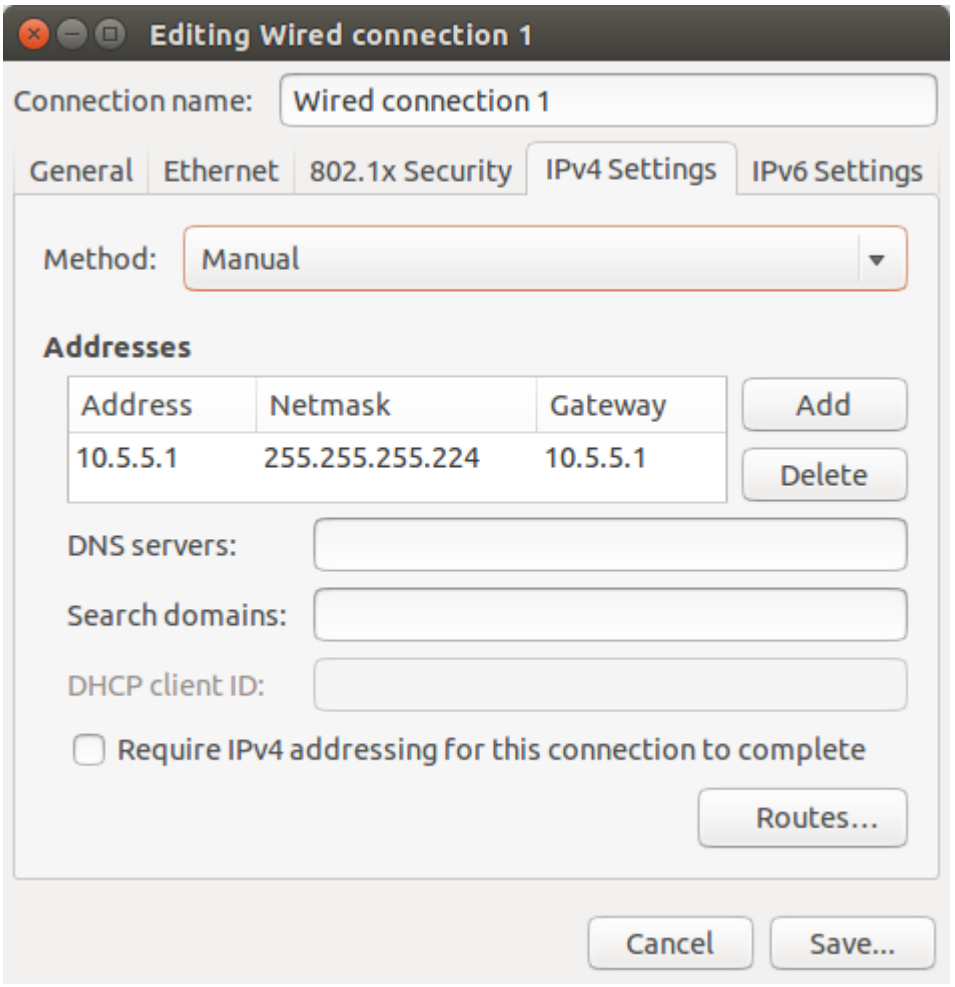


FIGURE 5.2: Assigning a static IP.

5.2 Kinetic key-value objects overview

This section briefs the technical details of the key-value objects that can be stored in a Kinetic drive. According to the Kinetic API, such an object is called *Entry*. In other words, there is a Java class called *Entry* which represents the key-value objects.

Every entry is identified by its unique key, which, in Java terms, is a byte array field in the class Entry, called key. Its maximum size is 4 KB. Similarly, every entry has another byte array field for storing the value of the key-value object, called value. Its maximum size is 1MB.

So, a key-value object in Kinetic drive is a couple of byte arrays. It is programmer's exclusive task to decide what to store in both key and value fields. To be more precise, we have seen in the second chapter several aspects that the programmer should consider in order to take advantage of the key ordering. Correspondingly, the value can be formed in a way so that it fulfills the needs of the application. Both key and value usage in dataClay case are detailed in the following sections, step by step.

5.3 Representation of classes on the Kinetic persistence layer

In the previous chapter, it was mentioned that every user class is modelled by a dataClay class, called *DBClass*. In addition, it was said that instances of that class are passed to Data Service in order to prepare the data infrastructure for future storing. Every handler decides how classes will be represented on the persistence layer because this depends solely on the nature of the data infrastructure. Here, we examine the Kinetic case and it is also compared with relational database backends.

When someone deals with OOP programming languages and relational databases, it is quite common every class to be mapped into a database table and every object to be mapped into a row in the corresponding table. dataClay does the same for its Postgres handler. In order to achieve this, it traverses through the representation class (that is, DBClass), creates the proper "CREATE TABLE" SQL command and then this command is executed by the RDBMS. Future instances of that user class will be rows of the newly created table.

On the contrary, Kinetic architecture has adopted the much simpler key-value objects abstraction. It can only write a pair of byte arrays, one for the key and one for the value. Nothing more. So, looking for a structured representation of data like database tables or file systems does not make much sense in Kinetic. It would only add an overhead to the performance. The most straightforward and probably efficient solution is to store every serialized object as a key-value object, where value is the bytes of the serialized object (the key schema is detailed in the following section). Hence, Kinetic handler does not need to do anything for preparing the data infrastructure for storing instances of a class in the future. Actually, every object is independent of its class inside the Kinetic drive.

On the other hand, tables in relational databases offer a very nice notion of grouping similar things: Every row in a table represents an object of the same class. For example, querying all the instances of a class is just a "SELECT * FROM" command. Kinetic handler takes this fact into consideration and makes its best. Since every entry is ordered by its key, grouping objects of the same class is an easy task for Kinetic too. If someone uses the same prefix for the key of same class objects, then each of them reside next to the other, because of the entries ordering. In our case, using the class ID as prefix of the key schema seems to fit perfectly. Soon, the exact key schema for storing objects will be explained.

Last but not least, in the previous chapter it was said that DBClass plays a double role: in addition for preparing the data infrastructure, it is also useful for providing meaning to the serialized objects. For this reason, it is important to have always this information handy. Indeed, in Postgres handler, once a class is installed (that is, creating the corresponding table in the database), its schema is also saved in a special table which contains the representation of every installed class. Kinetic handler saves the schema information as well. It creates a new key-value entry whose key is the class ID and its value is the serialized instance of the DBClass. Once the representation of a class is needed, its serialized schema is retrieved from the Kinetic drive, calling *get(key)* where key is the class ID, and, then, the value is deserialized, which leads to an instance of DBClass. Moreover, dataClay has a software defined cache which contains instances of DBClass. This prevents the continuous accessing of data infrastructure,

when the representation of a user class is needed. Only if a DBClass object is missing from cache, access to the persistence layer follows.

5.4 Storing objects on the Kinetic persistence layer

Probably the reader has already gained some insight into how objects are organized in the Kinetic device. However, there are still several details that need to be demystified. Here, most of the design choices made for the Kinetic handler are presented, as well as the motivation that led to them.

5.4.1 Overview

As it has been stated above, Kinetic handler stores every object of a dataClay user as a key-value entry on the Kinetic drive. This is a rule that must never be violated in both directions: Every object should be stored as a whole entity and never be segmented. On the other side, a key-value entry on Kinetic drive cannot contain information (that is, bytes) for more than one object. Probably, this is the most important rule in the Kinetic handler and every aspect around this design choice will be explained further. From now and on, this rule will be called "Single-object rule".

5.4.2 Pitfalls for breaking the single-object rule

Let's run through an example for understanding that if we do not pay attention, the single-object rule can be violated. Let's suppose that the makePersistent method has been called for an object. Let's call it objectA. Moreover, let's assume that objectA contains a reference to another non persistent object, objectB. This implies that both objects must be stored on the persistence layer (in our case, the Kinetic drive). According to the serialization mechanism (which was described in the previous chapter), objectB will be serialized "inside" objectA, since objectB is not persistent either.

If we store objectA at once, without setting apart objectB, it would cause violation of the single-object rule. Specifically, the resulting key-value entry on the Kinetic drive would contain information (bytes) for more than one dataClay object. Thus, storing dataClay objects in the Kinetic drive cannot be achieved with just a put operation.

