
**National Technical University of Athens**

School of Electrical and Computer Engineering

Division of Computer Science


# Wavelet-based Algorithms for Approximate Processing in the Big Data Era

DOCTORAL DISSERTATION


**IOANNIS A. MYTILINIS**



Athens, November 2019

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

# Wavelet-based Algorithms for Approximate Processing in the Big Data Era

DOCTORAL DISSERTATION

**IOANNIS A. MYTILINIS**

**Advisory Committee:** Nectarios Koziris
Dimitrios Tsoumakos
Panayiotis Tsanakas

Accepted by the seven-member committee on 20/11/2019.

| | | |
|---|---|---|
| . . . . . | . . . . . | . . . . . |
| Nectarios Koziris | Dimitrios Tsoumakos | Panayiotis Tsanakas |
| Professor NTUA | Associate Professor | Professor NTUA |
| | Ionian University | |
| . . . . . | . . . . . | . . . . . |
| Antonios Deligiannakis | Symeon Papavassiliou | Aris Pagourtzis |
| Associate Professor TUC | Professor NTUA | Associate Professor NTUA |

. . . . . .
Yannis Kotidis
Associate Professor AUEB

Athens, November 2019

. . . . . .

**IOANNIS A. MYTILINIS**
Doctor of Electrical and Computer Engineering, NTUA

To my family and my beloved Ioanna.

# Acknowledgments

This dissertation contains the main results of my research as a graduate student in the Computing Systems Laboratory (CSLab) of the National Technical University of Athens. This work would not have been completed without the invaluable support of my advisors Assoc. Prof. Dimitrios Tsoumakos and Prof. Nectarios Koziris.

Dimitrios Tsoumakos, who was my main advisor and mentor, acted as a source of encouragement and inspiration throughout all these years. His guidance was of paramount importance and made my PhD years a unique experience that helped me evolve both as a researcher and as a person. Apart from his contribution to technical stuff like paper writing, Dimitrios had a great impact on my mindset in general. He taught me to set high quality standards in my work and not get discouraged no matter what obstacles may appear. Whatever I have achieved during my PhD, would not have come true without his help and I am deeply thankful for that.

I would also like to thank Nectarios Koziris for the opportunity that he gave me to join CSLab and be a part of a wonderful team. Nectarios always cares to provide all the necessary support, both intellectual and material, to his students and colleagues. I was lucky enough to be a part of his team and carry out my PhD research in an environment that stimulates innovation and where access to cutting edge technologies is always available.

Finally, I would like to thank my friends and colleagues Katerina Doka, Giannis Giannakopoulos, Ioannis Konstantinou and Nikos Papailiou that marked these years and without them the beautiful journey of PhD would not have been the same.

# Contents

# List of Figures

# List of Tables

# Περίληψη

Τα σύγχρονα συστήματα αναλυτικής επεξεργασίας καλούνται να αντιμετωπίσουν έναν τεράστιο όγκο δεδομένων. Ο όγκος αυτός των δεδομένων καθώς και οι αυστηρές απαιτήσεις για τον χρόνο απόκρισης των ερωτημάτων δίνουν όλο και αυξανόμενη έμφαση στην αποδοτικότητα των τεχνικών Προσεγγιστικής Επεξεργασίας Ερωτημάτων (ΠΕΕ). Η βασική ιδέα της ΠΕΕ είναι η κατασκευή μιας συμπιεσμένης αναπαράστασης ενός συνόλου δεδομένων και η εκτέλεση των ερωτημάτων, που θέτουν οι χρήστες, πάνω σε αυτή τη σύνοψη αντί για τα αρχικά δεδομένα. Μία σημαντική πρόκληση τα τελευταία χρόνια είναι η κατασκευή συνόψεων που παρέχουν αιτιοκρατικές εγγυήσεις για την ποιότητα του αποτελέσματος. Οι ντετερμινιστικές εγγυήσεις παρέχουν ισχυρά αποτελέσματα και είναι ευκολότερο για τους χρήστες να τις κατανοήσουν και να τις ερμηνεύσουν. Καθώς τα δείγματα και τα sketches συνήθως παρέχουν στατιστικές εγγυήσεις, για την παροχή αιτιοκρατικών εγγυήσεων καταφεύγουμε κυρίως σε τεχνικές όπως τα ιστογράμματα και τα wavelets. Λόγω της ικανότητάς του να προσεγγίζει έντονες ασυνέχειες, ο μετασχηματισμός wavelet έχει αποδειχτεί ένα αρκετά αποδοτικό εργαλείο για τη μείωση του μεγέθους των δεδομένων. Ωστόσο, οι υπάρχουσες τεχνικές οι οποίες είναι βασισμένες στην χρήση των wavelets και οι οποίες παράλληλα στοχεύουν στην ελαχιστοποίηση του παρατηρούμενου μέγιστου σφάλματος πάσχουν από μεγάλη πολυπλοκότητα που καθιστά την χρήση τους μη πρακτική. Επιπλέον, δεν μπορούν να χειριστούν αποδοτικά το πρόβλημα σε πολυδιάστατα δεδομένα. Ως εκ τούτου, στο πρώτο μέρος της διατριβής προτείνω παράλληλους αλγορίθμους που εκμεταλλεύονται τις βασικές ιδιότητες του μετασχηματισμού wavelet και κατασκευάζουν αποδοτικά συνόψεις που ελαχιστοποιούν μη-Ευκλείδιες μετρικές σφαλμάτων. Η πειραματική αξιολόγηση στο κατανεμημένο σύστημα επεξεργασίας Hadoop έδειξε ότι οι προτεινόμενοι αλγόριθμοι επιτυγχάνουν γραμμική κλιμακωσιμότητα και μπορούν

να επιταχύνουν την κατασκευή της σύνοψης μέχρι και 20 φορές όταν ο αλγόριθμος μπορεί να τρέξει πλήρως παράλληλα στην συστοιχία. Το δεύτερο μέρος της διατριβής μελετάει το πρόβλημα σε περιβάλλοντα ροών δεδομένων που συναντάμε σε εφαρμογές IoT. Η εποχή του IoT έχει προκαλέσει μια μετατόπιση των συστημάτων από ισχυρούς υπολογιστικά διακομιστές σε συσκευές που λειτουργούν "στην άκρη του δικτύου" κι έχουν περιορισμένες δυνατότητες επεξεργασίας και μνήμης. Οι αλγόριθμοι που σχεδιάζονται για τέτοιες αρχιτεκτονικές θα πρέπει να έχουν χαμηλή χρονική πολυπλοκότητα και ελάχιστο αποτύπωμα στη μνήμη. Επίσης, σε πολλές εφαρμογές ροών δεδομένων, τα πιο πρόσφατα δεδομένα θεωρούνται πιο σημαντικά. Το μοντέλο κυλυομένου παραθύρου είναι μια ιδιαίτερη περίπτωση επεξεργασίας ροών δεδομένων, όπου διαρκώς μόνο τα πιο πρόσφατα στοιχεία παραμένουν ενεργά και τα υπόλοιπα απορρίπτονται. Καθώς στις IoT εφαρμογές η διαθέσιμη μνήμη είναι συνήθως πολύ μικρότερη από το μέγεθος του παραθύρου, τα ερωτήματα απαντώνται από συνόψεις που κατασκευάζονται σε πραγματικό χρόνο. Για την αποτελεσματική κατασκευή τέτοιων συνόψεων παρουσιάζονται αλγόριθμοι βασισμένοι σε wavelets. Οι προτεινόμενοι αλγόριθμοι παρέχουν ντετερμινιστικές εγγυήσεις και παράγουν σχεδόν ακριβή αποτελέσματα για μια ποικιλία κατανομών δεδομένων και φόρτου ερωτημάτων.

# Abstract

Modern analytics involve computations over enormous numbers of data records, which often arrive in the form of high-throughput streams. The need for real-time processing of huge amounts of data places increasing emphasis on the efficiency of approximate query processing (AQP). A common practice for enabling AQP is to construct a lossy, compressed representation of a dataset and execute user queries against these synopses instead of the original data. A major challenge over the past years has been the construction of synopses that provide deterministic quality guarantees, often expressed in terms of maximum error metrics. Deterministic guarantees are strong and easier for the user to understand and interpret. As samples and sketches usually provide statistical guarantees, deterministic schemes are mainly supported by space-partitioning techniques such as histograms and wavelets. By approximating sharp discontinuities, wavelet decomposition has proven to be a very effective tool for data reduction. However, existing wavelet thresholding schemes that minimize maximum error metrics are constrained with impractical complexities for large datasets. Furthermore, they cannot efficiently handle the multidimensional version of the problem. In order to provide a practical solution, the first part of this dissertation proposes parallel algorithms that take advantage of key-properties of the wavelet decomposition and efficiently construct synopses that minimize non-Euclidean errors. The experimental evaluation over the Hadoop distributed processing framework showed linear scalability with both the data and cluster size; when the whole execution fits in the cluster and all workers can run fully in parallel, a synopsis construction speedup of $20\times$ is witnessed. The second part of the thesis targets the problem in an IoT streaming environment. The IoT era has brought forth a computing paradigm shift from traditional high-end servers to "edge" devices

of limited processing and memory capabilities. Thus, the designed algorithms for such architectures should be "cheap" in time complexity and have a minimal memory footprint. Moreover, in many streaming scenarios, fresh data tend to be prioritized. A sliding-window model is an important case of stream processing, where only the most recent elements remain active and the rest are discarded. As in IoT scenarios the available memory is typically much less than the window size, queries are answered from compact synopses that are maintained in an online fashion. For the efficient construction of such synopses, wavelet-based algorithms are presented. The proposed algorithms provide deterministic guarantees and near exact results for a variety of data distributions and query workloads.

# Εκτεταμένη Περίληψη

## 0.1 Εισαγωγή

Τα τεχνολογικά επιτεύγματα και οι κοινωνικές εξελίξεις της εποχής μας έχουν σαν αποτέλεσμα μια άνευ προηγουμένου παραγωγή τεράστιων όγκων δεδομένων, τα οποία συχνά αναφέρουμε ως "Μεγάλα Δεδομένα"[big]. Επιχειρήσεις, κυβερνήσεις και ψηφιακές υποδομές καθημερινά συνεισφέρουν σε αυτή τη νέα πραγματικότητα. Η αφθονία των συνόλων δεδομένων που υπάρχουν διαθέσιμα για επεξεργασία έχει οδηγήσει τόσο τον ακαδημαϊκό κόσμο, όσο και τη βιομηχανία στην υιοθέτηση *data-driven* προσεγγίσεων.

Ακολουθώντας το πρότυπο OLAP [CD97], οι σύγχρονες εφαρμογές ανάλυσης δεδομένων εμπεριέχουν υπολογισμούς συναθροίσεων πάνω από σύνολα δεδομένων που περιλαμβάνουν μεγάλο πλήθος εγγραφών αλλά και διαφορετικών διαστάσεων. Για παράδειγμα, μια χρηματιστηριακή εταιρία χρειάζεται να συγκρίνει τις τρέχουσες τιμές των μετοχών με το ιστορικό των μέσων όρων σε διαφορετικές κλίμακες (πχ., ανά εβδομάδα, μήνα κλπ.) προκειμένου να αποφανθεί αν ένα προϊόν είναι υπερεκτιμημένο ή υποτιμημένο. Ο υπολογισμός τέτοιων μέσων όρων παραδοσιακά επιτυγχάνεται μέσω της σειριακής προσπέλασης ενός μεγάλου τμήματος μιας βάσης δεδομένων. Σε πολλές περιπτώσεις, οι προσπελάσεις αυτές μπορεί να γίνουν ιδιαίτερα ακριβές και τα εργαλεία επεξεργασίας δεδομένων που διαθέτουμε να μην είναι ικανά να τις διαχειριστούν αποδοτικά. Στην περίπτωση όπου έχουμε μεγάλα, ετερογενή δεδομένα, ακόμα και τα γρηγορότερα συστήματα βάσεων δεδομένων μπορεί να χρειαστούν ώρες ή μέρες για να απαντήσουν και τα πιο απλά ερωτήματα. Ο υπολογισμός ενός μέσου όρου πάνω από 10

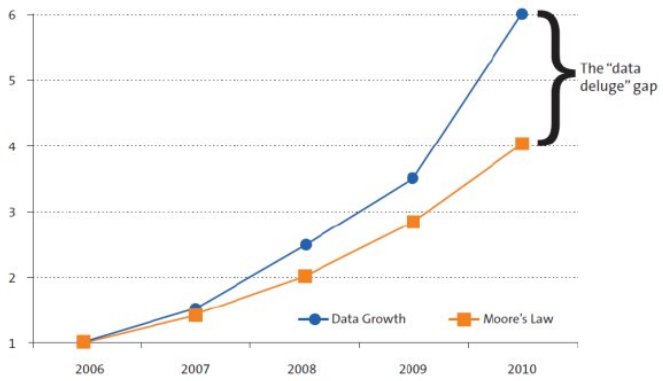
terabytes δεδομένων, τα οποία βρίσκονται αποθηκευμένα σε 100 μηχανήματα μπορεί να χρειαστεί περίπου 30-45 λεπτά επεξεργασίας στο Hadoop αν τα δεδομένα είναι στο δίσκο και γύρω στα 5-10 λεπτά αν τα δεδομένα είναι στη μνήμη [AMP+13].

Καθώς η εξερεύνηση ενός συνόλου δεδομένων είναι μια *διαδραστική κι επαναληπτική διαδικασία* [Moz15], τέτοιοι χρόνοι απόκρισης δεν είναι αποδεκτοί σε πολλές περιπτώσεις εφαρμογών. Σε ένα άλλο παράδειγμα, οι αναλυτές σε μια εμπορική επιχείρηση αναλύουν τα δεδομένα των πωλήσεων για να κατανοήσουν τις επιδόσεις της επιχείρησης με βάση διαφορετικές διαστάσεις (πχ, επίδοση σε διαφορετικά προϊόντα ή σε διαφορετικές γεωγρφικές περιοχές). Σε ένα τέτοιο παράδειγμα, οι διαδραστικοί χρόνοι απόκρισης είναι κρίσιμης σημασίας, καθώς οι αναλυτές θα πρέπει να μπορούν γρήγορα κι εύκολα να μεταβαίνουν μεταξύ υποθέσεων και πραγματικότητας. Μελέτες πάνω στην αλληλεπίδραση ανθρώπου-μηχανής έχουν δείξει ότι οι άνθρωποι χάνουν την προσοχή τους αν ο χρόνος απόκρισης ενός ερωτήματος είναι μεγαλύτερος του ενός δευτερολέπτου [Shn84]. Προκειμένου να ανταποκριθούν στις όλο και αυξανόμενες απαιτήσεις που προβάλει η ανάλυση δεδομένων στις μέρες μας, τόσο τα εμπορικά συστήματα, όσο και τα συστήματα ελεύθερου λογισμικού προσπαθούν να βελτιώσουν τους χρόνους απόκρισης που προσφέρουν μέσω διαφόρων τεχνικών όπως: ο παραλληλισμός, η δεικτοδότηση των δεδομένων, και η βελτίστοποίηση της εκτέλεσης των ερωτημάτων.