5.4.3 Motivation for having single objects on the Kinetic drive

Before moving on the description of how the single-object rule is fulfilled, it is crucial to present the motivation for having such a rule. In our case, we will see that the "housekeeper" rule is really important: A little extra effort put into routine maintenance can pay off handsomely in the long run, by forestalling major calamities.

The first reason for setting apart the objects is set by the Kinetic device itself. As it has been mentioned, every key-value entry has limitations on the size of both key and value. Specifically, the key can be up to 4 KB and the value up to 1 MB. Imagine an object which contains a big collection to other (big) objects. Trying to store the initial object as a whole entity will possibly exceed device's capacity. Which, in turn, would cause an Exception.

In addition, keeping things organized makes sense for the performance of the Kinetic handler as well. Let's suppose we have already stored an object (call it objectA), which contains also another object (call it objectB). Moreover, let's assume that somehow we are aware of this Has-A relationship between objectA and objectB (which, by the way, is quite hard with direct put operations). If objectB is modified, dataClay itself will issue an update operation to the persistence layer for objectB. But since objectB is inside objectA (and the handler knows it), the handler is compelled with the extra work of finding which bytes are dedicated for objectB and not for objectA, and eventually doing the desired update operation.

From the previous paragraph and its example, it was slightly implied that we cannot easily know what we store when doing direct put operations. In other words, we are able to know that we store the outer object, but not the ones included within the outer. Having this information is crucial, as we have seen in the previous chapter: Once an object got persistent, it is tied with its (dataClay) object ID, which is the criterion for an object to be persistent. If this step is not done, it is quite possible of having multiple copies of the same object in data infrastructure. Let's see the possible inconsistency issues through an example: objectA (which contains objectB) is stored with a direct put operation. Thus, we don't know which objects got persistent other than objectA. Similarly, another object (call it objectC) which also contains a reference to objectB is stored with direct put operation. Since we do not know that objectB is already persistent (inside objectA entry), it will be stored again (within objectC bytes). Afterwards, if we update objectB in any of the two replicas, it will cause inconsistency to the other.

In conclusion, the importance of separating objects and storing them individually makes much sense. Otherwise, problems (inconsistency, performance, excess of data limits) will evoke in a domino fashion. Hence, the extra effort for tidying up the objects when they are stored for first time is more than worth.

5.4.4 Content of the value in a key-value entry

Across this section, the pattern for the value of a key-value entry is presented. It is quite useful to think of Kinetic drive as the Java heap. In the heap, every object contains data for its fields. Specifically, it contains information for every field of primitive type, as well as references to other objects. Similarly, key-value entries in the Kinetic drive behave like objects in the heap: If an object has fields of primitive type, these fields are serialized and stored within the key-value entry of the object they belong to. Moreover, key-value entries point to other key-value entries in the Kinetic drive and they do not contain other key-value entries, as objects do not contain other objects in the heap[1].

[1]Even though inner objects exists as term in Java, they are strongly dependent on outer ones. In our case, we are talking about independent objects.

However, serialized objects probably contain information for more than one object. This implies that we need to process this information, extract the nested objects and store them separately. This requires a very meticulous work since we have to process information in byte level mostly, and in bit level in special cases.

Thankfully, both serialization mechanism and Kinetic drive are of the same nature. They both understand bytes. Nothing else. Nothing more. Thus, what we need to store in Kinetic drive is driven more or less by the previous layer, the serialization mechanism. It does not make much sense to attempt for a completely different representation of the objects in Kinetic drive. It would only add overhead on performance, in both directions: While putting an object, we would have to translate the serialized object to its Kinetic compatible representation and while getting it back to do the reverse. Instead, it is quite plausible to keep the result of the serialization and modify it only when it is needed.

In addition, the serialization mechanism gives us the solution for referencing to other objects: During the serialization of an object, if any reference to already persistent object is found, the (dataClay) object ID of the persistent object is appended to the serialization message and serialization continues to the next field. Kinetic handler mimics this pattern: When a serialized object contains bytes for another serialized object, the "inner" object is stored into a separate key-value entry and the "outer" object just stores the (dataClay) object ID of the "inner" one. Furthermore, Kinetic handler returns the object IDs of all the newly stored objects. Data Service receives this result and refreshes its knowledge about persistent objects. Without processing the initial byte buffer, it would be impossible to know which objects become persistent.

The truth is that, currently, the key-value entries contain some extra information which is useful/necessary only for the Kinetic handler. If the key-value entries were formed as in the previous paragraph, their content would be fully understandable by the Data Service: Any object could be retrieved and be passed for deserialization to the Data Service, without any need for processing by the Kinetic handler. However, things cannot be so dreamy in the general case; class enrichments impose storing some extra information. This case is detailed in the corresponding section. Nevertheless, there are some special cases (arrays, maps, collections) where key-value entries can be passed directly from Kinetic drive to Data Service.

5.4.5 Processing of serialized objects

In the previous section, the content of value in key-value entries was covered intuitively. From now and on, we are focusing on the processing part of Kinetic handler. Kinetic handler's main task is to distinguish objects that exist into the serialization message of other objects and to separate them. Nevertheless, it only receives a byte array from the previous layer, the serialization mechanism. A single array of bytes does not bear any semantics and its processing would be impossible, especially in our case, where the morphology of an object always varies. As it was stated several times before, the structure of a user class is modeled in an instance of DBClass. Using the information from this instance, the bytes from a serialized object get meaning and their process is possible.

In Kinetic handler, two types of byte buffers are used: 1) One byte buffer, the input buffer, which contains the outcome of the serialization. There is always only one input buffer, and it probably contains multiple serialized objects. 2) There are also the output byte buffers, one per serialized object. The result in every output byte buffer is going to be the value for the key-value entry that corresponds to the object. From now and on, we will name storeObject the process of storing an object, like a method name. The first step of storeObject is to read some metadata from input buffer. The most important among this metadata is the notNullBitmap, which was explained in the previous chapter. This bitmap is of variable size and its size in bytes is calculated due to the DBClass, which knows how many nullable references exist in a user class. After this step, bytes for fields of the object follow.

While processing the bytes for the fields, there are two cases storeObject may encounter. The first and the easier one is to deal with a field of primitive type. In this case, since we know the type of the field (due to DBClass instance), we copy from the input buffer to the output buffer the exact amount of bytes that are dedicated for this field. Thankfully, primitive field have always value: Even if they have not been instantiated, they have their default value. It is worth to mention the difference between Kinetic and relational databases case. In Kinetic, we just copy bytes from one buffer to another and continue to the next field. On the opposite, Postgres handler requires the appropriate formation of a SQL statement, which implies extensive string manipulation using apostrophes, parentheses, etc. At the end, this results into a quite big and hard-to-understand piece of code whose purpose is mostly the formation of the SQL statement, rather than the data handling from the buffer.

The second case is when we have to deal with a reference. This case is not as trivial as before and has several sub-cases. When we encounter a reference, it is always quite possible for the reference to be null. However, this information does not depend on the morphology of the user class. So, it cannot be found in DBClass. Instead, it is highly dependent on the instance of the class. notNullBitmap, which comes with every serialized object (except the ones which have only primitive fields), plays this role: One bit per reference informs us whether a reference is null or not. Thus, the first step while encountering a reference is to check the appropriate bit in the notNullBitmap. If the reference is null (according to the bitmap), no bytes exist for the reference in the input buffer and the buffer handling continues to the next field.

On the opposite side, there are several cases when the reference is not null:

- **Case 1 – Reference to an already persistent object**: If the reference points to a persistent object, the serialization mechanism has appended only the (dataClay) object ID of the persistent object (along with some other metadata like the tag).
- **Case 2 – New serialized object**: This happens when the input buffer contains bytes for more than one object. In this case, the sub-object has to be extracted from the input buffer, to be copied into another output buffer and the remainder of the input buffer to be processed for the initial object. For this purpose, Kinetic handler calls recursively storeObject, since the serialization mechanism acts in a recursive way too.
- **Case 3 – Reference to an already encountered object**: If the reference points to an object that has been encountered before, then only the serialization tag is appended for this object, instead of re-serializing the object.

It is very important to mention the difference between case 1 and case 3. In case 3, an object is considered "encountered" if the handler has already processed this object in the **same** call of the makePersistent method. In other words, an "encountered" object has not been persistent before the current call of makePersistent method. On the other hand, an object is considered persistent (case 1) if it has been stored in **previous** makePersistent call.

But, how are these three cases distinguished? Kinetic handler uses a map, called *alreadyEncounteredObjects*, that has as key the tag of the already processed objects and as value their object IDs. If a tag exists in the map, it means that the object has been processed in the past (in the same call of makePersistent, though) and the bytes after the tag are for the next field (case 3). Otherwise, the reference can be either to a persistent object (case 1) or to a non persistent one (case 2). These cases are distinguished thanks to the metadata that every object has in the input buffer. More precisely, there is a byte for a flag called isPersistent. If it is true, then only the object ID for the persistent object follows (case 1). Otherwise, the serialization of the non-persistent object follows (case 2).

Kinetic handler strives to store key-value objects in an (almost) understandable format for Data Service. By doing so, retrieving an object from Kinetic drive will require the less possible processing. Thus, tags in every key-value entry should act as unique identifiers, as they do in serialized objects. However, if we extract the inner objects out of an outer one, tags lose their identification property.

For example, suppose we have objectA and objectB, with a relationship as depicted in Figure 5.3.

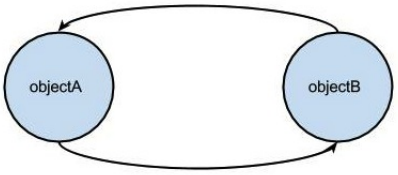


FIGURE 5.3: Example: Relationship between two objects.

If makePersistent is called for objectA, then both objects will be serialized. Furthermore, objectA will be tagged with 0, and objectB with 1. The reference of objectA to objectB falls into case 2, because objectB has never been encountered before, nor it is persistent. On the contrary, the reference of objectB to objectA falls into case 3, because objectA has been already encountered. So, only the tag 0 is used for this reference.

If objectA and objectB are separated and stored individually, they cannot reference each other anymore. For example, tag 0 in objectB cannot identify objectA. For this reason, the object ID should be used for referencing to other persistent object, as serialization mechanism does in case 1. Afterwards, any other reference to already encountered object should contain only the tag. For example, a second reference from objectB to objectA should append only the

tag of objectA, not its object ID. Thus, Kinetic handler should implement this inner tagging mechanism. One map and one set enable this mechanism:

- **alreadyEncounteredObjects** (Map from integer to object ID): This was described earlier: It is necessary for recognizing objects that have been processed in the past, in the same storeObject call. The reader should keep in mind that this map is unique and it is used by all the recursive storeObject calls.
- alreadyEncounteredTags (Set of integers): Every time storeObject is called, such a set is instantiated empty. This set contains the tags that the current storeObject call has already encountered. If a tag does not exist in the set, the tagged object is encountered for first time from the current storeObject call, and its object ID should be appended, along with the tag (the object ID can be found from alreadyEncounteredObjects). Otherwise, if the tag is in the set, it is enough for referencing the object.

Every tag that is into alreadyEncounteredTags is in the alreadyEncounteredObjects, too. But, not the reverse.

The pseudocode of the algorithm can be found at Algorithm 6. This algorithm is executed only when a reference to another object is found.

Last but not least, the reader can imagine the complexity of the SQL statement for handling references in the case of relational database handler.

5.4.6 Key schema

After this long description of what is stored in the value of a key-value entry, the only thing left is the key schema of the entry. The key schema does not make any surprises and follows the quite plausible pattern: class ID_object ID

The motivation that led to this schema derives from the entries ordering according to their key. Since entries are ordered, objects of the same class are grouped together, like tables of relational databases do. According to the Kinetic technology description, retrieval of objects that reside close to each other can be optimized. Later, we will see specific cases where we take advantage of this feature.

5.4.7 Superclasses storing

In OOP, when a class extends another class, the subclass inherits superclass' fields. dataClay takes this fact into consideration in its serialization mechanism, and Kinetic handler does too. As it was mentioned in the previous chapter, every stub class inherits at least one class, the class DataClayObject. Thus, serialized objects always contain bytes that correspond to its superclasses, at least DataClayObject. Kinetic handler appends these bytes to the output buffer of the corresponding object. Furthermore, processing the DataClayObject bytes is the criterion for reaching to the end of processing.

Algorithm 6 The algorithm for processing references

```
1: procedure processReference()
2:     tag ← inputBuffer.readInt()                                  ▷ Read tag
3:     if !alreadyEncounteredObjects.containsKey(tag) then
4:         isPersistent ← inputBuffer.readBoolean()
5:         if isPersistent = true then
6:             objectID ← inputBuffer.readObjectID()                ▷ Case 1
7:             alreadyEncounteredObjects.add(tag, objectID)
8:             alreadyEncounteredTags.add(tag)
9:             outputBuffer.writeInt(currentTag)
10:            outputBuffer.writeBoolean(true)
11:            outputBuffer.writeObjectID(objectID)
12:        else
13:            objectID ← newobjectID()                             ▷ Case 2
14:            alreadyEncounteredObjects.add(tag, objectID)
15:            alreadyEncounteredTags.add(tag)
16:            outputBuffer.writeInt(currentTag)
17:            outputBuffer.writeBoolean(true)
18:            outputBuffer.writeObjectID(objectID)
19:            inputBuffer.storeObject()
20:        end if
21:    else                                                         ▷ Case 3
22:        if alreadyEncounteredTags.contains(tag) then
23:            outputBuffer.writeInt(tag)
24:        else
25:            alreadyEncounteredTags.add(tag)
26:            objectID ← alreadyEncounteredObjects.get(tag)
27:            outputBuffer.writeInt(currentTag)
28:            outputBuffer.writeBoolean(true)
29:            outputBuffer.writeObjectID(objectID)
30:        end if
31:    end if
32: end procedure
```

Here, it is worth to mention a corner case which indicates that direct put on the Kinetic drive can be harmful. DBClass has a public method called hasNullableReferences which returns true if the modeled user class contains references, and vice versa. So, in case of receiving false after calling this method, it means that the user class has only primitive fields. Which, in our case, are just copied. So, someone might think that the byte processing can be avoided if the hasNullableReferences returns false, and that a direct put operation is possible. But, the truth is that DBClass models only the class itself and not its superclasses. Thus, hasNullableReferences is not aware of the superclasses, and whether they contain references or not. So, it is quite possible a class to have only primitive fields and its superclass a reference to another object. A direct put operation in this case would cause violation of the single-object-rule.

5.4.8 Iterative version

As it has been mentioned above, the serialization acts in a recursive way. Thus, doing the same in the Kinetic handler was more than straightforward. However, in cases where objects are nested inside others in big depth, it is quite probable to run out of memory resources, and end up with a stack overflow. For this purpose, an additional iterative method for storing objects and the corresponding class that acts as stack frame have been developed. This class contains among others several flags for overcoming code segments that must be run only once. This ends up having an iterative version which is a quite hard to understand, but memory safe. Describing the iterative version in detail would not add new concepts. That's why no further explanation is not attempted.

5.5 Retrieval of objects from the Kinetic drive

While the store operation has been detailed thoroughly, almost nothing has been said for its counterpart, the read operation. The truth is that the store operation has been designed so that it will facilitate the retrieval of an object from the Kinetic drive and it will be as more performant as possible. Thankfully, this optimization can be achieved in many cases.

Before proceeding with the description of the read operation in the Kinetic handler, it is crucial to learn some more things about dataClay and how it usually behaves: dataClay usually requests single objects from the persistence layer. In other words, it usually does not expect to receive many objects from the Kinetic drive in a single operation. For example, it requests for an object, which might have references to other objects, but it does not expect the referenced objects along with the initial one. Instead, it receives the initial object serialized, which contains the dataClay object IDs of the other serialized objects, though. And actually, it does not make sense to retrieve a big amount of objects with a single operation. It would lead to a very demanding request, which would take a lot of time to be served, and, at the end, only a small part of these objects would be accessed. Instead, every object is retrieved from the persistence layer, only when it is not handy in the memory.

However, there is a corner case, in which an object, along with its references objects, are expected. This case happens when we want to transfer/copy data from one data infrastructure to another. As someone might think, this does not happen often.

So, how is it specified whether we want to retrieve objects exhaustively or not? The answer is that this is achieved thanks to a boolean called *stopsAtPersistent*. When access to the data of persistence layer is needed, Data Service issues the appropriate call to the handler. This call contains several parameters, like the object and class IDs of the object we want to access, and other parameters. *stopsAtPersistent* belongs to the complementary parameters. If it is true, the access is exhaustive. Otherwise, only a single serialized object is expected.