Παραδοσιακά, οι περισσότερες από αυτές τις τεχνικές προσπαθούν να χρησιμοποιούν με αποδοτικό τρόπο την διαθέσιμη μνήμη. Ωστόσο, η αποθήκευση όλων των χρήσιμων δεδομένων στην μνήμη δεν αποτελεί μια ρεαλιστική επιλογή στην εποχή των Μεγάλων Δεδομένων. Η αγορά μνήμης που να χωράει εξ' ολοκλήρου ένα μεγάλο σύνολο δεδομένων είναι υπερβολικά ακριβή. Επιπλέον, αν θεωρείται ακριβή αυτή τη στιγμή, αναμένουμε να είναι ακόμα πιο ακριβή του χρόνου. Το υλικό (hardware), συμπεριλαμβανομένης της μνήμης, εκτιμάται ότι βελτιώνεται ή γίνεται πιο φτηνό ακολουθώντας τον νόμο του Moore. Στο Σχήμα 1 μπορούμε να παρατηρήσουμε ότι ο ρυθμός της αύξησης των δεδομένων έχει ήδη ξεπεράσει αυτόν του νόμου του Moore. Ως ένα πραγματικό παράδειγμα που το αποδεικνύει αυτό, το Facebook [fbr] ανέφερε ότι μέσα σε έναν χρόνο και λαμβάνοντας δεδομένα με ρυθμό 600 TB ημερισίως, έχει παρατηρήσει τον τριπλασιασμό της ποσότητας της πληροφορίας που αποθηκεύεται στις υποδομές του.

Οι τεχνικές που βασίζονται στην κρυφή μνήμη (cache) μετριάζουν το πρόβλημα καθώς αποθηκεύουν μόνο τα δεδομένα που ζητιούνται συχνά. Παρόλα αυτά, ακόμα και η αποθήκευση ενός μικρού συνόλου δεδομένων της τάξεως μερικών GB δεν λύνει το πρόβλημα. Η αναλυτική επεξεργασία δεδομένων συχνά περιλαμβάνει επαναληπτικές διαδικασίες, όπου διαφορετικά δεδομένα μπορεί να απαιτούνται σε κάθε επανάληψη. Η φόρτωση στη μνήμη διαφορετικών μερών ενός συνόλου δεδομένων μπορεί να επιφέρει σημαντικές καθυστερήσεις λόγω I/O λειτουργιών.

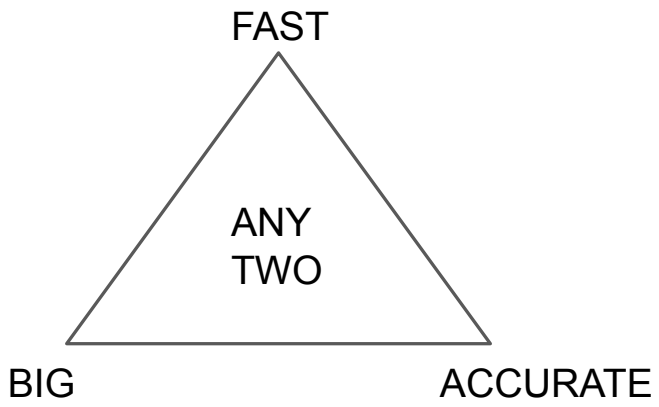
Η έμφαση στη σημασία της μνήμης είναι ακόμα μεγαλύτερη στην περίπτωση της επεξεργασίας ροών δεδομένων. Ένα σημαντικό μέρος της ψηφιακής πληροφορίας που παράγεται στις




Σχήμα 1: Η ραγδαία αύξηση των δεδομένων με βάση την έκθεση του HiPEAC VISION 2015.

μέρες μας εμφανίζεται υπό τη μορφή ροών δεδομένων. Οι εφαρμογές που απαιτούν επεξεργασία μεγάλου όγκου δεδομένων σε πραγματικό χρόνο παρουσιάζουν ενδιαφέρουσες προκλήσεις στα κλασσικά συστήματα επεξεργασίας δεδομένων. Το 2005, ο Stonebraker όρισε τις 8 βασικές απαιτήσεις που πρέπει να ικανοποιεί ένα σύστημα επεξεργασίας ροών δεδομένων [SÇZ05]. Σύμφωνα με την πρώτη απαίτηση, προκειμένου να επιτυγχάνεται χαμηλός χρόνος απόκρισης, ένα σύστημα θα πρέπει να μπορεί να επεξεργαστεί ένα μήνυμα χωρίς να χρειαστεί να αποκτήσει πρόσβαση σε κάποιο "ακριβό" μέσο αποθήκευσης, όπως ο δίσκος. Σε αντίθεση με τις συμβατικές βάσεις δεδομένων, που επιτρέπουν πολλαπλές προσπελάσεις πάνω από στατικά δεδομένα, οι αλγόριθμοι επεξεργασίας ροών δεδομένων συχνά στηρίζονται σε μια μόνο σειριακή προσπέλαση της ροής. Η απαίτηση για επεξεργασία σε πραγματικό χρόνο έχει πυροδοτήσει πληθώρα ερευνητικής δραστηριότητας στην περιοχή. Ορισμένες χαρακτηριστικές εφαρμογές περιλαμβάνουν τα δίκτυα αισθητήρων [CÇC$^+$02, MF02, YG$^+$03], συστήματα παρακολούθησης και ελέγχου σε datacenters [GGRS07], συστήματα που υπολογίζουν στατιστικά με βάση τις τιμές των μετοχών [ZS02] και συστήματα που αναλύουν σε πραγματικό χρόνο logs διαφόρων τύπων συναλλαγών [CFPR00].

Η Προσεγγιστική Επεξεργασία Ερωτημάτων (ΠΕΕ) έχει προκύψει ως μια βιώσιμη εναλλακτική για την διαχείριση του τεράστιου όγκου δεδομένων και των αυστηρών απαιτήσεων στου χρόνους απόκρισης [CGRS01]. Λόγω της *διερευνητικής φύσης* πολλών εφαρμογών επεξεργασίας δεδομένων, δεν αναζητούμε πάντα μια ακριβή απάντηση. Αυτό που ενδιαφέρει περισσότερο στις εφαρμογές αυτές είναι η ανακάλυψη των στατιστικών μοτίβων που υπάρχουν κρυμμένα στα δεδομένα. Ας σκεφτούμε το εξής παράδειγμα: ένας χρήστης θέλει να φιλτράρει ένα σύνολο δεδομένων με βάση διάφορα προκαθορισμένα κριτήρια και να κάνει κάποιους υπολογισμούς χρησιμοποιώντας μόνο μια συγκεκριμένη περιοχή των δεδομένων. Για να το επιτύχει αυτό με αποδοτικό τρόπο, θα υποβάλει στο σύστημα μια ακολουθία ερωτημάτων, όπου τα αρχικά ερωτήματα θα έχουν ως μοναδικό σκοπό τον εντοπισμό της περιοχής ενδιαφέροντος [HHW97]. Σε




Σχήμα 2: Συσχετισμός μεταξύ όγκου δεδομένων, χρόνου απόκρισης και ποιότητας αποτελεσμάτων.

αυτά τα αρχικά ερωτήματα, είμαστε διατεθιμένοι να θυσιάσουμε την ακρίβεια υπερ χαμηλότερων χρόνων απόκρισης. Η ΠΕΕ συσχετίζει την ακρίβεια των αποτελεσμάτων με τον χρόνο εκτέλεσης και την καταναλησκώμενη μνήμη και μας δίνει τη δυνατότητα να ανταλλάσσουμε το ένα με το άλλο. Η σχέση αυτή αποτυπώνεται στο Σχήμα 2. Εάν επιθυμούμε να επιτύχουμε γρηγορότερους χρόνους απόκρισης, θα πρέπει να μειώσουμε είτε το μέγεθος των δεδομένων στη μνήμη είτε να θυσιάσουμε την ακρίβεια του αποτελέσματος. Αν το κρίσιμο ζητούμενο είναι ακριβή αποτελέσματα, τότε κάποιος συμβιβασμός θα πρέπει να γίνει για τον χρόνο εκτέλεσης και το μέγεθος της μνήμης. Οπτικοποιώντας αυτούς τους συσχετισμούς, οι αναλυτές μπορούν να ρυθμίσουν και να βελτιστοποιήσουν την εκτέλεση των ερωτημάτων τους [TK15]. Επίσης, οι προσεγγιστικές απαντήσεις που προέρχονται από κατάλληλα κατασκευασμένες *συνόψεις* μπορεί να είναι η μοναδική εναλλακτική όταν ένα σύνολο είναι αποθηκευμένο κάπου απομακρυσμένα ή δεν είναι διαθέσιμο [AFTU97].

Με βάση αυτή τη λογική, στο παρελθόν έχουν αναπτυχθεί πολλές τεχνικές προσεγγιστικής επεξεργασίας συμπεριλαμβανομένων των: δειγματοληψίας [AMP$^+$13, GM98, AGPR99b], ιστογραμμάτων [IP99, GMP97, JKM$^+$98], sketches [GKMS01, AMS96] και wavelets [CGRS01, GK04, KM05, KSM07, KM07]. Εκτός από τις πολυπληθείς ερευνητικές προσπάθειες, πολλές εταιρίες κατασκευής συστημάτων επεξεργασίας έχουν επίσης συνειδητοποιήσει την αναγκαιότητα για την ΠΕΕ κι έχουν εισάγει προσεγγιστικές τεχνικές στα προϊόντα τους (π.χ., το Facebook Presto [pre], το Druid της Yahoo [yah], SnappyData [RMW$^+$16], και το Oracle 12C [SZB$^+$16]).

Ένα τυχαίο δείγμα αποτελεί ένα "αντιπροσωπευτικό" υποσύνολο των τιμών ενός συνόλου δεδομένων και κατασκευάζεται μέσω ενός στοχαστικού μηχανισμού. Τα δείγματα είναι εύκολο να κατασκευαστούν και μπορούν να χρησιμοποιηθούν για των υπολογισμό μιας ευρείας γκάμας ερωτημάτων. Χάρην αυτών των χαρακτηριστικών τους, τεχνικές δειγματοληψίας

χρησιμοποιούνται στην πλειοψηφία των συστημάτων προσεγγιστικής επεξεργασίας. Για παράδειγμα, η BlinkDB [AMP+13], η VerdictDB [PMSW18] και το Quickr [KSV+16] κυρίως δουλεύουν με δείγματα.

Τα ιστογράμματα συνοψίζουν ένα σύνολο δεδομένων ομαδοποιώντας τις τιμές σε κλάσεις και για κάθε μια τέτοια κλάση υπολογίζουν ένα σύνολο στατιστικών. Η τεχνικές ιστογραμμάτων έχουν εκτενώς μελετηθεί στη βιβλιογραφία και τα ιστογράμματα έχουν ενσωματωθεί στους βελτισοποιητές ερωτημάτων όλων των εμπορικών σχεσιακών βάσεων δεδομένων [mss, mar, ora].

Τα sketches είναι μια κατηγορία μαθηματικών κατασκευασμάτων που ταιριάζει πολύ καλά στο μοντέλο των ροών δεδομένων. Μερικά από τα χαρακτηριστικά τους είναι ότι είναι εύκολα παραλληλοποιήσιμα και συνθέσιμα. Επίσης μπορούν να διαχειριστούν περιπτώσεις όπου το σύστημα αντιμετωπίζει όχι μόνο εισαγωγή αλλά και διαγραφές δεδομένων. Λόγω αυτών των ιδιοτήτων τους και τα sketcehs έχουν υιοθετηθεί από διάφορα εμπορικά συστήματα. Το SnappyData [RMW+16] χρησιμοποιεί Count-Min sketches [CM05] για να υπολογίσει top-k ερωτήματα σε μια ροή δεδομένων. Το Yahoo Druid [yah] χρησιμοποιεί sketches για τον υπολογισμό COUNT DISTINCT ερωτημάτων κι επίσης αυτή τη στιγμή γίνονται προσπάθειες για την ενσωμάτωση sketching τεχνικών στο Apache Flink [fli]. Παρότι συνήθως χρησιμοποιούνται σε περιπτώσεις ροών δεδομένων, τα sketches έχουν επίσης χρησιμοποιηθεί με επιτυχία και σε περιπτώσεις όπου τα δεδομένα είναι στατικά. Για παράδειγμα, το Apache Hive προσφέρει μια ποικιλία sketching αλγορίθμων [yah].

Ο μετασχηματισμός wavelet [SDS96] αποτελεί ένα πολύ αποτελεσματικό εργαλείο για την συμπίεση των δεδομένων, με εφαρμογές στην εξόρυξη δεδομένων [LLZO02], στην εκτίμηση της επιλεκτικότητας των ερωτημάτων (selectivity estimation) [MVW98], στην προσεγγιστική επεξεργασία σχεσιακών πινάκων [CGRS01, VW99] καθώς και ροών δεδομένων [GKMS03, CGS06]. Με απλά λόγια, για να κατασκευάσουμε μια wavelet σύνοψη, εφαρμόζουμε τον μετασχηματισμό wavelet σε ένα σύνολο δεδομένων και στη συνέχεια επιλέγουμε ένα υποσύνολο από τους παραγόμενους *συντελεστές wavelet*.

Παρόλη την εκτενή βιβλιογραφία στην περιοχή, οι τεχνικές που βασίζονται σε wavelets έχουν χρησιμοποιηθεί ελάχιστα στην πράξη για σκοπούς προσεγγιστικής επεξεργασίας ερωτημάτων. Wavelet τεχνικές συναντώνται μόνο σε ορισμένα ακαδημαϊκά, ερευνητικά συστήματα, ενώ απ' όσο γνωρίζω δεν χρησιμοποιούνται σε κανένα εμπορικό προϊόν. Σε αυτή την διατριβή αναλύονται τα προβλήματα που αντιμετωπίζουν οι υπάρχουσες τεχνικές για την κατασκευή συνόψεων μέσω wavelets, και προτείνοται καινούριοι αλγόριθμοι που εκμεταλλεύονται τα χαρακτηριστικά των wavelets και τα καθιστούν αποδοτικά στην εποχή των Μεγάλων Δεδομένων. Καθώς διαφορετικά σενάρια επεξεργασίας παρουσιάζουν διαφορετικές ανάγκες, η διατριβή αυτή προτείνει αλγορίθμους τόσο για την περίπτωση των στατικών όσο και των ροών δεδομένων.

## 0.2 Wavelet Συνόψεις σε Στατικά Δεδομένα

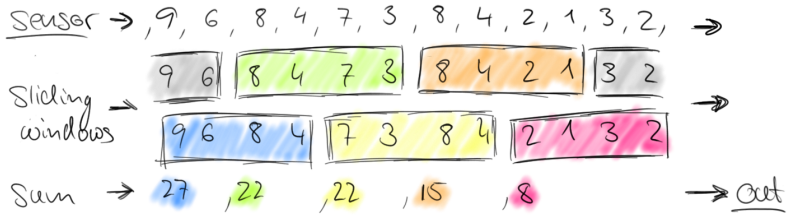
Ο μετασχηματισμός wavelet ενός διανύσματος $A$ είναι μια αναπαράσταση ίσου μεγέθους με τον αρχικό πίνακα. *Wavelet thresholding* καλείται το πρόβλημα του καθορισμού των συντελεστών που πρέπει να κρατήσουμε στην σύνοψη, δοθέντος ενός περιορισμού στον διαθέσιμο χώρο μνήμης. Μια συμβατική προσέγγιση στο θέμα αποτελεί ένας γραμμικός αλγόριθμος που ελαχιστοποιεί το μέσο τετραγωνικό σφάλμα [SDS96]. Οι συνόψεις όμως που κατασκευάζονται με αυτή τη μέθοδο παρουσιάζουν σημαντικά μειονεκτήματα [GK04], όπως υψηλή διακύμανση στην ποιότητα της προσέγγισης, προτίμηση για πιο ακριβή κάλυψη σε συγκεκριμένες περιοχές των δεδομένων και έλλειψη κατανοητών εγγυήσεων για μεμονωμένα ερωτήματα. Από την άλλη μεριά, οι συνόψεις που ελαχιστοποιούν μετρικές μεγίστων σφαλμάτων έχουν αποδειχτεί πιο αξιόπιστες για την ακριβή ανακατασκευή ενός συνόλου δεδομένων [GG02, GK04].