From now and on, we will expose how Kinetic handler faces both situations. Let's first see the case when *stopsAtPesistent* is true. A few pages above, the motivation for having single objects in every key-value entry (the single-object rule) has been described extensively; moreover, we usually request single objects from the data infrastructure. Things seem to converge: It

would be ideal if objects were retrieved from the persistence layer and pushed directly to Data Service without any byte processing. Kinetic handler does almost so. There are few cases (maps, collections, array) that the retrieval is done directly without any processing. Regarding the rest, it depends on whether the class of the retrieved object has been enriched or not. For this purpose, every stored key-value Entry has some metadata in the beginning of the stored bytes, which are necessary for determining whether the class of the retrieved object has been enriched. If not, no more processing is needed. Otherwise, processing is inevitable.

On the other side, things cannot be dreamy when *stopsAtPersistent* is false. Multiple accesses to the Kinetic drive are needed. Byte processing is done and code that takes much longer time is run. The good thing is that stopsAtPersistent is usually true. Since the procedure follows a quite symmetrical pattern to the store operation, it is described.

At the end, it is worth to mention that the get operation acts recursively when *stopsAtPersistent* is false. No iterative version has been developed, since the recursive call is prevented most of the times.

5.6 Updating objects in the Kinetic drive

In a data sharing platform, the ability to modify data is probably the most important feature. If not, the cooperation among the users would be very restrictive and probably each of the users would end up having a personal copy of the data. dataClay, though, provides the modification feature and, actually, it is one of its fundamental ones. Thus, Kinetic handler has also to ensure that modification of data on the persistence layer is possible.

In the previous chapter, the bytecode analysis, which takes place during the registration of a class, was described. As it was said, one of the objectives of this analysis is to identify the methods that modify the state of the self-contained objects. Since dataClay knows what changes the state of an object it also knows when such a change takes places: Whenever a state-modifying method is called. So, when such a method is called, dataClay can also trigger the appropriate call for updating the data that rely on the persistence layer. More precisely, when a method that modifies at least one of the fields of an object, the update of the object on the persistence layer follows.

In Kinetic drive, when an entry must be updated, it is actually substituted by another one. Kinetic handler is not the exception: When dataClay issues an update operation, the workflow is exactly the same as when an object is first stored. Kinetic handler receives a serialized object, the handler processes it, finds the non persistent object which are stored individually and then overwrites the old object.

There is only one thing left for the update operation. In the second chapter, the Kinetic's inner mechanism for dealing with concurrency issues was described. Actually, this mechanism is not used in dataClay, in the Kinetic handler. There is no need to use it because dataClay itself does not permit the access to the persistence layer by more than one user per time. Thus, there cannot be any inconsistency in the shared object, by design. Furthermore, the double effort of Kinetic for updating an already updated object is avoided: if a user tries to update

an object and receives a VersionMismatchException, then he has to repeat his modification on the freshly modified object.

5.7 Objects deletion from Kinetic drive

After introducing the majority of the details that concern the Kinetic handler, the delete operation can be described quite fast. In analogous way as reading objects, every delete operation has the boolean parameter stopsAtPersistent. If it is true, the deletion of a single key-value entry is needed. On the other side, we need to delete the whole objects tree, whose the object ID of the root is known. In this case, the delete method calls itself and every node deletes its children nodes before deleting itself. Unfortunately, there is not much margin for optimization for this operation.

5.8 Removal of a class from Kinetic drive

There are some rare cases where we want to delete from the data infrastructure every instance of a class. If the objects were stored in a relational database, then a simple SQL statement "DROP TABLE" would be sufficient for accomplishing the remove operation. On the contrary, there is not any actual grouping for the key-value entries in the Kinetic drive, like tables do in the relational databases. Instead, deletion of every instance must be held individually. Thus, the keys for each instance of the class must be retrieved and then delete(key) must be issued for every key.

Before moving to the implementation part, a little refresher would be useful: Every key (and its entry) is ordered **lexicographically** in the Kinetic drive. So, this fact can facilitate drastically the grouping of objects. And our key schema does so. Since each instance of a class shares the same prefix in the key schema, each object of the class reside next to the other. And Kinetic promises an optimization for accessing adjacent key-value entries.

The ordering fact gives us an insight for the planning inside the Kinetic drive, but we still do not know how to receive the keys for the key-value entries that we are interested in. According to Kinetic API, two solutions are possible.

The first one is to use the method getNext method until we delete every instance of the class. More precisely, we are only given the class ID of the class we want to remove. As it was described in the section "Representation of classes on the Kinetic persistence layer", there is a key-value entry for the class schema, whose key is only the class ID. This is the first member of the class group. So, we can use getNext and remove every instance one by one. This operation stops only when the key of the next entry is of different class.

The second option involves the getKeyRange method. According to this method's declaration, three of the parameters passed to it are the first and the last key of the range we are interested and the maximum number of keys (call it max) we want to receive. If there are more keys in the range than max, then the first max will be returned. Otherwise, the whole range of keys is

returned. So, we know the first key of the range but not the last one. Nevertheless, there is no need to know the exact key, since every key is ordered lexicographically. Thus, if we define as upper key a non existent key which is bigger than each object we want to remove, but smaller from the next class group, then it can serve as the upper key limit in our purpose. In our case, the following key is used: *classID*˜ . Since ˜is bigger than _, the key range from *classID* to *classID*˜ includes every key of the class group we are interested of. Of course, when max keys are returned, the getKeyRange is executed again in order to remove the remaining objects of the class.

5.9 Class enrichment

In the first chapter, it was emphatically mentioned that, in current approaches of data sharing, 3rd parties are always restricted in the functionality that is provided by the data owner. The only solution for them is to create a copy of the data and then manipulate it at their wish; however, real cooperation and data sharing cease to exist after creating a copy.

Instead, dataClay enables both sides to work seamlessly. More precisely, class enrichments are possible in three ways: 1) Adding new fields in a class, 2) adding new methods in a class, 3) adding a new implementation for an existing method in a class. Obviously, Kinetic handler deals only with the first case, because the second and the third case affect the behavior of classes which has nothing to do with the persistence layer. Instead, changes in the fields of a class affect the state of its instances. And, actually persistence layer saves only the state of objects. On the contrary, changes in methods are reflected in the class files. Moreover, which method is exposed to whom depends solely on the model contracts.

In the case of relational databases, the representation of a class is achieved using a corresponding table, where the fields of the class are mapped to the columns of the table. So, enriching a class by adding new fields is reflected on the database by adding new columns in the corresponding table. On the contrary, Kinetic drive does not represent classes somehow. Every instance is independent of the other in Kinetic drive, even though they reside next each other. Nevertheless, modifications on the structure of a class affect all of its instances, at once. Even more, a modification also affects the objects of classes that extend the enriched one. Thus, if we wanted to update all the affected objects at the moment of their class enrichment, we would not be able, because we cannot find easily the instances of other classes that extend the enriched one.

Thus, we are not able to do anything with reasonable complexity, at the moment of the enrichment. The only action at the enrichment moment is to update the key-value entry for the instance of DBClass, whose class has been enriched. This update does not affect any of the "dirty" objects. Nonetheless, the updated DBClass enables to recognize the "dirty" objects. The update of a "dirty" happens at the moment of retrieval.

Before moving further to the adopted solution, it is important to clarify what is expected by Data Service when it requests an object whose class/superclass has been enriched. It expects a serialized object which has information (bytes) for all the fields, both old and new, of the class. More precisely, the deserialization method has been updated in order to deserialize

"enriched" objects. Thus, when we request a "dirty" object, we have to add the default values in the new fields: If the new field is of primitive type, it gets its default value. Otherwise, it will be a reference and we will not append extra bytes since null references in serialized objects have no bytes. Furthermore, the notNullBitmap must contain bits for every reference, even the new ones.

How is a "dirty" object recognized by Kinetic handler? In order to recognize an affected object, some metadata is added to the value of a key-value entry, while it is processed for storing. During the store operation, every object is stored using the latest DBClass for its class and superclasses. Thus, we mark every object with the number of fields for each of its classes. Afterwards, when an object is retrieved, the stored metadata are compared with the number of fields of the current DBClass instances. At least one difference means that the object is "dirty".

So, that's why every key-value entry is not shaped exactly in an understandable way for Data Service. The value of every key-value pair contains 4 more bytes for an integer in the beginning of the serialized object. These 4 bytes denote the number of classes and superclasses that are saved in the serialized object. Let's call it *classes*. Thereafter, *20 * classes* bytes of metadata follow. Every 20 bytes consist of 16 bytes for the class ID and 4 bytes for the number of fields of the class at the moment of store. If anything has not changed for any of the classes, then the object is not dirty. Otherwise, its new fields should be set to default values. This metadata are put in the beginning of the buffer, on purpose. If an object is not dirty and stopsAtPersistent is true, we want the retrieval of an object to be as fast as possible. So, in this case we just remove the initial metadata bytes from the buffer and return the remaining, which is fully understandable by Data Service. No byte processing is needed. So, a huge workload is avoided, which would take a lot time and it would be meaningless.

Last but not least, how are the new fields recognized when we need to update a dirty object? The solution comes by dataClay itself. Thanks to the bytecode analysis, every new field is added at the end of the fields. This change is also reflected in the DBClass in the same way. Thus, since we know the number of the already stored fields, which are in the beginning of the DBClass, we can also distinguish the new fields. Furthermore, fields are never removed from user classes. Hence, once a field is added in a user class, its position in the class regarding the other fields will never change.

5.10 Storing collections in the Kinetic drive

As we have seen in the previous chapter, there are some special cases where the serialization does not follow the general pattern. Namely, collections, maps and arrays are serialized in a special way. Obviously, the Kinetic handler treats serialized objects of these classes in a special way, too. In this section, handling of collections is going to be presented. The other cases follow in the next sections.

Honestly talking, the differences between the general serialization algorithm and the collection one concern mostly some extra metadata rather than the actual way that collection members

are serialized. Thus, it would be meaningless and repetitive to describe in detail the processing of serialized collections. Instead, we will focus on the things that make collections unique.

Someone might wonder how a collection wrapper is recognized. Every serialized object contains 16 bytes that denote the class ID of the object. These bytes are written in the second step of the algorithms presented in the previous chapter. In the case of the three wrappers, there is no exception. The class ID that is appended in the serialized object is the one of the corresponding wrapper. So, if any of these 3 class IDs is recognized during the handling of serialized objects, the general path is bypassed and special methods take care of the wrappers, instead. After reading all the metadata that precedes the bytes for the serialized collection members, the handling of the members is held by the same method that handles reference fields in the general case. After all, collection members are objects!

It has been already mentioned that retrieval of collections, maps, arrays from Kinetic drive can be direct. The reason is that the java wrappers are never enriched in comparison to other user classes. Thus, no metadata are appended like in user classes. Only the serialized wrapper. Which is fully understandable by Data Service. But, it is important to emphasize that just the wrappers do not need enrichment check, not the collection members, which are regular objects.

The rest of the section describes how to optimize retrieval of collections members in the Kinetic drive. It has been mentioned several times the notion of grouping by selecting the appropriate keys for the key-value entries. Regarding collections, taking advantage of this feature makes sense more than any other time.

According to the selected key schema, objects of the same class always reside together in the Kinetic drive. Thus, members of a collection reside inside the outer group of their class. But it is very possible other objects which are not members of the collection to reside between the collection members. This happens because the second part of every key is the object ID which is generated randomly.

Before describing the proposed solution, let's see how an object ID is formed. An object ID is of type UUID. It consists of 16 bytes, which can be broken into two parts, 8 bytes for the *mostSignificantPart*, and 8 bytes for the *leastSignificantPart*. In other words, an objectID is made out of 2 longs.

The proposal is: We can generate randomly an object ID, which will be used for the wrapper entry itself. Then, the *mostSignificantPart* of this ID will be used for the *mostSignificantPart* of all the collection members' IDs. By doing so, we achieve having collection members into a subgroup inside the outer group of the class. However, if the *leastSignificantPart* is random, the members of the collection probably will not have the same order as in the collection. Instead, we can create the *leastSignificantPart* manually, by taking the *leastSignificantPart* of the wrapper ID and increment it for every collection member. By doing so, the members of the collection are ordered. Moreover, the probability to overwrite a collection member is very small, since 8 bytes can produce IDs for 2^64 - 1 collection members.

Last but not least, it is worth to mention the motivation of keeping objects IDs 16 bytes long, despite the fact that we can store keys up to 4 KB in the Kinetic drive. The motivation is to

keep every key-value in an understandable format by Data Service. If we were using a kind of internal object ID which exceeded the 16 bytes (for example, 16 bytes same for every member and incrementing number), this ID would be useful only for the Kinetic handler. Even if only the wrapper were requested with stopsAtPersistent true, it would have to map the internal Kinetic ID to the dataClay ones. Which would impose extra processing of bytes.

5.11 Storing maps in the Kinetic drive

After introducing the handling of collections, there is no mystery left to reveal for maps. Just, most of the processing is done twice. Moreover, the aforementioned optimization can be adapted so that the keys and the values will be grouped.

5.12 Storing arrays in the Kinetic drive

Arrays can contain two types of elements: either primitive elements or references to other objects. In the latter case, the handling is done more or less in an analogous way as collections. Moreover, the grouping optimization can be applied in the array case too.

On the other side, when we want to store an array which contains primitive elements, the process can be quite fast and without any need for calling the same method recursively (or its iterative equal). Since we know both the type of the elements and the number of them in the array, the number of bytes dedicated for the array into the serialized buffer can be calculated really fast and the corresponding bytes are read at once.

However, if we do not pay attention, we will probably run into trouble. As the reader might remember, every key-value in the Kinetic drive has restriction on the data size. The maximum size of the value is 1 MB. If we want to store an array of 300,000 integers, it cannot fit into one key-value entry, which would lead to an Exception. It is the only time we need to break the single-object rule: The wrapper cannot contain the whole array.

The solution in this case is to break the array into small enough pieces, store them individually and keep track of the array components into the array wrapper. When a big array of primitive type is requested, its wrapper is retrieved in the beginning, it is checked if its array is segmented, and if yes, its parts are retrieved and recomposed into the wrapper. The only thing left is: How can we recognize a big array? Thanks to the bytebuffer API, a single of the method readableBytes() returns the number of bytes that the buffer contains. Last but not least, the grouping trick that was described in the collection section can be used here too, in order to retrieve the array segments as fast as possible.

Chapter 6

Evaluation of Kinetic handler

Objective of this chapter is to provide sense about the performance of Kinetic handler. Firstly, the time needed for basic operations (such as put and get) is presented. Afterwards, the impact of class enrichment is examined. Last, we see the behavior of Kinetic handler for arrays of primitives (of variable lengths).

6.1 Main Operations

The first test run was for the elapsed time of store operation. Fifty simple objects (only with 2 primitive fields, and no references) were stored. The time needed for their store can be found in Figure 6.1.

According to the results, the time needed for storing a single object is around 60 ms, most of the times. Nevertheless, there are two more conclusions for which there is no explanation, right now. It depends solely in the inner mechanisms of Kinetic drive.The first fact is that similar results are grouped. In Figure 6.1, we see 2 groups: One group with elapsed time around 60ms, and another with elapsed time around 80ms. The second fact is there are some spikes during the store operation. In such cases, the elapsed time is even tripled.

The second case involves the store of an object which contains a reference to another un-stored object. Thus, two objects are going to be stored, after processing the serialization message by Kinetic handler. The result are in Figure 6.2.

As someone might expect, the elapsed times are doubled in comparison to those of Figure 6.1. More precisely, grouping is present here too. More precisely, there is one group of 120ms and another of 180ms, which are doubled of the values of Figure 6.1. Last, spikes are not missing from this case.