Ωστόσο, οι υπάρχοντες αλγόριθμοι που ελαχιστοποιούν μη-Ευκλείδια σφάλματα σε σημειακά ερωτήματα είναι αυστηρά κεντρικοί και συχνά βασίζονται σε τεχνικές δυναμικού προγραμματισμού, οι οποίες απαιτούν αρκετή μνήμη και υπολογιστική ισχύ. Το ίδιο ισχύει και για αλγορίθμους που βελτιστοποιούν πιο σύνθετα ερωτήματα, όπως τα ιεραρχικά ερωτήματα σε διαστήματα τιμών [GPS08]. Παρουσιάζοντας υπερ-τετραγωνική πολυπλοκότητα, οι αλγόριθμοι αυτοί αποτυγχάνουν να επιτύχουν κλιμάκωση σε μεγάλα σύνολα δεδομένων.

Ο GreedyAbs [KM05] είναι ένας ευριστικός αλγόριθμος που έχει προταθεί για την αντιμετώπιση των προαναφερθέντων προβλημάτων. Ο αλγόριθμος αυτός είναι πιο αποδοτικός από τους αλγορίθμους δυναμικού προγραμματισμού αλλά για να το επιτύχει αυτό πληρώνει κάποιο κόστος στην ακρίβεια των αποτελεσμάτων. Παρόλα αυτά, ούτε αυτός μπορεί να πετύχει κλιμάκωση σε Μεγάλα Δεδομένα καθώς ακολουθεί ένα σειρακό τρόπο εκτέλεσης.

Ο λόγος που οι παράλληλοι αλγόριθμοι είναι ιδιαίτερα σημαντικοί στην εποχή των Μεγάλων Δεδομένων βρίσκεται πίσω από την αρχιτεκτονική των υπαρχόντων συστημάτων επεξεργασίας. Η αναλυτική επεξεργασία δεδομένων συνήθως λαμβάνει χώρα σε κατανεμημένες πλατφόρμες όπως είναι τα Apache Hadoop και Spark. Τα συστήματα αυτά μπορούν να στείλουν πολλές εργασίες ταυτόχρονα σε διαφορετικούς υπολογιστές κι έτσι επιτρέπουν εγγενώς την παράλληλη εκτέλεση. Επίσης, με τους επιταχυντές σε επίπεδο υλικού να γίνονται ιδιαίτερα δημοφιλείς για εφαρμογές Μηχανικής Μάθησης και Μεγάλων Δεδομένων, υπάρχουν πλέον δυνατότητες για επίτευξη ακόμα μεγαλύτερου βαθμού παραλληλίας [KSH12]. Η υψηλή ζήτηση για επιταχυντές σε επίπεδο υλικού, έχει οδηγήσει τους παρόχους υπολογιστικών νεφελωμάτων να συμπεριλάβουν τέτοιες ειδικές συσκευές στις προσφορές τους (π.χ., Amazon's EC2 Elastic GPUs [amaa], FPGA instances [amab]).

Εκτός από την εγγενή δυσκολία για κλιμάκωση, ένα άλλο πρόβλημα που αντιμετωπίζουν πολλές τεχνικές που είναι βασισμένες σε wavelets είναι ότι μπορούν να διαχειριστούν μόνο μονοδιάστατα δεδομένα. Στην περίπτωση πολλαπλών διαστάσεων οι αλγόριθμοι που υπάρχουν




Σχήμα 3: Παράδειγμα ροής κυλυομένου παραθύρου.

είναι τόσο ακριβοί που καθίστανται απαγορευτικοί για Μεγάλα Δεδομένα. Ωστόσο, η ύπαρξη πολυδιάστατων δεδομένων είναι ένα συχνό φαινόμενο στις σύγχρονες, πραγματικές εφαρμογές και η αδυναμία διαχείρισής του αποτελεί ένα βασικό εμπόδιο για την υιοθέτηση των wavelets.

Προκειμένου να προτείνει λύση στους παραπάνω περιορισμούς, η διατριβή αυτή εισάγει παράλληλους αλγορίθμους που κατασκευάζουν συνόψεις για μονοδιάστατα αλλά και για πολυδιάστατα δεδομένα. Οι προτεινόμενοι αλγόριθμοι υλοποιούνται κι αξιολογούνται στην κατανεμημένη πλατφόρμα επεξεργασίας Apache Hadoop.

## 0.3 Ροές Δεδομένων

Στις μέρες μας, οι ροές δεδομένων αποτελούν ένα ιδιαίτερα σημαντικό μέρος των συστημάτων επεξεργασίας δεδομένων. Ένας βασικός περιορισμός στους αλγορίθμους ροών είναι η απαίτηση να κάνουν μια μοναδική προσπέλαση πάνω στα δεδομένα. Για να μπορούν να απαντάνε σε διάφορα ερωτήματα χωρίς να παραβιάζουν αυτόν τον περιορισμό, οι αλγόριθμοι αυτοί συχνά βασίζονται στην κατασκευή συνόψεων. Συνήθως οι συνόψεις αυτές καταλαμβάνουν πολύ μικρό χώρο και πρέπει να μπορούν να ενημερωθούν και να ερωτηθούν σε πραγματικό χρόνο (υπο-γραμμικό στο μεγεθος της εισόδου).

Επιπλέον, καθώς για πολλές εφαρμογές τα πιο πρόσφατα δεδομένα έχουν μεγαλύτερη αξία, αποκτούν επίσης και μεγαλύτερη προτεραιότητα κατά την επεξεργασία. Τα στατιστικά που ανακτούμε από συνόψεις για τα πιο πρόσφατα δεδομένα πρέπει να είναι υπολογισμένα με μεγαλύτερη ακρίβεια από αυτά που υπολογίζουμε για παλαιότερες τιμές. Για τον σκοπό αυτό, πολλά μαθηματικά μοντέλα έχουν προταθεί στη βιβλιογραφία [CS03]. Λαμβάνοντας υπόψιν μόνο τις $W$ τελευταίες τιμές που έχει δει, το μοντέλο του *κυλυομένου παραθύρου* [DGIM02] είναι ένα από τα πιο κατανοητά και εύκολα ερμηνεύσιμα. Το Σχήμα 3[1] αναπαριστά ένα παράδειγμα όπου υπολογίζεται το άθροισμα των τιμών του παραθύρου. Οι μετρήσεις του αισθητήρα στην πάνω γραμμή του σχήματος επεξεργάζονται από ένα κυλυόμενο παράθυρο μεγέθους τέσσερα και βήματος δύο. Ο αλγόριθμος βγάζει ως έξοδο το άθροισμα των αντίστοιχων τιμών για κάθε παράθυρο.

[1]Πηγή εικόνας: https://flink.apache.org/news/2015/12/04/Introducing-windows.html

Όπως είναι αναμενόμενο, μια εφαρμογή δεν μπορεί να αποθηκεύσει εξ' ολοκλήρου μια άπειρη ακολουθία τιμών. Ως εκ τούτου, τα *παράθυρα* είναι μια θεμελιώδης έννοια στην θεωρία της επεξεργασίας ροών. Σε πολλές περιπτώσεις μάλιστα, μπορεί να μην είναι δυνατή η αποθήκευση στη μνήμη ακόμα και για τα στοιχεία ενός παραθύρου. Για παράδειγμα, τα ενσωματωμένα συστήματα που συναντάμε συχνά σε IoT εφαρμογές μπορεί να έχουν μνήμη χωριτικότητας μερικών MB. Ο υπολογισμός ενός μέσου όρου σε μια ροή δεδομένων που αποτελείται απο αριθμούς μήκους 8 bytes ο καθένας, χρειάζεται 800 MB μνήμης (θεωρώντας ένα παράθυρο $W = 100M$ στοιχείων) τα οποία μπορεί να μην είναι διαθέσιμα. Επομένως συχνά καταφεύγουμε σε προσεγγιστικές τεχνικές που κατασκευάζουν μια σύνοψη του παραθύρου.

Πολλοί αλγόριθμοι έχουν προταθεί για τον υπολογισμό διαφόρων στατιστικών χρησιμοποιώντας το μοντέλο του κυλυομένου παραθύρου. Η πλειοψηφία των αλγορίθμων αυτών πετυχαίνει πολυ-λογαριθμική πολυπλοκότητα στο μέγεθος του παραθύρου τόσο ως προς τον χρόνο όσο και τον χώρο [DGIM02, GT02, QAEA03, XTB08]. Ωστόσο, το πρόβλημα δεν έχει μελετηθεί επαρκώς στην περίπτωση των wavelets. Η διατριβή αυτή διερευνά την ικανότητα των wavelets να προσεγγίσουν βασικά στατιστικά μεγέθη (COUNT, SUM, AVG) σε ροές δεδομένων κυλυομένου παραθύρου.

## 0.4 Συνεισφορά της Διατριβής

Η πρώτη συνεισφορά αυτής της διατριβής είναι η παραλληλοποίηση αλγορίθμων για την κατασκευή wavelet συνόψεων. Στο Κεφάλαιο 3 παρουσιάζονται αλγόριθμοι για την παραλληλοποίηση μονοδιάστατων δεδομένων. Πιο συγκεκριμένα, παρουσιάζεται μια τεχνική για την παραλληλοποίηση και κλιμάκωση όλων των αλγορίθμων δυναμικού προγραμματισμού που υπάρχουν για το πρόβλημα. Η προτεινόμενη προσέγγιση βασίζεται σ' ένα σχήμα διαμοιρασμού των δεδομένων που επιτρέπει την παράλληλη επεξεργασία των γραμμών του πίνακα του δυναμικού προγραμματισμού. Για να δούμε τα αποτελέσματα της τεχνικής αυτής στην πράξη, την εφαρμόζουμε σε έναν πολύ γνωστό αλγόριθμο [KSM07] και δημιουργούμε μια παράλληλη εκδοχή του με πολύ καλύτερες ιδιότητες κλιμακωσιμότητας.

Καθώς όμως οι αλγόριθμοι δυναμικού προγραμματισμού είναι ακριβοί σε χρόνο και πόρους, στην διατριβή αυτή προτείνονται επίσης δυο ευριστικοί αλγόριθμοι που βελτιώνουν τον χρόνο εκτέλεσης με κάποιο κόστος στην ποιότητα των προσεγγιστικών αποτελεσμάτων. Οι αλγόριθμοι αυτοί βασίζονται σε τρεις βασικές ιδέες: (i) ιεραρχικό διαμοιρασμό της δομής του wavelet μετασχηματισμού, (ii) πολλαπλές εκτελέσεις του κεντρικού αλγορίθμου, και (iii) συγχώνευση και φιλτράρισμα των ενδιάμεσων αποτελεσμάτων. Ο πρώτος ευριστικός αλγόριθμος που προτείνεται στο Κεφάλαιο 3 χρειάζεται πολλές κατανεμημένες εκτελέσεις για να κατασκευάσει τη σύνοψη. Για την περαιτέρω βελτίωση του χρόνου εκτέλεσης, προτείνεται ένας ακόμα αλγόριθμος, ο οποίος απαιτεί μόνο μια κατανεμημένη εκτέλεση. Τα πειράματα που διεξήχθησαν

δείχνουν ότι η επίδοση που επιτυγχάνουν και οι δύο ευριστικοί αλγόριθμοι είναι καλύτερη από τους αντίστοιχους δυναμικού προγραμματισμού. Επιπλέον δεν παρουσιάζουν κάποια έκτπωση στην ποιότητα του αποτελέσματος σε σχέση με τον κεντρικό ευριστικό GreedyAbs.

Όλοι οι αλγόριθμοι υλοποιήθηκαν στο Apache Hadoop και πραγματοποιήθηκε εκτενής πειραματική αξιολόγηση χρησιμοποιώντας τόσο συνθετικά όσο και πραγματικά δεδομένα. Για την αξιολόγηση αλγορίθμων πάνω στο συγκεκριμένο πρόβλημα, όλες οι προηγούμενες ερευνητικές δουλειές έχουν χρησιμοποιήσει σύνολα δεδομένα που περιέχουν μέχρι 262K τιμές. Για να δείξουμε τις ιδιότητες κλιμακωσιμότητας των προτεινόμενων αλγορίθμων, τα πειράματα που διεξήχθησαν χρησιμοποιούν σύνολα δεδομένων που είναι μεγαλύτερα μέχρι και τρεις τάξεις μεγέθους.

Στο Κεφάλαιο 4 οι προτεινόμενοι αλγόριθμοι για την κατασκευή συνόψεων επεκτείνονται σε σύνολα δεδομένων πολλαπλών διαστάσεων. Πρώτα παρουσιάζεται ένας κεντρικός αλγόριθμος δυναμικού προγραμματισμού που είναι ιδιαίτερα αποδοτικός και βασίζεται στον IndirectHaar [KSM07]. Στη συνέχεια, δείχνουμε ότι η τεχνική που χρησιμοποιούμε για την παραλληλοποίηση των αλγορίθμων δυναμικού προγραμματισμού σε μονοδιάστατα δεδομένα μπορεί να εφαρμοστεί με μικρές παραλλαγές και σε δεδομένα πολλών διαστάσεων. Το ίδιο ισχύει και για τους ευριστικούς αλγορίθμους του Κεφαλαίου 3. Δείχνω πώς μπορούν να επεκταθούν ώστε να διαχειρίζονται πολυδιάστατα δεδομένα και παρουσιάζω μια θεωρητική ανάλυση για τον χρόνο εκτέλεσής τους.

Στο Κεφάλαιο 5 εξετάζεται η αποδοτικότητα των wavelets σε περιπτώσεις ροών δεδομένων κυλυομένου παραθύρου. Παρότι λαμβάνονται υπόψιν και σημειακά ερωτήματα αλλά και ερωτήματα σε διαστήματα τιμών, δίνεται ιδιαίτερη έμφαση στην περίπτωση συναθροιστικών ερωτημάτων όπως COUNT, SUM και AVG. Αυτά είναι τα πιο χαρακτηριστικά ερωτήματα σε κυλυόμενα παράθυρα και η επίδοση των wavelets στην περίπτωση αυτή δεν έχει ερευνηθεί επαρκώς στο παρελθόν.

Πιο συγκεκριμένα, θεωρώ ένα περιβάλλον κυλυομένου παραθύρου και παρουσιάζω καινούριους wavelet αλγορίθμους για τον υπολογισμό ερωτημάτων πάνω σε διαστήματα τιμών. Η πολυπλοκότητα των αλγορίθμων αυτών αναλύεται θεωρητικά, όπως επίσης παρέχονται και αιτοκρατικές εγγυήσεις για τα σφάλματα των προσεγγίσεων. Οι προτεινόμενοι αλγόριθμοι εφαρμόζονται επίσης σε ένα κατανεμημένο περιβάλλον όπου πολλαπλές ροές δεδομένων υπολογίζουν ατομικά, η καθεμιά τη σύνοψή της και υπάρχει ένας μοναδικός κόμβος-συντονιστής που συχγωνεύει τις συνόψεις σε πραγματικό χρόνο και υπολογίζει απαντήσεις ερωτημάτων πάνω στην ένωση των ροών. Η πειραματική αξιολόγηση σε συνθετικά και πραγματικά δεδομένα δείχνει ότι για μια πληθώρα περιπτώσεων, οι αλγόριθμοι που παρουσιάζονται σε αυτήν τη διατριβή υπερτερούν σε ακρίβεια σε σχέση με άλλες γνωστές τεχνικές όπως τα εκθετικά ιστογράμματα. Επιπλέον, στο Κεφάλαιο 5 παρουσιάζεται κι ένα σύστημα το οποίο βελτιώνει

περαιτέρω την ακρίβεια των αποτελεσμάτων δοθέντων επιπλέον πληροφοριών σχετικά με τον φόρτο των ερωτημάτων.