In Figure 6.3, we see that if an object contains a reference to another persistent object, the store operation is not affected and it behaves similarly with the case of Figure 6.1. This makes sense, since dealing with references to other persistent objects requires just the processing of some extra bytes in the serialized object. On the contrary, references to non-persistent objects

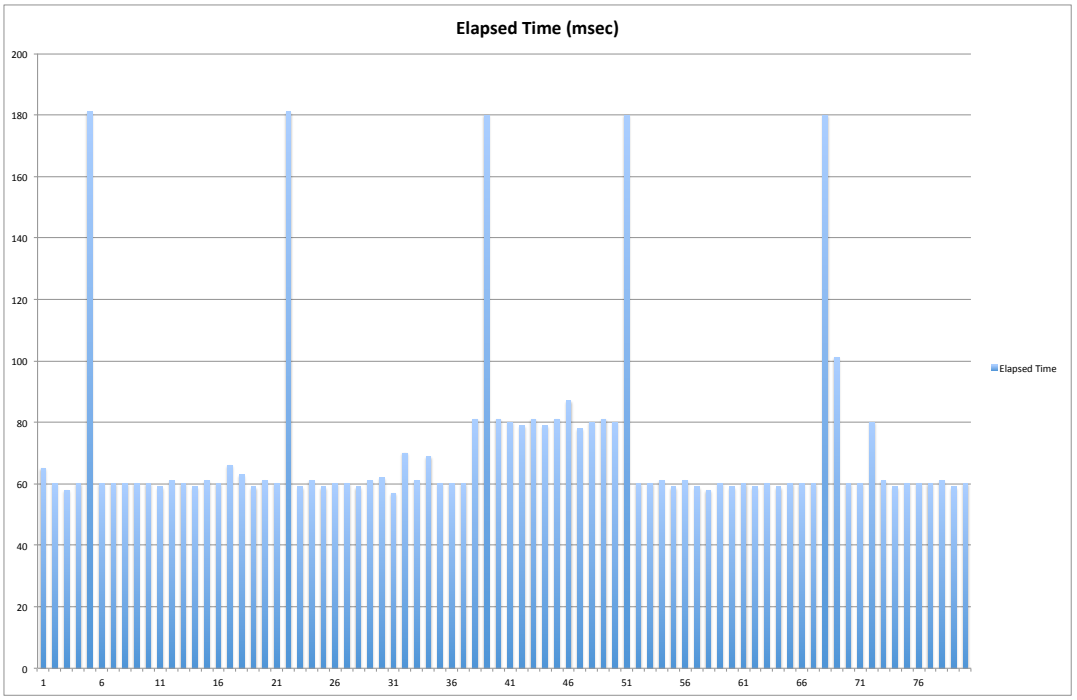


FIGURE 6.1: Elapsed time for storing simple objects.

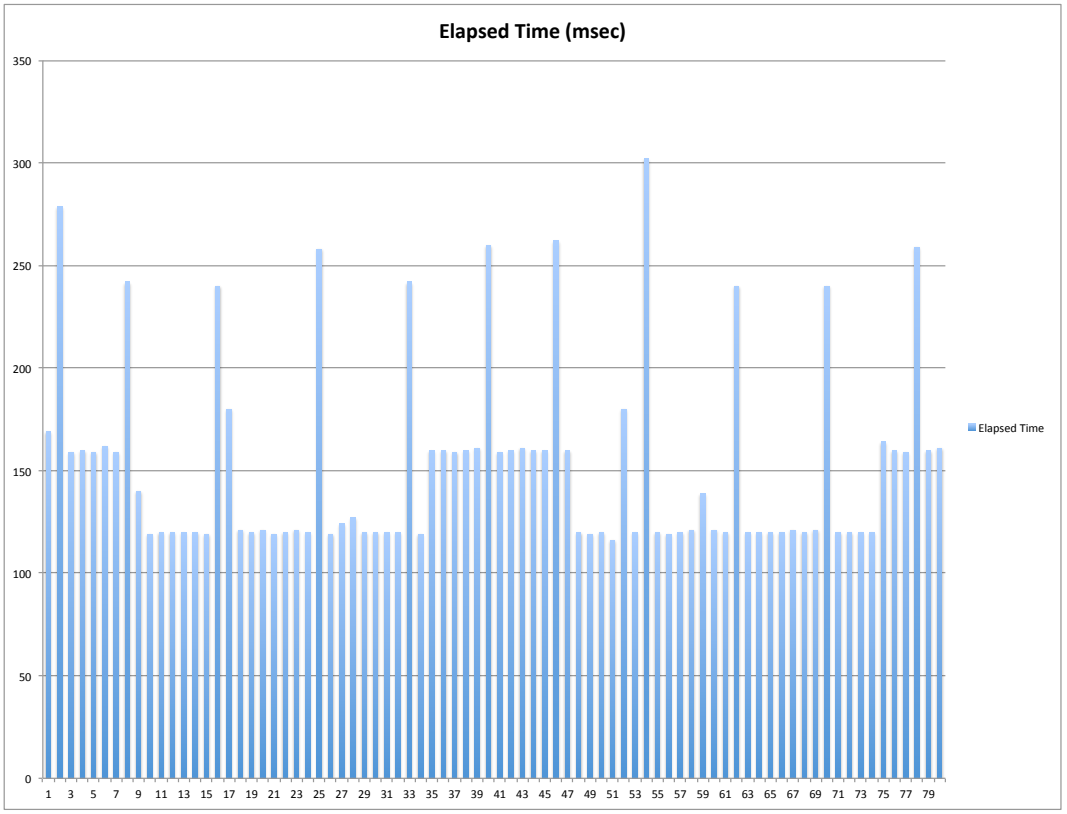


FIGURE 6.2: Elapsed time for storing objects with one reference.

require the processing of the whole serialized referenced object and, of course, one more put operation to Kinetic drive.

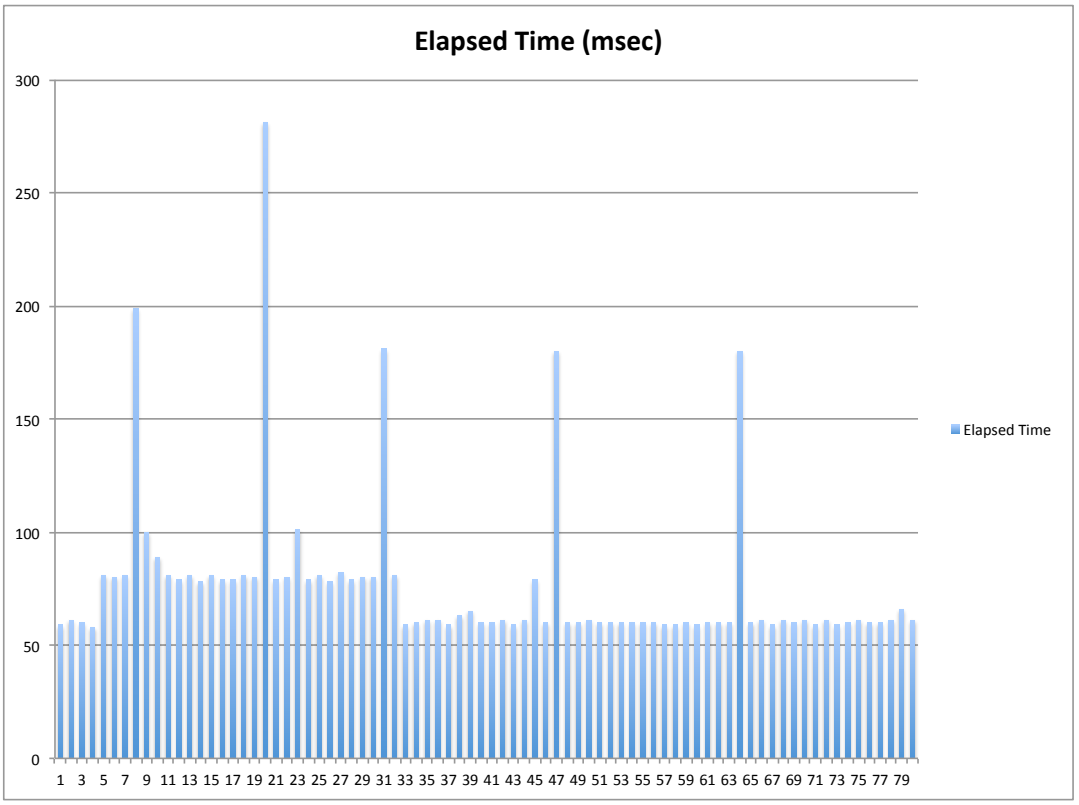


FIGURE 6.3: Elapsed time for storing objects with one reference to persistent object.

Regarding the retrieval of objects (Figure 6.4), we notice that they are one order of magnitude faster than storing (as in Figure 6.1).

6.2 Impact of class enrichments

In Figure 6.5, the required time for obtaining objects of an enriched class is presented. It is very important to mention that the total elapsed time (red lines in Figure 6.5) consists of both enriching a "dirty" object and updating the "dirty" key-value entry in Kinetic drive. As it is depicted in the blue sublines, most of the time is consumed for updating the key-value entry in Kinetic drive, rather for retrieving and enriching a "dirty" object. According to the current implementation, the user is compelled wait for the whole operation, while he is interested for the enriched object which is ready much earlier. Such cases like this, can be dramatically optimized by both using the Kinetic API for asynchronous operations and implementing the corresponding callback handler. This was not possible throughout this master thesis, because of the time limits.

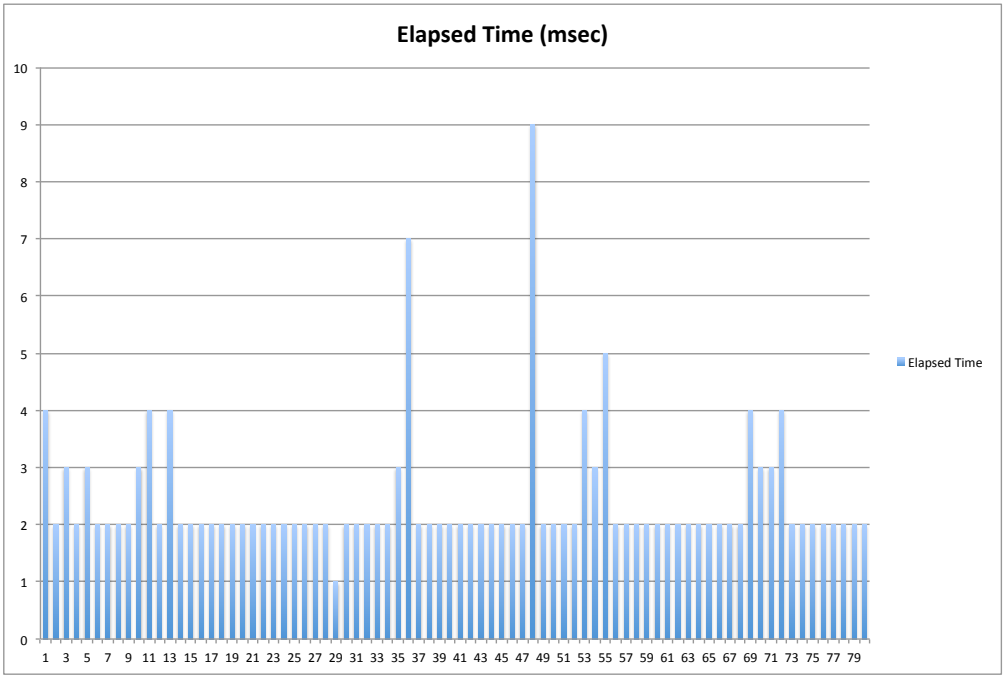


FIGURE 6.4: Elapsed time for retrieving objects.

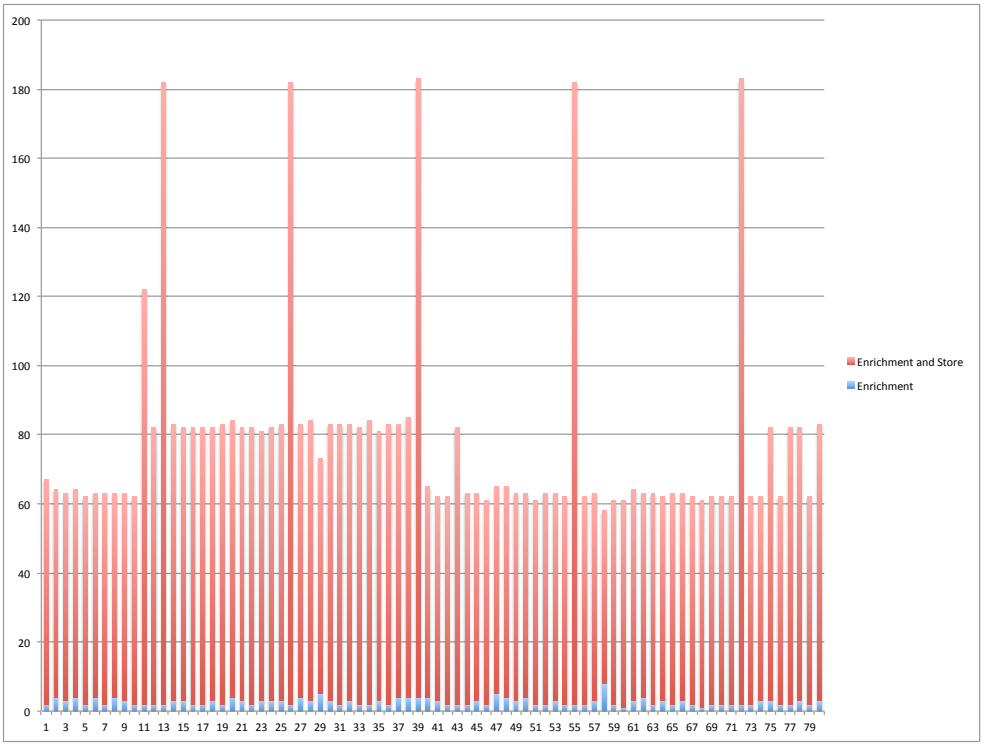


FIGURE 6.5: Elapsed time for retrieving objects of an enriched class.

6.3 Retrieving big arrays

In this section, the retrieval of big arrays of type int is examined. The objective is to check whether the continuous key schema can perform better in comparison to using random suffixes. In other words, it is tried to check the key schema trick mentioned in section 4.10.

Unfortunately, no optimization was noticed. For this test, 10 arrays of 25,000,000 ints were stored, with and without the continuous key schema for their segments. The size of these arrays is around 95MBs, which ends up into having 95 segments of the array because of the 1MB limit of the key-value entries. Afterwards, the arrays were retrieved. Figure 6.6 shows elapsed times for retrieving the 10 arrays, when they were stored with random suffix for their keys. Figure 6.7 shows elapsed times for retrieving the 10 arrays, when they were stored with continuous suffix for their keys. Their times do not present any big difference. Time unit is the millisecond.

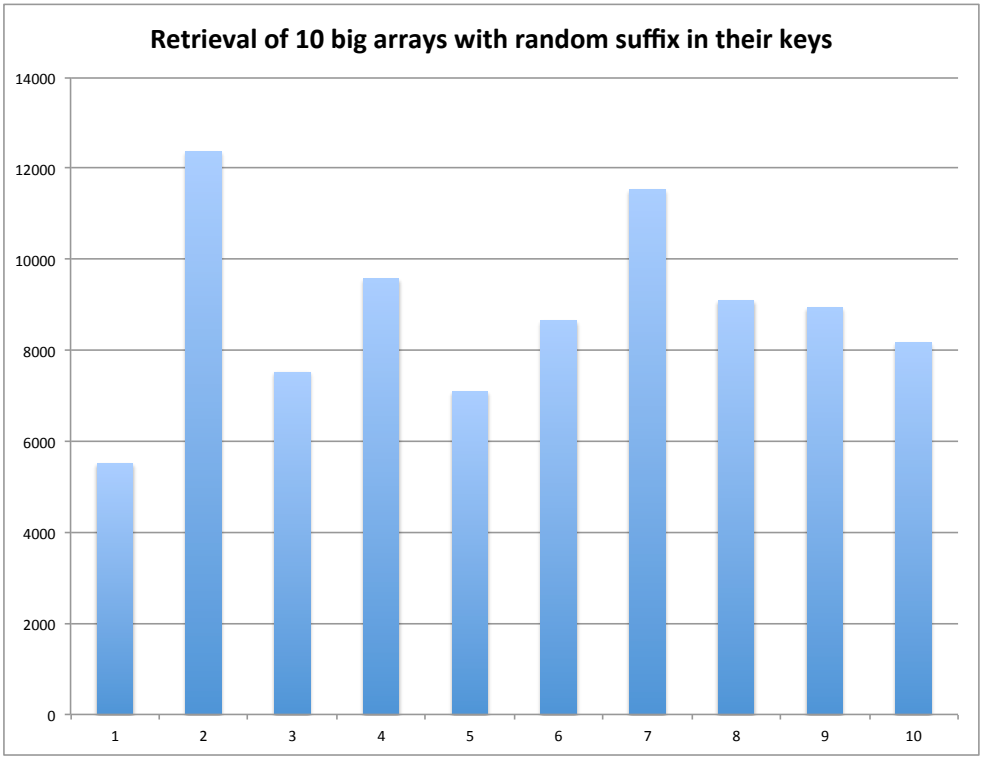


FIGURE 6.6: Elapsed time for retrieving arrays with random suffix in their keys.