CHAPTER 1

# Introduction

This dissertation introduces efficient wavelet-based algorithms that enable approximate query processing (AQP) over big data. In the following, the rationale behind this work is presented. Why AQP, why wavelets and what cases are covered in this thesis are some of the questions I try to answer in this introductory Chapter.

## 1.1 Motivation

The technological and societal developments of our era have resulted in an unprecedented production and processing of enormous data volumes, referred to with the term 'Big Data' [big]. Businesses, government organizations and digital infrastructures alike contribute to this Big Data reality. This abundance of datasets has, in turn, given rise to data-driven approaches in both academia and industry.

Following the OLAP paradigm [CD97], modern data analytics applications involve computing aggregates over a large number of records and along a variety of different dimensions. For instance, a financial trading firm needs to compare the prices of securities to historical averages of various granularities in order to determine items that are under- or over-valued. Computing such averages has been traditionally accomplished via sequential scans of large fractions of a database. Yet, there exist cases where existing data processing tools have become the bottleneck and such scan operations can become extremely expensive. When huge heterogeneous data is the case, even the fastest database systems can take hours or even days to answer the simplest of

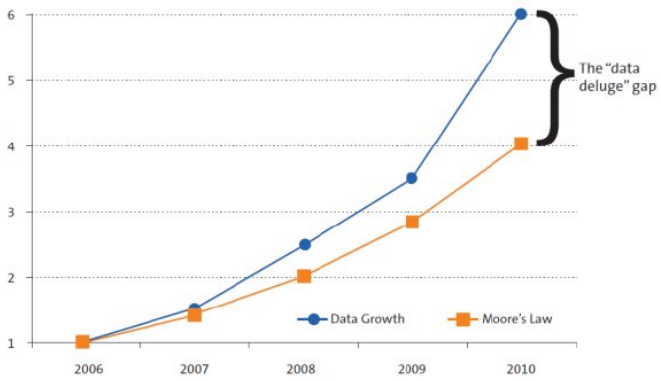



Figure 1.1: The "Data Deluge" gap according to the HiPEAC VISION 2015 report.

queries. Computing a simple average over 10 terabytes of data stored on 100 machines can take in the order of 30 - 45 minutes on Hadoop if the data is striped on disks, and up to 5 - 10 minutes even if the entire data is cached in memory [AMP+13].

As data-driven discovery is often an *interactive and iterative process* [Moz15], such response times are unacceptable to most users and applications. In another example, data analysts in a retail enterprise slice and dice their sales data to understand the sales performance along different dimensions (such as product and geographic location) using a varying set of filtering conditions. Interactive query response time is critical in such data exploration; human analysts should be able to rapidly iterate between hypotheses and evidence. Studies in human-computer interaction show that the analyst typically loses the analysis context if the response time is above one second [Shn84]. In order to meet the demands of interactive, human-in-the-loop data analytics, both commercial and open source systems continuously strive to provide lower response times through various techniques such as parallelism, indexing, materialization and query optimization.

Traditionally, most of the aforementioned approaches try to better utilize available memory. However, keeping all useful data in main memory may not be an affordable or realistic option in the Big Data era. Buying memory that is big enough to hold the entire dataset does not consist an affordable option. Furthermore, if it is too expensive now, it will be even more expensive to do so next year. Hardware, including memory, is expected to improve or get cheaper according to Moore's law. In Figure 1.1, we can see that the data growth has already surpassed that rate. In a real-world testimony, Facebook reported [fbr] that within a year, it has seen a 3× growth in the amount of stored data, with an incoming daily rate of 600 TB.

Caching techniques mitigate the problem, as they only store *hot* data in memory. Nevertheless, even caching only a working-set of some GB does not do the trick. Analytics usually include

iterative processes, where different data may be of interest at each iteration. Loading different parts of the dataset each time incurs significant I/O delays that may be not acceptable.

Memory is even more stressed in the case of stream processing. A significant part of the digital information currently produced comes in the form of data streams, i.e., continuous sequences of items. Applications that require real-time processing of high-volume data steams are pushing the limits of traditional data processing infrastructures. In [SÇZ05], Stonebraker et al. have defined the 8 requirements for stream processing systems. According to the first requirement, in order to achieve low latency, a system must be able to perform message processing without having a costly storage operation in the critical processing path. Unlike conventional database query processing that allows several passes over static data, data-stream processing algorithms often rely on a single pass over the stream. The requirement of real-time processing of continuous data in high-volumes has triggered a flurry of research activity in the area. Some typical applications include sensor networks [CÇC$^+$02, MF02, YG$^+$03], datacenter monitoring [GGRS07], financial data trackers [ZS02], and real-time analysis of various transaction logs [CFPR00].

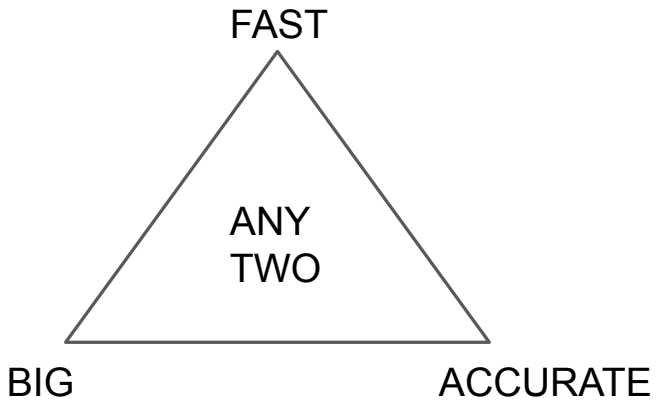



Figure 1.2: Trade-off among data volume, query response time and accuracy in the results.

Approximate Query Processing has emerged as a viable alternative for dealing with the huge amount of data and the increasingly stringent response-time requirements [CGRS01]. Due to the *exploratory nature* of many data analytics applications, there exists a number of scenarios in which an exact answer is not required; we are interested in discovering statistical patterns rather than obtain answers precise to the last decimal. For example, during a drill-down query sequence in ad-hoc data mining, initial queries in the sequence frequently have the sole purpose of determining the truly interesting query regions of the dataset [HHW97]. Thus, in these initial queries, we are willing to forgo accuracy in favor of better response-times. AQP provides a trade-off between accuracy, execution-time and memory consumption. This trade-off is depicted in Figure 1.2: If we wish to achieve fast responses, we either reduce the size of the data or the accuracy of the result. If we want accurate results, we need to compromise on time or data

volumes. By visualizing the trade-off, analysts can fine-tune the execution of queries [TK15]. Moreover, approximate answers obtained from appropriate *synopses* of the data may be the only option when the base data is remote or unavailable [AFTU97].

To that end, several approximation techniques have been developed, including: sampling [AMP+13, GM98, AGPR99b], histograms [IP99, GMP97, JKM+98], sketches [GKMS01, AMS96] and wavelets [CGRS01, GK04, KM05, KSM07, KM07]. Apart from the numerous research efforts, various industrial vendors have realized the necessity for AQP and have added approximation features in their products (e.g., Facebook's Presto [pre], Yahoo's Druid [yah], SnappyData [RMW+16], and Oracle 12C [SZB+16]).

A random sample comprises a "representative" subset of the data values of interest, obtained via a stochastic mechanism. Samples are usually fast to obtain, and can be used to approximately answer a wide range of queries. Due to their efficient computation and wide applicability, sampling techniques are employed by the majority of AQP systems. To name a few, BlinkDB [AMP+13], VerdictDB [PMSW18], Quickr [KSV+16] mostly work with samples.

A histogram summarizes a dataset by grouping the data values into subsets, or "buckets", and then, for each bucket, computes a small set of summary statistics. Histograms have been extensively studied and have been incorporated into the query optimizers of virtually all commercial relational DBMSs [mss, mar, ora].

Sketch summaries are particularly well suited to streaming data, they are massively parallelizable and easily composable. They can accommodate streams of transactions in which data is both inserted and removed. Sketches have also been successfully used to estimate the answer in COUNT DISTINCT queries, a notoriously hard problem in stream processing. Due to their nice properties, sketches have been incorporated in industrial streaming systems. For example, SnappyData [RMW+16] uses Count-Min sketches [CM05] to compute top-k queries on streams. Yahoo Druid [yah] uses sketches for computing COUNT DISTINCT and quantile queries. There is also an active effort to integrate sketches in the Apache Flink [1] system [fli]. While they particularly fit in the streaming case, sketches have also been used in batch processing systems too. For example, Apache Hive [2] offers a variety of sketching algorithms as built-in functions [yah].

Wavelet decomposition [SDS96] provides a very effective data reduction tool, with applications in data mining [LLZO02], selectivity estimation [MVW98], approximate and aggregate query processing of massive relational tables [CGRS01, VW99] and data streams [GKMS03, CGS06]. In simple terms, a wavelet synopsis is extracted by applying the wavelet decomposition on an input collection (considered as a sequence of values) and then summarizing it by retaining only a subset of the produced *wavelet coefficients*. The original data can be approximately reconstructed

[1] https://flink.apache.org/
[2] https://hive.apache.org/

based on this compact synopsis. Previous research has established that reliable and efficient approximate query processing can then be performed solely over such concise wavelet synopses [CGRS01].

However, wavelet-based techniques have hardly been adopted in practice for AQP purposes. There are only a few academic prototypes (e.g., [SJBK08, MP04]) that approximate aggregates based on wavelets and, to the best of my knowledge, no industrial product. In this dissertation, I discuss the shortcomings of existing wavelet-based synopsis construction algorithms and propose new ones that unleash the power of wavelets and render them "affordable" in the Big Data era. As different scenarios have different demands, this thesis covers the cases of both static and streaming data.

## 1.2 Wavelet Synopses Over Static Data

The wavelet decomposition of a data vector $A$ is a representation of equal size as the original array. *Wavelet thresholding* is the problem of determining the coefficients to be retained in the synopsis given an available space budget $B$. A conventional approach to this problem features a linear-time deterministic thresholding scheme that minimizes the overall mean squared error [SDS96]. Still, the synopses produced by this method exhibit significant drawbacks [GK04], such as the high variance in the quality of data approximation, the tendency for severe bias in favor of certain regions of the data and the lack of comprehensible error guarantees for individual approximate answers. On the other hand, synopses that minimize maximum error metrics on individual data values prove more robust in accurate data reconstruction [GG02, GK04].

However, the existing algorithms that minimize non-Euclidean error metrics in point queries are strictly centralized and are usually based on dynamic programming (DP) approaches that demand a lot of memory and processing power. The same also holds for algorithms that optimize more complex queries such as hierarchical range queries [GPS08]. Featuring super-quadratic complexities, all these algorithms fail to scale to big datasets.

In [KM05], GreedyAbs, a heuristic solution is proposed. This algorithm is more time-efficient than the DP algorithms but at the cost of loosened quality guarantees. Yet, it cannot scale to Big Data either, as it follows a sequential path of execution that prevents a data-parallel approach.

The reason why data-parallel algorithms are of paramount importance lies behind the architecture of the available data processing systems. Modern analytics usually take place in distributed, scale-out platforms such as Apache Hadoop [3] and Spark [4]. These systems can dispatch multiple concurrent tasks across the workers of a cluster and we need to take advantage of that capability. Furthermore, with hardware accelerators becoming extremely popular for Machine

[3] https://hadoop.apache.org/
[4] https://spark.apache.org/

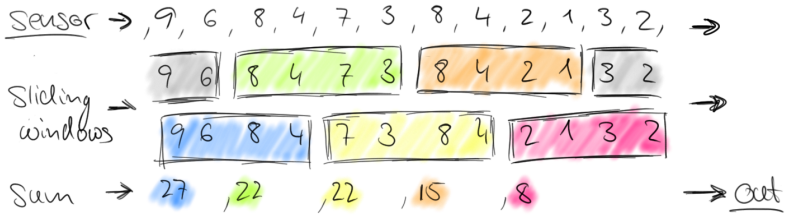



Figure 1.3: Example of a sliding-window stream.

Learning and Big Data Analytics workloads, there are massive capabilities for parallel execution [KSH12]. The high demand for hardware accelerators has led cloud vendors to include specialized devices in their offerings, alongside general purpose CPUs (e.g., Amazon's EC2 Elastic GPUs [amaa] or FPGA instances [amab]).

Apart from their inherent difficulty in scaling-out, another shortcoming of existing wavelet techniques is that most of them handle strictly one-dimensional data and come at a prohibitive complexity when more dimensions are involved. Nevertheless, multidimensional datasets are a common case in real-world applications and such a limitation makes the use of wavelets impractical.

To address all the aforementioned limitations, this dissertation introduces parallel algorithms that construct wavelet synopses for both one- and multi-dimensional data. The proposed algorithms are implemented and evaluated on top of the Apache Hadoop processing framework.

## 1.3 The Streaming Case

Nowadays, streams are a first class citizen in data processing infrastructures. Streaming algorithms are generally restricted to allow only a single pass over the data. In order to achieve this, they often rely on building real-time, concise synopses of the underlying streams. These synopses typically need small space, update and query time (sub-linear to the input size) and can be used to provide approximate, yet accurate answers.

Furthermore, as for most applications there is more value in real-time information, recent data tend to be prioritized; statistics in fresh data items should be represented with higher precision than in older ones. For this purpose, various time-decay models have been proposed in the literature [CS03]. The *sliding-window* model [DGIM02] is one of the most intuitive ones as it only considers the most recent data items seen so far. Figure 1.3[5] illustrates an example of a sliding-window stream, where a SUM aggregation takes place. The sensor measurements of the topmost row are processed in a sliding window of length four and slide-step two. The algorithm outputs the sum of the corresponding elements for each window.

[5]image source: https://flink.apache.org/news/2015/12/04/Introducing-windows.html

Windows are a central concept in stream processing because an application cannot store an infinite stream in its entirety. Nevertheless, in many cases, even storing only a window may not be an option. For example, embedded devices, that are often met in IoT scenarios, have a memory capacity of only a few MB. Keeping track of the average value in a stream of 8 byte long numbers and a window size of $W = 100M$ data elements requires 800 MB of RAM which may not be available. Therefore, approximation techniques that summarize a sliding-window should be employed.

Several algorithms have already been proposed for maintaining different types of statistics over sliding-windows while requiring time and space poly-logarithmic to the window size [DGIM02, GT02, QAEA03, XTB08]. However, the problem has not attracted much attention when using wavelets. In this dissertation, I investigate the capacity of wavelets to efficiently approximate basic aggregates over a data stream under the sliding-window model. I focus on COUNT, SUM and AVG queries, since more complex queries in sliding-windows usually need to compute such basic aggregates under the hood [PGD12].

## 1.4 Contributions

The first contribution of this thesis is the parallelization of wavelet thresholding algorithms over large-scale, static, one-dimensional data (Chapter 3). More specifically, a general theoretical framework is presented for scaling-out the existing DP algorithms for the problem. The proposed approach is based on a novel partitioning scheme that allows for the parallel processing of DP table rows. In order to demonstrate the benefits of this framework, it is applied on the state-of-the-art DP algorithm [KSM07] and a new parallel algorithm with much better scalability properties is produced.

However, as DP algorithms are quite costly, two heuristic-based algorithms that improve on the running-time at the cost of loosened error guarantees are also proposed. The first heuristic algorithm follows three key-ideas: 1) hierarchical partitioning of the wavelet structure, 2) multiple executions of the centralized algorithm, and 3) merging and filtering of intermediate results. Nevertheless, as it initiates multiple distributed jobs, I further improve on its running-time and also propose a second heuristic algorithm that requires only a single job for constructing the synopsis. The conducted experiments show that the achieved performance of both distributed greedy algorithms exhibits no quality degradation compared to their centralized counterpart.

All algorithms are implemented on top of the Hadoop processing framework and an extensive experimental evaluation is performed using both synthetic and real datasets. Previous approaches to the problem used datasets of up to 262K datapoints. To put emphasis on the scalability properties of this work, experiments with datasets larger by three orders of magnitude are conducted.

After analyzing the problem for the one-dimensional case, the proposed ideas are extended to also handle datasets of multiple dimensions (Chapter 4). First, a new centralized, multidimensional algorithm based on IndirectHaar [KSM07] is presented. Then, it is shown that the proposed framework for parallelizing DP algorithms can be applied in that case too. The proposed heuristic algorithms for one-dimensional data are also extended and a complete theoretical analysis for the multidimensional case is provided.

In Chapter 5, I investigate the efficiency of wavelets for summarizing a sliding-window stream. While workloads of both point and range queries are considered, particular emphasis is put on basic aggregates such as COUNT, SUM and AVG. This is the most common query type in the sliding-window context and the performance of streaming wavelets in such queries has not been studied before.

Specifically, new wavelet-based algorithms are presented for answering range queries over a single stream in the sliding-window model. The complexity of these algorithms is theoretically analyzed and deterministic error guarantees are provided. The proposed approach is also applied and validated in a distributed setup, where multiple streams compute individual synopses and a single coordinator merges them in real-time to produce global answers. The experimental evaluation, in both synthetic and real data, shows that the work of this thesis outperforms in terms of accuracy state-of-the-art techniques such as exponential histograms and deterministic waves for a variety of workloads. Moreover, this dissertation also introduces a system that can further improve on accuracy, given some information about the query workload.

## 1.5 Document Outline

The remainder of this document is organized as follows. Chapter 2 provides the mathematical background needed for the understanding of the presented ideas.

Chapter 3 deals with the parallelization of algorithms that construct optimal synopses over one-dimensional data. Section 3.2 presents a theoretical framework for the parallelization of DP algorithms and discusses its application on top of the Hadoop processing platform as well as on top of GPU accelerators. Section 3.3 introduces parallel greedy algorithms for the problem. These algorithms achieve better a running-time at the cost of loosened quality guarantees. A thorough experimental evaluation for wavelet algorithms over static, one-dimensional data is presented in Section 3.5.

Chapter 4 extends the proposed algorithms to multiple dimensions. Both optimal and heuristic algorithms are presented for the construction of synopses over multidimensional datasets. Section 4.2 presents the optimal algorithm, Section 4.3 the heuristic one and in Section 4.4 there is a qualitative discussion. In Section 4.5, the experimental evaluation for the multidimensional case is presented.

While up to this point, all algorithms target batch processing scenarios, Chapter 5 introduces wavelet-based techniques for streams. Section 5.2 describes an algorithm that can efficiently approximate range queries under the sliding-window model. Extra improvements for the case we have workload information are presented in Section 5.6 and an extension to distributed streams can be found in Section 5.4. An experimental evaluation of the proposed streaming algorithms is presented in Section 5.7.

Chapter 6 contains a thorough literature review related to approximate query processing. Some important research works in the domain are listed and classified according to the employed technique. Of course, particular emphasis is put on wavelet-based techniques in order to highlight the contributions of this thesis.

Finally, the contributions of this work are summarized in Chapter 7.

CHAPTER 2

# Mathematical Background

Wavelet analysis is a major mathematical technique for hierarchically decomposing functions in an efficient way. Wavelets are functions which have prescribed smoothness, are well localized in both time and frequency, and form well-behaved bases for many of the important function spaces of mathematical analysis. What makes wavelet bases especially interesting is their *self-similarity*: every function in a wavelet basis is a dilated and translated version of one (or possibly a few) *mother functions*.

The wavelet decomposition of a function consists of a coarse overall approximation together with detail coefficients that influence the function at various scales [SDS96]. All wavelet coefficients are of the form: $\langle A, \phi_{l,k} \rangle$, where $A$ represents the input data. As such, the wavelet decomposition is computationally efficient (linear time) and has excellent energy compaction and decorrelation properties, which can be used to effectively generate compact representations that exploit the structure of data.

## 2.1 One-Dimensional Haar Wavelets

*Haar wavelets* constitute the simplest possible orthogonal wavelet system. Assume a onedimensional data vector $A$ containing $N = 8$ data values $A = [5, 5, 0, 26, 1, 3, 14, 2]$. The Haar wavelet transform of $A$ can be computed as follows: We first average the values in a pairwise fashion to get a new "lower-resolution" representation of the data with the following average values: $[5, 13, 2, 8]$. The average of the first two values (i.e., 5 and 5) is 5, the average of the next two values

Table 2.1: Wavelet decomposition example

| Resolution | Averages | Detail Coef. |
|---|---|---|
| 3 | $[5, 5, 0, 26, 1, 3, 14, 2]$ | – |
| 2 | $[5, 13, 2, 8]$ | $[0, -13, -1, 6]$ |
| 1 | $[9, 5]$ | $[-4, -3]$ |
| 0 | $[7]$ | $[2]$ |

(i.e., 0 and 26) is 13, etc. It is obvious that, during this averaging process, some information has been lost and thus the original data values cannot be restored. To be able to restore the original data array, we need to store some *detail coefficients* that capture the missing information. In Haar wavelets, the detail coefficients are the differences of the (second of the) averaged values from the computed pairwise average. In our example, for the first pair of averaged values, the detail coefficient is 0 (since $5 - 5 = 0$) and for the second is $-13$ ($13 - 26 = -13$). After applying the same process recursively, we generate the full wavelet decomposition that comprises a single overall average followed by three hierarchical levels of 1, 2, and 4 detail coefficients respectively (see Table 2.1). In our example, the wavelet transform (also known as the wavelet decomposition) of A is $W_A = [7, 2, -4, -3, 0, -13, -1, 6]$. Each entry in $W_A$ is called a *wavelet coefficient*. The main advantage of using $W_A$ instead of $A$ is that, for vectors containing similar values, most of the detail coefficients tend to have very small values. Therefore, eliminating such small coefficients from the wavelet transform (i.e., treating them as zeros) introduces only small errors when reconstructing the original array and thus results to a very effective form of lossy data compression.

### 2.1.1 Error-Trees

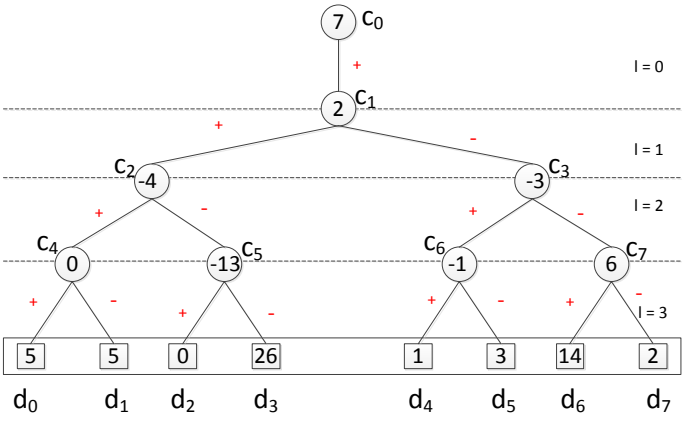



Figure 2.1: An error-tree that illustrates the hierarchical structure of the Haar wavelet decomposition

The *error-tree*, introduced in [MVW98], is a hierarchical structure that illustrates the key properties of the Haar wavelet decomposition. Figure 2.1 depicts the error-tree for our simple example data vector $A$. Each internal node $c_i$ $(i = 0, ..., 7)$ is associated with a wavelet coefficient value, and each leaf $d_i$ $(i = 0, ..., 7)$ is associated with a value in the original data array. Given an error-tree $T$ and an internal node $c_k$ of T, we let $leaves_k$ denote the set of data nodes in the subtree rooted at $c_k$. This notation is extended to $leftleaves_k$ ($rightleaves_k$) for the left (right) subtree of $c_k$. We denote $path_k$ as the set of all nodes with nonzero coefficients in T which lie on the path from a node $c_k$ ($d_k$) to the root of the tree T. Moreover, for any two data nodes $d_l$ and $d_h$, we use $d\,(l : h)$ to denote the range sum $\sum_{i=l}^{h} d_i$.

Given the error-tree representation of a one-dimensional Haar wavelet transform, we can reconstruct any data value $d_i$ using only the nodes that lie on $path_i$. That is

$$d_i = \sum_{c_j \in path_i} \delta_{ij} \cdot c_j, \delta_{ij} = \begin{cases} 1 & d_i \in \text{leftleaves}_j \\ -1 & otherwise \end{cases}$$

For example, in Figure 2.1, value $d_5 = 7 - 2 - 3 - (-1) = 3$. A range sum $d(l : h)$ can be computed using only nodes $c_j \in path_l \cup path_h$, by $d(l : h) = \Sigma_{c_j \in path_l \cup path_h} x_j c_j$, where

$$x_j = \begin{cases} (h - l + 1) & j = 0 \\ \Big( |\text{leftleaves}_{j,l:h}| - |\text{rightleaves}_{j,l:h}| \Big) & otherwise \end{cases} \tag{2.1}$$

Here, $leftleaves_{j,l:h} = leftleaves_j \cap \{d_l, d_{l+1}, .., d_h\}$ and $rightleaves_{j,l:h} = rightleaves_j \cap \{d_l, d_{l+1}, .., d_h\}$. That means that node $c_j$ contributes to the range sum $d\,(h : l)$ positively as many times as there are leaf nodes of the left sub-tree of $c_j$ in the summation range, and negatively as many times as there are leaf nodes of the right sub-tree of $c_j$, while the value of $c_0$ contributes positively for each leaf node in the summation range. In our example, $d\,(3 : 6) = -1 \cdot (-13) + (-1) \cdot (-4) + (-2) \cdot 2 + 4 \cdot 7 + 1 \cdot (-3) + 6 = 44$.

Thus, reconstructing a single data value involves summing at most $logN + 1$ coefficients and reconstructing a range sum involves summing at most $2logN + 1$ coefficients, regardless of the width of the range.

## 2.2 Multidimensional Haar Wavelets

The Haar wavelet decomposition can be extended to multiple dimensions using two distinct methods, namely the *standard* and *nonstandard* decomposition [CGRS01]. Each of these transforms results from a natural generalization of the one-dimensional decomposition. Considering a D-dimensional array $A$ of size $N$, where $N$ is the number of datapoints, the wavelet transform

produces a D-dimensional array $W_A$ of the same shape with $A$. To simplify the exposition to the basic ideas of multidimensional wavelets, we assume all dimensions of the input array to be of equal size.

The work presented in this thesis is based on the nonstandard decomposition. Abstractly, the nonstandard decomposition alternates between dimensions during successive steps of pairwise averaging and differencing: given an ordering for the data dimensions $(1, 2, ..., D)$, we perform one step of pairwise averaging and differencing for each one-dimensional row of array cells along dimension k, $\forall k \in [1, D]$. The results of earlier averaging and differencing steps are treated as data values for larger values of k. One way of conceptualizing this procedure is to think of a $2^D$ hyper-box being shifted across the data array, performing pairwise averaging and differencing. We then gather the average value of each individual $2^D$ hyper-box and we form a new array of lower resolution. The process is then repeated recursively on the new array. An example of this process for a two-dimensional $4 \times 4$ data array is illustrated in Figure 2.2. We demonstrate the process for the lower left quadrant of the array. Initially, we have the values: 1, 4, 9 and 6. By pairwise averaging and differencing along the first dimension we get: $(1 + 4)/2 = 2.5$, $(1 - 4)/2 = -1.5$ and $(9 + 6)/2 = 7.5$, $(9 - 6)/2 = 1.5$. The quadrant is now transformed to the values: 2.5, $-1.5$, 7.5, 1.5. We repeat the same process along the second dimension and we have: $(2.5 + 7.5)/2 = 5$, $(2.5 - 7.5)/2 = -2.5$ and $(-1.5 + 1.5)/2 = 0$, $(-1.5 - 1.5)/2 = -1.5$ and the quadrant is transformed to the values: 5, 0, $-2.5$, $-1.5$ as shown in Figure 2.2. We apply the same process on the other three quadrants of the array in order to complete the first level of the wavelet decomposition. In the next step, we gather the computed averages of each hyperbox (highlighted with grey color) and this way we form an array of lower resolution as shown in the 3rd step of Figure 2.2. We then repeat the same procedure for the next level of resolution. More information about the wavelet transform can be found in [CGRS01].

Error tree structures are also defined for multidimensional Haar wavelets and can be constructed (once again in linear time) in a manner similar to the one-dimensional case. Nevertheless, the semantics and structure are somewhat more complex. Figure 2.3 illustrates the error-tree structure for the two-dimensional decomposition presented above, annotated with the sign-information for each coefficient. A major difference is that in a D-dimensional error-tree, each node $t$ (except for the root) contains a set of $2^D - 1$ wavelet coefficients $c_{ti}$ that have the same support region but different signs and magnitudes for their contribution. Furthermore, each node $t$ in a D-dimensional error-tree has $2^D$ children corresponding to the quadrants of the support region of all coefficients in node $t$. The sign of each coefficient's contribution ($sign\,(j, i)$) to the j-th child of node $t$ is determined by the coefficient's position in the $2^D$-hyperbox. Coefficients located in the same position of different $2^D$-hyperboxes will be assigned the same internal index. Thus, internal indexing determines the sign of contribution of a coefficient $c_{ti}$ to each child of node $t$. For example, we observe in Figure 2.3 the sign-information for the first coefficient

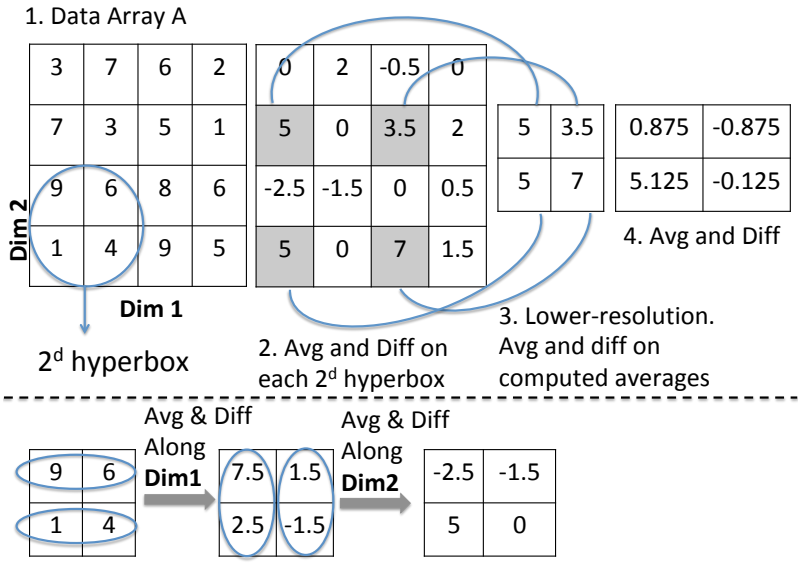



Figure 2.2: Example of two-dimensional Haar wavelet decomposition

of each node (internal index 0). Every coefficient with internal index equal to zero contributes positively to the first and third child and negatively to the second and fourth.

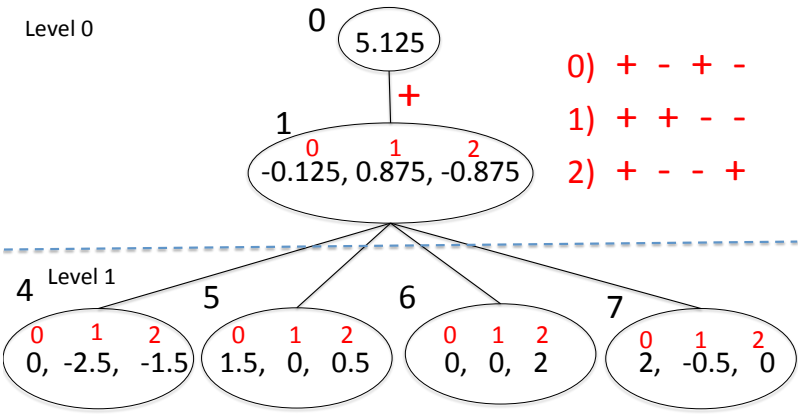



Figure 2.3: Two-dimensional error-tree. Each node contains $2^2 - 1 = 3$ coefficients and has $2^2 = 4$ children. The numbers in red color indicate the coefficients' indexing within a node.