However, it was noticed the impact of the internal cache. Every array, in both cases (either continuous or random key), was retrieved twice. Always the second retrieval was much faster. The case of continuous key schema is presented in Figure 6.8. The behavior is the same for random key schema too.

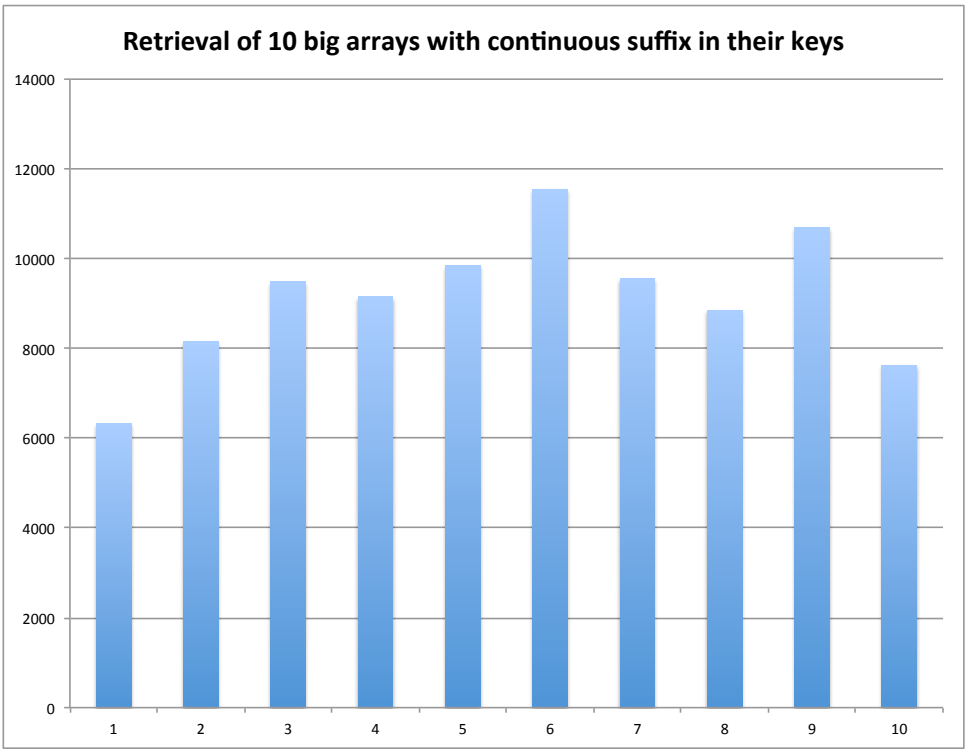


FIGURE 6.7: Elapsed time for retrieving arrays with continuous suffix in their keys.

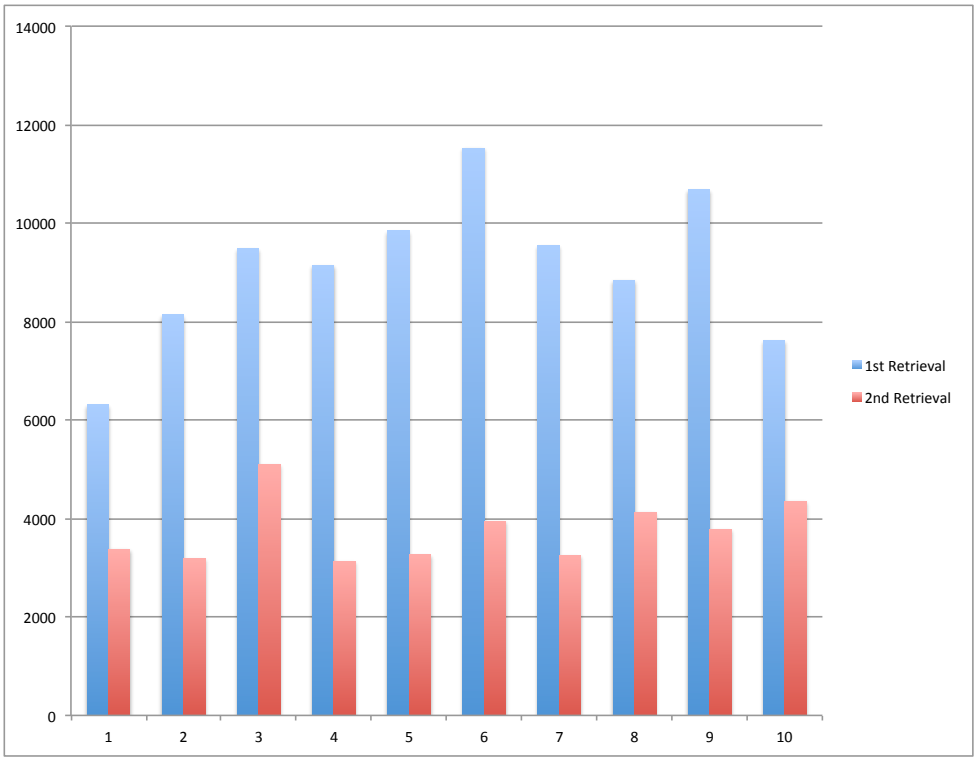


FIGURE 6.8: Elapsed time for retrieving arrays twice in the row.

Chapter 7

Conclusion

At the end, what is the conclusion after completing this research? Even though the results derived from the tests, as they have been presented in the previous chapter, are not the optimum, the integration of Seagate Kinetic technology into dataClay looks very promising for dataClay. First of all, Kinetic technology fulfills the long desired feature of dataClay for byte addressability. In other words, serialization mechanism and the data infrastructure, that is, Kinetic drives, speak the same language which is bytes. But, moreover and more interestingly, there is huge margin for improvement for both technologies and, also, for their integration. Further details follow.

7.1 Serialization mechanism

Serialization mechanism has been designed in a way that persistent data will bear their semantics. For example, SCOs that have been stored in a relational database are semantically rich. Main reason of choosing this design was the potential to access data directly to the data infrastructure. However, such a need for applications that want direct access to the data infrastructure does not arise. Furthermore, accessing data this way breaks the long desired feature of computation close to the data. Thus, the current serialization mechanism seems to be obsolete. Indeed, serialization mechanism currently compels the data infrastructure handlers to do a lot of work that can be avoided. More precisely, processing right now takes place in both sides, serialization and handlers. This made sense only in the case of storing semantically rich data. Since there is not such a need, as it has been explained earlier, a more efficient serialization mechanism is needed. For this purpose, Storage Systems research group at BSC has started the development of a new serialization mechanism that avoids the double processing and makes handlers' work much easier: There will be no need anymore for processing serialized objects at handlers stage. Handlers will just be in charge of passing the serialized objects in the data infrastructure. On the opposite side, when retrieving objects whose class has been enriched, processing cannot be avoided as it has been explained in the fourth chapter. But, if the class is not affected the stored objects can be just passed from the data infrastructure to Data Service.

7.2 Seagate Kinetic Technology

Seagate Kinetic Technology still has a lot of progress to achieve (as dataClay has too). Even though Kinetic has adopted the simple abstraction of key-value objects, the truth under the hood is a bit more complicated. Actually, storing key-value objects has not been achieved on hardware level, yet. Instead, an embedded system running Linux is in charge of storing the key-value objects in a LevelDB database. In other words, the key-value store is software-defined at the moment. This middle layer adds only overhead on the performance of the Kinetic handler. When the key-value store will be implemented on hardware level, operation from and to the Kinetic drive are expected to be optimized.

Further optimization can be achieved by changing the technology of the drive itself. Right now, Kinetic drives are implemented on classical Hard Disk Drives. Implementing Kinetic technology on Solid State Drives would probably offer further improvement to the performance.

Bibliography

[1] Amir Gandomi and Murtaza Haider. Beyond the hype: Big data concepts, methods, and analytics. *International Journal of Information Management*, 35(2):137–144, 2015.

[2] Edd Dumpbill. What is big data? https://beta.oreilly.com/ideas/what-is-big-data, January 2012. Accessed: 2015-07-03.

[3] John Gantz and David Reinsel. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the Future*, 2007: 1–16, 2012.

[4] Kinetic open storage documentation wiki. https://developers.seagate.com. Accessed: 2015-07-03.