Based on the above generalization of the error-tree structure to multiple dimensions, we can naturally extend the formula for data-value reconstruction to multidimensional Haar wavelets. Once again, the reconstruction of a data-value $d_i$ depends only on the coefficients for all error-tree nodes $\in path_{d_i}$, where the sign of the contribution for each coefficient W in node $t \in path_{d_i}$ is determined by the sign-information for W. Thus, in our example of Figure 2.3, $A[0,1] = 5.125 - 0.125 + 0.875 - 0.875 - 2.5 - (-1.5) = 4$.

Table 2.2: Notation

| **Symbol i∈0..N-1** | **Semantics** |
|---|---|
| A | Input data array |
| $W_A$ | Wavelet transform array |
| N | Number of datapoints |
| D | Number of dataset dimensions |
| B | Target size of synopsis |
| $T_i$ | Error tree rooted at node i |
| $T_L(c_i)$ ($T_R(c_i)$) | Sub-tree rooted at left (right) child of node i |
| $d_i$ | Data value at cell i of the data array |
| $\hat{d}_i$ | Reconstructed data value at cell i |
| $leaves_i$ | Set of data nodes in $T_i$ |
| $c_i$ | Wavelet coefficient at cell i |
| M | Matrix used by DP algorithm |
| $err_i$ | Signed accumulated error for $d_i$ |
| R | Size of the root subtree |
| S | Size of a base subtree |

## 2.3 The Haar Wavelet Basis for $\mathbb{R}^N$

In the previous Sections of this Chapter, we understood how the Haar Wavelet Transform (HWT) works, we saw an efficient process for constructing it and we also saw how original data values can be reconstructed. Here, I provide a more formal mathematical presentation of the transform. The formalism used in this Section and the described computations are particularly useful in stream processing cases.

Let us consider an array $A$ of $N$ data values; that is $A \in \mathbb{R}^N$. The mathematical foundation of HWT relies on vector inner-product computations over the vector space $\mathbb{R}^N$ using the Haar wavelet basis. In general, a wavelet basis $\{\phi_i\}_{i=0}^{N-1}$ for $\mathbb{R}^N$ is a basis where each vector is constructed by dilating a single function, referred to as the *mother wavelet* $\phi$. The Haar mother wavelet is defined as:

$$\phi_H(t) = \begin{cases} 1 & 0 \leq t < \frac{1}{2} \\ -1 & \frac{1}{2} \leq t < 1 \\ 0 & otherwise. \end{cases}$$

The following Lemma describes the required dilation and translation process over $\phi_H$ in order to create the Haar wavelet basis vectors.

**Lemma 1.** *The Haar wavelet basis for $\mathbb{R}^N$ is composed of the vectors*

$$\phi_{l,k}[i] = \sqrt{\frac{2^l}{N}} \cdot \phi_H \left( \frac{i - k \cdot 2^{logN-l}}{2^{logN-l}} \right) = \sqrt{\frac{2^l}{N}} \cdot \phi_H \left( \frac{i \cdot 2^l - kN}{N} \right)$$

*where $i \in [0, N-1], l = 0, ..., logN - 1$ and $k = 0, ..., 2^l - 1$, plus their orthonormal complement vector $\psi_N = \frac{1}{\sqrt{N}} \mathbf{1}^N$* [1].

Note that the $\phi_{l,k}$ vectors are essentially dilated and translated versions of the mother wavelet function $\phi_H$ over the corresponding $R_{l,k}$ dyadic support intervals. To simplify notation, we denote the Haar wavelet basis of $\mathbb{R}^N$ as the collection of vectors $\{\phi_i : i = 0, ..., N\text{–}1\}$, where $\phi_0 = \psi_N$ and $\phi_i = \phi_{l,k}$ with $l = \lfloor logi \rfloor$ and $k = i\text{–}2^{\lfloor logi \rfloor}$ for $i = 0, ..., N\text{–}1$. Considering this notation, the Haar wavelet coefficients can be defined based on the following Lemma:

**Lemma 2.** *Each of the (normalized) coefficients $c_i^*, i = 0, ..., N\text{–}1$ in the HWT of the data array $A \in \mathbb{R}^N$ can be expressed as the inner product of $A$ with the corresponding Haar basis vector $\phi_i$, i.e.,*

$$c_i^* = \langle A, \phi_i \rangle = \sum_{j=0}^{N-1} A[j]\phi_i[j].$$

It can be shown that the above Haar vector basis $\{\phi_i\}$ $i = 0, ..., N\text{–}1$ is an orthonormal basis of $\mathbb{R}^N$. For any pair of basis vectors $\phi_k, \phi_l$, it holds that $\langle \phi_k, \phi_l \rangle = 1$ if $k = l$ and 0 otherwise. The reconstruction of the original data array $A \in \mathbb{R}^N$ is then based on the linear combination of the Haar wavelet basis vectors and the corresponding HWT coefficients. More formally:

$$A = \sum_{j=0}^{N-1} c_i^* \phi_i$$

.

Haar wavelets are also an example of a wavelet system with compact support. The notion of compact support is described in Lemma 3.

**Lemma 3.** *A wavelet system is considered to have compact support if for any any basis vector $\phi_k$ there exists a closed interval $I = [a, b]$ such that $\phi_k[x] = 0$ for any $x \notin I$.*

Haar wavelets, discovered in 1910, were the only known wavelets of compact support until the discovery of the Daubechies wavelet families in 1988 [Dau92].

[1] $\mathbf{1}^N$ denotes the $N$-vector whose entries are all equal to 1.

## 2.4 Wavelet Thresholding

The complete Haar wavelet decomposition $W_A$ of a data array $A$ is a representation of equal size as the original array. Given a budget constraint $B < N$, the problem of *wavelet thresholding* is to select a subset of at most $B$ coefficients that minimize an aggregate error measure in the reconstruction of data values. The non-selected coefficients are implicitly set to zero. The resulting wavelet synopsis $\hat{W_A}$ can be used as a compressed approximate representation of the original data. For assessing the quality of a wavelet synopsis, many aggregate error-measures have been proposed [CGHJ12]. Among the most popular metrics are the *mean squared error* ($L_2$), the *maximum absolute error* and the *maximum relative error*:

$$L_2\left(W_A, \hat{W_A}\right) = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(\hat{d}_i - d_i\right)^2} \tag{2.2}$$

$$max\_abs\left(W_A, \hat{W_A}\right) = max_{i=1}^{N}\left\{|\hat{d}_i - d_i|\right\} \tag{2.3}$$

$$max\_rel\left(W_A, \hat{W_A}\right) = max_{i=1}^{N}\left\{\frac{|\hat{d}_i - d_i|}{max\{|d_i|, S\}}\right\} \tag{2.4}$$

In the above equations, $\hat{d}_i$ denotes the approximate value for datapoint $d_i$ and $S$ is a *sanity bound* used to prevent the influence of very small values in the aggregate error [VW99, GG02, GK04].

A preliminary approach to the thresholding problem is based on two basic observations about a coefficient's contribution in the reconstruction of the original data values. The first observation is that coefficients of larger values are more important, since their absence causes a larger absolute error in the reconstructed values. Second, a coefficient's significance is larger if its level in the error-tree is higher, as it participates in more reconstruction paths of the error-tree. Putting both together, the significance $c_i^*$ of a coefficient is defined by $c_i^* = |c_i|/\sqrt{2^{level(c_i)}}$, where $level\,(c_i)$ denotes the level of resolution at which the coefficient resides (0 corresponds to the "coarsest" resolution level).

Accordingly, the conventional thresholding scheme is a greedy algorithm that retains the $B$ largest normalized wavelet coefficients; that is those with the greatest significance. It has been shown [SDS96] that this approach minimizes the $L_2$-error. By the orthonormality of the Haar wavelet basis, the HWT preserves the Euclidean length or $L_2$-norm of any vector (*Parseval's Theorem*) [Mal99, SDS96]; then, for the error vector we have:

$$L_2\left(A,\hat{A}\right) = \|A-\hat{A}\|_2 = \sqrt{\sum_{i=0}^{N-1}\left(A[i]-\hat{A}[i]\right)^2} = \sqrt{\sum_{c_i^*\in\hat{W}_A}\left(c_i^*\right)^2} = L_2\left(W_A,\hat{W}_A\right) \quad (2.5)$$

Thus, minimizing the $L_2$-norm in the wavelet domain also results in its minimization in the domain of the original data. Nevertheless, the minimization of the $L_2$-error does not provide maximum error guarantees for individual approximate answers. As a result, the approximation error of individual values can be arbitrarily large, resulting into high variance in the quality of data approximation and severe bias in favor of certain regions of the data. This problem is particularly striking whenever a series of omitted coefficients lies along the same path of the error-tree. Maximum error metrics are more robust [GG02, GK04], since they set a maximum error guarantee on individual values. The problem of minimizing maximum error metrics can be formulated as follows:

**Problem 1** (Wavelet Thresholding for Non $L_2$-errors)**.** *Given a data array $A$ of size $N$ and a budget $B$, construct a representation $\hat{W}_A$ of $A$ that minimizes a maximum error metric, while it retains at most $B$ non-zero coefficients.*

In order to assist the discussion of some algorithms, a definition of the dual of Problem 1 is also provided:

**Problem 2** (Dual Problem)**.** *Given a data array $A$ of size $N$ and an error bound $\epsilon$, construct a representation $\hat{W}_A$ of $A$ such that $max\_abs \leq \epsilon$ and the number of non-zero entries $s^*$ in $\hat{W}_A$ is minimized.*

In this dissertation, I focus on designing algorithms for Problem 1 that can specifically scale in Big Data scenarios. The majority of existing algorithms for the problem are of quadratic complexity and either need to load the whole dataset in memory or operate on a small working set and make very frequent disk accesses to update it. The increasing sizes of data to be processed render centralized approaches unusable in terms of performance and scalability. To overcome these shortcomings, in this thesis, novel parallel algorithms are proposed. The initial problem is decomposed to smaller local sub-problems which are solved in parallel and partial solutions are derived; we then utilize these solutions to derive the final one.

As we are going to see in Chapter 3, to demonstrate the benefits of the proposed approach, I apply it on a state-of-the-art DP algorithm [KSM07]. As the algorithm of [KSM07] makes use of unrestricted Haar wavelets, in the next Section, a presentation of unrestricted wavelets is provided.

## 2.5 Unrestricted Haar Wavelets for Non-$L_2$ Error

As described in the previous Section, wavelet synopsis construction is a sparse wavelet representation problem where, given a wavelet basis $\{\phi_i\}_{i=0}^{N-1}$ for $\mathbb{R}^N$ and an input data vector $A \in \mathbb{R}^N$, the goal is to construct an approximate representation $\hat{A}$ as a linear combination of at most $B$ basis vectors so as to minimize some normed distance between $A$ and $\hat{A}$. The sparse $B$-term representation $\hat{A}$ belongs to the non-linear space $\left\{\sum_{i=0}^{N-1} z_i\phi_i : z_i \in \mathbb{R}, \|Z\|_0 \leq B\right\}$, where the $L_0$ norm $\|Z\|_0$ denotes the number of non-zero coefficients in the vector $Z = (z_0, z_1, ..., z_{N-1})$. In the analysis of this Section, the $z_i \in \mathbb{R}$ values do not have to be Haar wavelet coefficients.

For the case of $L_2$ error, by Parseval's theorem, the $L_2$ norm of $A - \hat{A}$ is preserved in the wavelet space; thus, generalizing Equation 2.5, we have:

$$\|A - \hat{A}_Z\|_2^2 = \sum_i \left(A[i] - \sum_j z_j\phi_i[j]\right)^2 = \sum_i \left(\langle A, \phi_i\rangle - z_i\right)^2.$$

It is clear that the optimal solution under the $L_2$ error measure is to retain the largest $B$ inner products $\langle A, \phi_i\rangle$ which are exactly the largest (normalized) coefficients $c_i^*$ in the HWT expansion of $A$. Thus, the greedy thresholding approach [2] is optimal for $L_2$-error minimization even in this generalized setting. For other error norms, however, restricting the $z_i$-values to the set of computed HWT coefficients of $A$ can result in suboptimal solutions.

A first step in solving the generalized (unrestricted) sparse Haar wavelet representation problem is demonstrating the existence of a bounded set $R$ from which coefficient values in $Z$ can be chosen while ensuring a solution that is close to the optimal unrestricted solution (where $z_i$-values range over all reals). Guha and Harb [GH05] prove that, for $L_p$-error minimization, the maximum (un-normalized) coefficient value in the optimal solution $Z^*$ satisfies $max_i\{|z_i^*|\} \leq 2N^{\frac{1}{p}}a_{max}$, where $a_{max} = max_i\{|A[i]|\}$ (i.e., the maximum absolute value in the input data). Furthermore, they demonstrate that, by rounding the coefficient values in the optimal solution $Z^*$ to the nearest multiple of some $\delta > 0$ (obtaining a rounded solution $\hat{Z}_\delta$) introduces bounded additive error in the target $L_p$ norm; more specifically,

$$\|A - \hat{A}_{\hat{Z}_\delta}\|_p \leq \|A - \hat{A}_{Z^*}\|_p + \delta N^{\frac{1}{p}} min\{B, logN\}.$$

Thus, the above additive error over the optimal solution can be guaranteed while restricting the search for coefficient values over a set of size [GH05]:

$$|R| = 2 \cdot \frac{max_i\{|z_i^*|\}}{\delta} \leq \frac{4N^{\frac{1}{p}}a_{max}}{\delta}.$$

[2] Keeping the $B$ largest normalized coefficients

CHAPTER 3

# Parallel synopsis construction for maximum error metrics

## 3.1 Introduction

In Chapter 2, we saw that the construction of a wavelet synopsis that minimizes the $L_2$-error is an easy task to accomplish. However, this is not the case for non-Euclidean errors. The majority of algorithms that target the problem of $L_p$ minimization with $p \neq 2$, present time complexities that prevent scaling to big datasets. A complete survey of such algorithms is provided in Section 6.2.

In this Chapter, we are going to see how we can overcome the shortcomings of existing methods through parallelization. The proposed algorithms follow the MapReduce [DG08] paradigm and target one-dimensional wavelets, where the original data array is of the form $A \in \mathbb{R}^N$. The presented experiments verify the linear scalability of the proposed design and demonstrate results on datasets more than three orders of magnitudes larger than the ones used in previous related research.

## 3.2 Scaling DP algorithms

Since the majority of the proposed algorithms for Problem 1 are based on DP, in this Section I present a general framework that can be used for their parallelization and efficient execution

over modern distributed platforms. To achieve that, a locality-preserving partitioning scheme is proposed. The proposed scheme exploits the structure of the error-tree and assign different sub-trees to different workers.

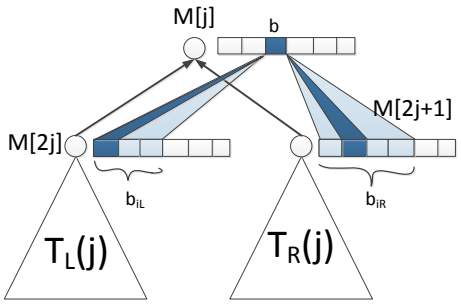



Figure 3.1: DP recursion on the error-tree. Node $c_j$ combines the M-rows of its children in order to produce $M[j]$

In DP-based algorithms, each row of the DP-matrix M is assigned to a node of the error-tree. The contents of such a row differ between algorithms. Despite the different structure of the rows of M, all these algorithms follow a bottom-up fashion, where the rows corresponding to the leaves of the error-tree are computed first. The row for each internal node is computed by combining the already computed rows of its children according to an optimality criterion. Thus, computing the row for any node of the error-tree, demands two more rows to be in memory. To compute the values for a single cell of a row j, many cells of the children-rows are examined and, eventually the one that optimizes a defined metric is selected. This procedure is illustrated in Figure 3.1. The shadowed cells represent the examined values and the bold colored ones the finally selected values. In Figure 3.1, the left and right subtree of a node $c_j$ ($T_L(j)$ and $T_R(j)$ respectively) can be computed independently of each other. Based on this observation, the idea is to apply a partitioning scheme that hierarchically decomposes the error-tree to independent subtrees of a fixed height $h$, $h < logN$. This partitioning scheme is presented in Figure 3.2 and results to $\lceil \frac{logN}{h} \rceil$ layers of subtrees. We denote $Layer_i$ to be all the subtrees located in layer $i$ and it holds that:

$$|Layer_i| = \begin{cases} \frac{N}{2^{h \cdot i + i - 1}} & i = 1, .., \lfloor \frac{logN+1}{h+1} \rfloor \\ 1 & i = \lceil \frac{logN+1}{h+1} \rceil \end{cases} \tag{3.1}$$

For the parallelization of the existing DP algorithms for Problem 1, a MapReduce strategy is followed. The idea is to run a distributed job for each layer of sub-trees with the bottommost layer starting first. The *map* function of a job $J_i$ computes the DP matrix row for the local root of a sub-tree in $layer_i$. More specifically, if the local root is the node $c_j$, the emitted key-value is $(j, M[j])$. The pseudocode for the map function can be found in Algorithm 1. Naturally, proper partitioning

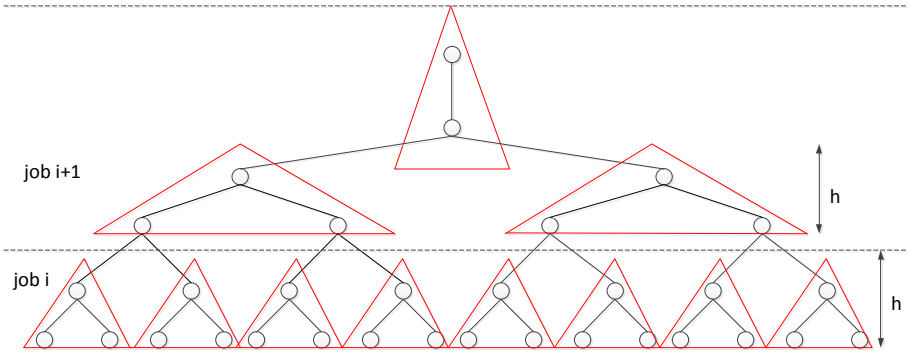



Figure 3.2: Partitioning for parallelizing DP algorithms for Problem 1

should be applied in order to preserve the sub-tree locality during suffling. The *reducers* collect the received key-values and output them in appropriately created partitions. This way the leaves of the sub-trees in $layer_{i+1}$ have been created and job $J_{i+1}$ is ready for execution. In the case of the topmost distributed job, instead of writing down the collected key-values, the reducer keeps them in-memory and directly runs the DP algorithm on the corresponding sub-tree. Algorithm 2 presents the whole procedure.

**Algorithm 1:** Map Function

**Require:** Data values for a sub-tree $s$

1: Run DP on $s$.
2: emit $(j, M[j])$ // $c_j$ is the local root

**Algorithm 2:** Parallel execution of a DP algorithm for Problem1

**Require:** Data size N, sub-tree height h

1: Partition the error-tree to sub-trees of fixed height h.
2: $i = 1$
3: **while** $i \leq \lfloor \frac{logN+1}{h+1} \rfloor$ **do**
4: **if** $i > 1$ **then** Combine M-rows from layer $i-1$
5: **for all** $T_j \in Layer_i$ in parallel **do**
6: Run DP on $T_j$
7: Send the computed row of node $j$ to the next layer
8: $i = i + 1$
9: Run DP on topmost subtree.

As a distributed approach, it is clear that this idea incurs a communication overhead. For every sub-tree of the error-tree, the row of M that corresponds to the local root is transferred over to the workers of the next stage. The following Lemma quantifies the cost of this overhead.

**Lemma 4.** *The overall communication cost of Algorithm 2 is:*

$$O\left(\frac{N \cdot max\left\{|M\left[j\right]|\right\}}{2^h}\right) \tag{3.2}$$

*Proof.* Let $|M\left[j\right]|$ denote the size of the row corresponding to node $c_j$. Then, according to Equation 3.1, the communication overhead for the i-th stage is:

$$O\left(|Layer_i| \cdot max_{j \in Layer_i}\left\{|M\left[j\right]|\right\}\right) \quad = \quad O\left(\frac{1}{2^{h \cdot i + i - 1}} N \cdot max_{j \in Layer_i}\left\{|M\left[j\right]|\right\}\right) \tag{3.3}$$

and thus, the overall communication overhead: $O\left(\frac{N \cdot max\{|M[j]|\}}{2^h}\right)$ □

Equation 3.2 represents the generic communication complexity of all DP algorithms when the proposed partitioning scheme is applied. The maximum M-row size $max\left\{|M\left[j\right]|\right\}$, which determines the complexity, depends on the used algorithm.

After the completion of Algorithm 2, it is only the optimal approximation error that is computed and not the synopsis itself. To compute the synopsis, all DP algorithms require one additional step: a top-down recursive procedure on the error-tree in order to select the appropriate coefficients. Starting from the root this time, we re-enter the sub-problem of the topmost subtree and select the coefficients to retain. When the processing of the topmost subtree is over, we know which coefficients are retained from this subtree and also the leaves of the subtree are aware of which cells of the M-rows of their children are the best choice in order to obtain the optimal synopsis. Thus, each leaf-node of the topmost subtree sends a message to its children to inform them about the optimal choice they can make. With this message, the children recursively re-enter the sub-problems of the next layer of subtrees. This procedure has $O\left(N\right)$ time complexity, as it needs to visit exactly once each node, and $O\left(\frac{N}{2^h}\right)$ communication complexity between the partitions-subtrees.

For demonstrating the merits of the proposed approach, the described methodology is applied on IndirectHaar [KSM07] creating DIndirectHaar; a distributed version of the centralized algorithm. The conducted experiments in Section 3.5 show that DIndirectHaar scales linearly over both data and cluster size.

At this point, I also want to discuss the choice of IndirectHaar. An exact solution for Problem 1 demands tabulation over all possible space allocations for each node of the error-tree. This burden renders the majority of DP-algorithms impractical in terms of memory consumption. IndirectHaar exploits the dual error-bound problem (Problem 2) which is easier to be solved and employs a binary search procedure (Algorithm 3) to derive a solution for the initial problem. Thus, in the case of IndirectHaar, the DP algorithm that is actually parallelized by the described

framework is MinHaarSpace[1] [KSM07] and targets Problem 2. Obviously, this results in multiple distributed jobs of input size N. Furthermore, in order to compute the lower and upper error bounds (lines 1-2), an overhead of two extra jobs is required. For the lower bound, we compute the (B+1)-largest coefficient. Each worker emits its local wavelet coefficients in reverse order, i.e., largest first, and in a next step these coefficients are merged and the first $B+1$ are retained. For the upper bound, assuming that a B-term synopsis fits in memory, we load the B-largest-terms synopsis in the main memory of each worker and we bottom-up compute the maximum absolute error. Assuming that $E(j)$ denotes the maximum absolute error in sub-tree $T_j$, it holds that $E(j) = max\{E(2j), E(2j+1)\}$. Thus, for computing the upper bound of the error for the binary search of Algorithm 3 a MapReduce job is required. The mappers compute the $max\_abs$ of a sub-tree in a bottom-up fashion and emit the key-value: $(j, E(j))$. The reducers collect and combine the errors in order to prepare the input for the next job.

**Algorithm 3:** DIndirectHaar

1: $e_u$ =maximum absolute error for B-largest-terms synopsis
2: $e_l = (B+1)$-largest coefficient
3: $e_{low} = e_l$;$e_{high} = e_u$
4: **while** not finished **do**
5: $e_{mid} = \frac{e_{high}+e_{low}}{2}$
6: $\hat{W}_A$ =DMHaarSpace($e_{mid}$);$\bar{B}$ =size of $\hat{W}_A$
7: $\bar{e}$ =actual maximum absolute error of $\hat{W}_A$
8: **if** $\bar{B} < B$ **then**
9: $\tilde{W}_A$ =DMHaarSpace($< \bar{e}$);$\tilde{B}$ =size of $\tilde{W}_A$
10: **if** $\tilde{B} > B$ **then** finished=1
11: **else** $e_{high} = \bar{e}$
12: **else**
13: **if** $\bar{B} > B$ **then** $e_{low} = e_{mid}$
14: **else** finished=1

For achieving even better results, IndirectHaar can also be applied on Haar+ trees [KM07]. However, as Haar+ trees have a slightly different structure and work on triads of coefficients, for the ease of understanding, I keep the presentation on the classic Haar error-tree.

### 3.2.1 Scaling DP Algorithms with Hardware Accelerators

The MapReduce algorithms naturally fit distributed platforms like Apache Hadoop and Spark. However, as they expose a SPMD style of programming, they can also be efficiently executed over accelerators that favor data parallelism (e.g., GPUs).

[1]For the ease of understanding, a detailed description of MinHaarSpace along with all technical details can be found in Appendix A

In this Section, it is described how MinHaarSpace can be further parallelized for taking full advantage of the available hardware. Although the discussion is restricted to MinHaarSpace, since all DP algorithms for constructing wavelet synopses share a common computational pattern, the same approach can be followed for any of them. In order to enable portability among different architectures (x86, GPU), an implementation in OpenCL [ope] is presented.

For the parallelization of MinHaarSpace in OpenCL, we follow a partitioning scheme similar to the one discussed above. Each layer of sub-trees corresponds to at least one kernel-launch. The idea is to first run dynamic programming in parallel over the sub-trees of the bottommost layer. This computation is packed into a kernel and takes place at the target device.

As OpenCL does not support dynamic memory allocation, we should pre-allocate the whole amount of space that is required for storing the DP matrix. However, depending on the $\epsilon$, $\delta$ parameters of MinHaarSpace, the DP matrix may not fit in global memory. In that case, we also apply a vertical partitioning as demonstrated in Figure 3.3 by the dashed line; we first execute the part of the error-tree that is at the left of the dashed line, followed by the part at the right side and then we combine them.

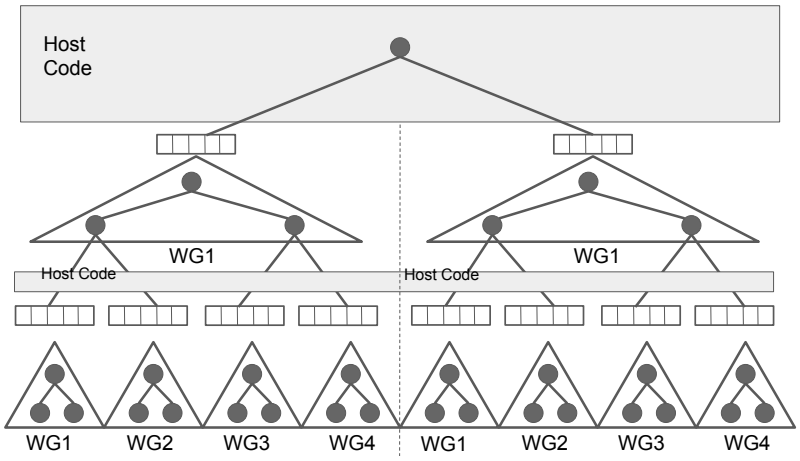



Figure 3.3: Partitioning used for parallelizing DP-based algorithms with OpenCL

When processing of the bottommost layer is over, the computed rows for the roots of these sub-trees are returned to the host that prepares the kernels and data for the next layer in order to repeat the same process towards the root.

In the presented implementation, the computation of a sub-tree is assigned to a work group. The size of a work group cannot be arbitrarily set but depends on the device. It follows that the maximum $h$ we can set and thus, the minimum number of kernel launches is hardware-dependent.

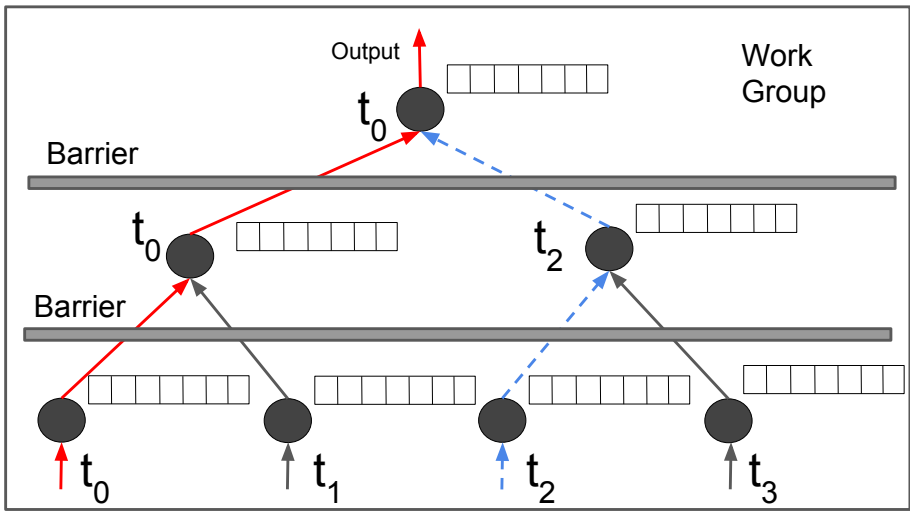



Figure 3.4: Parallelization within a work group

The execution of work items within a work group is depicted in Figure 3.4. Initially, each thread/work item takes on the computation of a leaf of the error-tree. The computed row is persisted into the global memory of the device. When processing of a leaf is over, a work item waits at a barrier until all work items of the same group finish their computations. As the first level of the error-tree contains half the number of the leaves, only half of the initially spawned work items continue processing. Then, a reduction takes place, and the active threads combine the $M$-rows of the leaves in order to produce the corresponding rows of the first level. This process is repeated towards the root of the work group by halving the number of active threads at each level. Finally, it is only the first work item of the group that computes and persists the output. In Figure 3.4, we illustrate threads of different lifetime with a different color.

### Discussion

One of the benefits of OpenCL is the support for heterogeneous computing. The same kernel can run exactly as is over different hardware devices. However, depending on the device, we can make targeted optimizations in order to boost performance. In our case, we slightly differentiate the kernels for x86 and GPU architectures with respect to memory access patterns. While the DP matrix is hierarchically represented in Figure 3.4, in reality it is a one dimensional array that resides in global memory. In the x86 case, a single thread is benefited from sequential memory accesses. Contrarily, in GPU architectures we try to make coalesced memory transactions and consecutive threads should access consecutive memory addresses. Thus, the kernels we provide for both architectures are identical apart from the pointer arithmetic in global memory.

With the OpenCL implementation, we can speedup the construction of a wavelet synopsis by taking advantage of accelerators that may locally exist. When using a framework such as

Hadoop, the construction process is scaled by distributing work to different machines. As the Hadoop workers described in Section 3.2 are single-threaded, I argue that these two approaches are orthogonal to each other and a combination of the two could bring the best results in the case of a large dataset. The idea is to partition the dataset into sub-trees at various levels: (i) in the big data framework level, a first partitioning is applied and each machine obtains a different sub-tree, (ii) then, in the node level, where the sub-tree of a single machine is further partitioned into work groups as explained. For implementing such an idea, one needs to run OpenCL code through Java[2]. However, there are already tools for this purpose, like SWAT [GS16] that permits writing Spark UDFs as OpenCL kernels.

## 3.3 Parallel Greedy Approaches

As the DP-based solutions incur high computational overhead, there is often a need for a faster approach at the cost of approximation quality. This is exactly what the GreedyAbs [KM05] algorithm achieves. However, this algorithm is not easily parallelizable and cannot scale for big datasets. In this Section, two novel, fully parallel greedy algorithms are presented that both are based on : (i) a partitioning scheme similar to the one presented in Section 3.2, and (ii) merging and filtering of partial results.

### 3.3.1 *GreedyAbs*: The Centralized Solution

For the ease of understanding, I first give a description of the GreedyAbs algorithm [KM05]. Let $err_j = \hat{d}_j - d_j$ be the signed accumulated error for a data node $d_j$ in a synopsis $\hat{W}_A$, yielded by the deletions of some coefficients. To assist the iterative step of the greedy algorithm, for each coefficient $c_k$ not yet discarded, we introduce the *maximum potential absolute error* $MA_k$ that $c_k$ will contribute on the running synopsis, if discarded:

$$MA_k = max_{d_j \in leaves_k}\{|err_j - \delta_{jk} \cdot c_k|\} \tag{3.4}$$

Computing $MA_k$ normally requires information about all $err_j$ values in $leaves_k$. A naive method to compute $MA_k$ is to access all $leaves_k$, where $err_j$ are explicitly maintained. The disadvantages of this approach are the explicit maintenance of all $err_j$ values at each step and the cost required to update $MA_k$ values after the removal of a coefficient.

A more efficient solution for updating $MA_k$ is reached by exploiting the fact that the removal of a coefficient equally affects the signed costs of all data values in its left or right sub-tree. For example, in Figure 2.1, the removal of coefficient $c_2 = -4$ increases the signed errors of data nodes $d_0$, $d_1$, and decreases the signed errors of $d_2$, $d_3$ by 4. Accordingly, the maximum

[2]Big data frameworks usually work over the JVM.

and minimum signed errors in the left (right) sub-tree of a removed coefficient $c_i$ are decreased (increased) by $c_i$. The maximum absolute error incurred by the removal necessarily occurs at one of these four positions of existing error extremum. Hence, the computation of $MA_k$ requires that only four quantities be maintained at each internal node of the tree. These are the maximum and minimum signed errors for the $leftleaves_k$ and $rightleaves_k$, and are denoted by $max_k^l$, $min_k^l$, $max_k^r$, and $min_k^r$, respectively. It follows that Equation 3.4 is equivalent to:

$$MA_k = max\{|max_k^l - c_k|, |min_k^l - c_k|, |max_k^r + c_k|, |min_k^r + c_k|\} \quad (3.5)$$

In the complete wavelet decomposition, these four quantities are all 0, since $err_j = 0, \forall d_j$. Thus, $MA_k = |c_k|, \forall k$ and the greedy algorithm removes the smallest $|c_k|$ first. In order to efficiently decide which coefficient to choose next, all coefficients are organized in a min-heap structure based on their $MA_k$. After the removal of a coefficient $c_k$, $err_j$ for all $leaves_k$ changes, so the information of all descendants and ancestors of $c_k$ must be updated. All the error quantities of the descendants in the left (right) sub-tree of $c_k$ are decreased (increased) by $c_k$. During this process, a new $MA_i$ is computed for each descendant $c_i$ of $c_k$. In accordance, the changes in error quantities are propagated upwards to ancestors $c_i$ of $c_k$ and $MA_i$ values are updated as necessary. While updating error quantities and $MA$ values, the position of $c_k$'s descendants and affected ancestors are dynamically updated in the heap. This procedure of removing nodes is repeated until only B nodes are left on the tree.

Another important thing to note is that the maximum absolute error does not change monotonically when a coefficient is removed. In other words, after deleting a coefficient $c_k$ the maximum absolute error of its affected data values may decrease. As a result, choosing exactly B coefficients may not be the best solution given a space budget B. For this reason, we keep removing coefficients after the limit of B has been reached, until no coefficient remains in the tree. From all B + 1 coefficient sets (B coefficients left, B-1 coefficients left, etc.) produced at the last B steps of the algorithm, the one with the minimum maximum absolute error is kept.

### 3.3.2 DGreedyAbs: Scaling the Greedy Algorithm

GreedyAbs presents an inherent drawback for its parallelization. At each step, it needs global knowledge of the whole error-tree. To solve the problem in parallel, I consider a partitioning similar to the one we used for the parallelization of the DP algorithms. In the proposed scheme, the error-tree is partitioned into one *root subtree* and multiple *base subtrees*, as shown in Figure 3.5.

At each iteration of GreedyAbs, the node $c_k$ with the smallest MA is selected to be discarded. After its deletion, all the other nodes that lie either in $path_k$ or $T_k$ may update their MA values. Ideally, we would like to take decisions at each base sub-tree independently of each other. For the

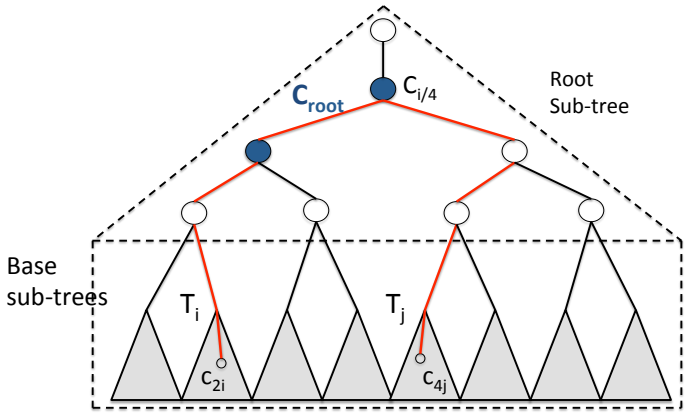



Figure 3.5: Partitioning for parallelizing *GreedyAbs*. The red line illustrates an example of communication between two base subtrees. The blue-filled nodes show a possible $C_{root}$ set.

parallelization of the algorithm, the main difficulty is that the base sub-trees communicate with each other through the root sub-tree. For example, consider a scenario, like the one depicted in Figure 3.5, where node $c_{2i}$ is selected to be removed from the base sub-tree $T_i$ and, at the same time, node $c_{4j}$ is selected from $T_j$. The removal of $c_{2i}$ may dictate that node $c_{i/4}$ should be discarded at the next step. On the other hand, discarding $c_{4j}$ can make $c_{i/4}$ a really important coefficient for sub-tree $T_j$ and thus its deletion could produce a big maximum error. It is clear that such situations lead to conflicts that prohibit a straight-forward parallel implementation.

In order to proceed towards a correct parallel computation, we need to offer more isolation to the base sub-trees. The idea behind our solution is the following: Let us assume that we somehow know which nodes of the root sub-tree are retained in the final synopsis and call this set of nodes $C_{root}$. Having selected a $C_{root}$, we can remove the remaining root sub-tree and there are $B - |C_{root}|$ nodes that still need to be selected for the synopsis.

Consider now a base sub-tree $T_j$. The deletion of nodes $c_i \in$ root sub-tree$\backslash C_{root}$ incurs an incoming error to $T_j$. For example, in the error-tree of Figure 3.6, if we delete nodes $\{c_0, c_2\}$, there is an incoming error $-7 - 4 = -11$ to sub-tree $T_5$. Thus, if the incoming error to sub-tree $T_j$ is $e_{in}$, we set the signed accumulated errors to: $err_i = e_{in}, \forall d_i \in T_j$ and run GreedyAbs on $T_j$. The output of $GreedyAbs\,(T_j)$ is an ordered list $L_j$ of $N_z\,(T_j)$ coefficients, where $N_z\,(T_j)$ is the number of non-zero coefficients in the sub-tree. The list is in reverse order of the one in which coefficients $c_i \in T_j$ were deleted by GreedyAbs. More specifically, each element of the list is a tuple $(delOrd, id, err)$ that indicates the order with which the coefficient with index $id$ was deleted and the incurred maximum absolute error $err$. This procedure of locally executing GreedyAbs on a sub-tree, is carried out in parallel for all base sub-trees in the map phase of a MapReduce job. In a pre-processing step, the error-tree is partitioned and each base sub-tree is stored in a separate HDFS file. Then, each mapper takes on a sub-tree and runs GreedyAbs on it.

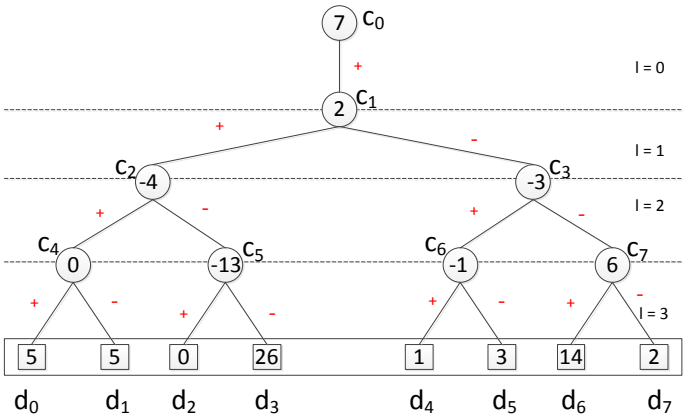



Figure 3.6: Error-tree example.

When this stage of parallel GreedyAbs runs is over, in the reduce phase, we collect and error-wise merge the outputs from all the base sub-trees (i.e., $\forall T_j \in$ base sub-trees merge $L_j$), thus obtaining a global list where the node deletion order of each sub-tree is preserved. The synopsis needs to contain those coefficients that are the most important for each sub-tree, i.e., the ones that were last emitted. Therefore, by keeping the last $B - |C_{root}|$ elements of the global list, let us call them $C_{base}$, we form the final synopsis: $C_{root} \cup C_{base}$. This pseudocode for the whole MapReduce job is presented in Algorithm 4.

**Algorithm 4:** distrGAbs: MR job for computing the synopsis given a $C_{root}$ set

**Require:** error-tree, space budget $B$, $C_{root}$ set
1: **for all** $T_j \in$ base sub-trees in parallel **do**
2: $err_i = err_i + e_{in}, \forall err_i \in T_j$ // $e_{in}$: incoming error from $C_{root}$
3: $L_j = GreedyAbs\,(T_j)$;emit $L_j$
4: $L = merge(L_j$ lists)
5: store last $C_{root} \cup (B - |C_{root}|)$ elements of $L$ as synopsis
6: **return** $min\,\{L\,[0]\,.error, ..., L\,[B - |C_{root}| - 1]\,.error\}$

So far, we have ignored the procedure that finds the appropriate nodes to be retained from the root sub-tree, assuming it is provided by an "oracle". As we cannot compute a-priori which these nodes are, we need to *speculatively* create the synopses for different $C_{root}$ sets and finally retain the one that produces the best approximation. Let $R$ denote the size of the root sub-tree. Since we do not know the number of nodes that should be retained from the root sub-tree, we should consider at least $min\{R, B\} + 1$ different $C_{root}$ sets, with each candidate $C_{root}$ having different size: The empty set, as we may keep none of these nodes, keep only 1 node, keep 2 nodes, etc., until we examine the case where $min\{R, B\}$ nodes are kept. In order to find $min\{R, B\} + 1$ candidate $C_{root}$ sets, we run GreedyAbs on the root sub-tree. The intuition behind this choice is that, since only the root sub-tree is considered known at this stage, we

should try to optimize the local problem and each time discard the node that incurs the minimum error. GreedyAbs on the root sub-tree runs in a centralized fashion. Since the root sub-tree can be exponentially smaller than the original dataset, its processing on a single machine is done without compromising performance. The candidate $C_{root}$ sets are generated by the $genRootSets$ function presented in Algorithm 5.

**Algorithm 5:** $genRootSets$: Generates candidate $C_{root}$ sets of different lengths

**Require:** root subtree,B

1: $L_{root} = GreedyAbs$(root sub-tree)
2: $C = \{\{\}\}; lastIndex = L_{root}.size$
3: **for** $(i = lastIndex; i > lastIndex - B; i = i - 1)$ **do**
4: $\quad C_{root,i} = \{L_{root}[i], .., L_{root}[lastIndex]\}; C = C \cup \{C_{root,i}\}$
5: **return** $C$

For example, we consider as root sub-tree the nodes $\{c_0, c_1, c_2, c_3\}$ of the error-tree depicted in Figure 3.6. The run of GreedyAbs selects to discard the nodes according to the following order: $[c_1, c_3, c_2, c_0]$. Thus, the candidate $C_{root}$ sets are the following 5:
$\Gamma = [\{c_1, c_3, c_2, c_0\}, \{c_3, c_2, c_0\}, \{c_2, c_0\}, \{c_0\}, \{\}]$. For constructing the synopsis, we perform a search in the space of possible solutions. We start by examining the achieved quality of the corner cases, i.e., keeping in the synopsis 0 and $min\{R, B\}$ coefficients from the root sub-tree. If these extreme cases result in errors $e_h, e_l$ with $|e_h - e_l| < \epsilon \rightarrow 0$, then the algorithm finishes and we keep as a final synopsis the one that produced the $min\{e_h, e_l\}$. Otherwise, we replace the $C_{root}$ that produced the $max\{e_h, e_l\}$ with another $C_{root}$ produced by Algorithm 5 and repeat the same process. The selection of the next $C_{root}$ does not come from a random choice. When the distributed execution of the greedy algorithm for a given $C_{root}$ set is over, we know the maximum absolute error that appeared in each base subtree. By knowing that information, we know which subtrees need further improvement. Thus, we select $C_{root}$ sets that contain coefficients which support these subtrees. In our example, we begin by running GreedyAbs on each base subtree for the $C_{root}$ sets: $\{\}$, $\{c_1, c_3, c_2, c_0\}$. Let us assume that they yield synopses with errors 10 and 5 respectively. In that case, as $\{\}$ produced the worst error, it is replaced by $\{c_0\}$ and we now compare the quality of the synopses yielded by $\{c_0\}$ and $\{c_1, c_3, c_2, c_0\}$ $C_{root}$ sets. The described procedure implies that $O(R)$ jobs may be demanded. However, our experiments in Section 3.5 show that the number of jobs that the algorithm needs in order to converge is constant in practice. The complete DGreedyAbs algorithm is presented in Algorithm 6.

The running-time and communication complexity of DGreedyAbs are provided by the following Lemma:

**Lemma 5.** *Let us denote with $R$ the size of the root sub-tree, $S$ the size of a base sub-tree and $Nz(S)$ the number of non-zero coefficients of a base sub-tree. Then, the asymptotic running-time*

**Algorithm 6:** DGreedyAbs

**Require:** error-tree, space budget $B$

1: $\Gamma$ = $genRootSets$(root subtree,$B$)
2: $C_{root,l} = \{\}$
3: $C_{root,h} =$ the $min\{R, B\}$-th set of $\Gamma$
4: $e_h = distrGAbs\left(C_{root,h}\right)$
5: $e_l = distrGAbs\left(C_{root,l}\right)$
6: **while** $e_h - e_l > \epsilon$ **do**
7: find maximum error $e_{l,i}$ per subtree from the run for $C_{root,l}$
8: sort $\{e_{l,i}\}$ errors in descending order
9: find the first k-largest errors $E$ that satisfy: $|e_i - e_j| < \epsilon$
10: Consider the set $C =$ all coefficients $\in path_e, \forall e \in E$
11: $C_{root,l} =$ next set $\in \Gamma$ that contains at least one more coefficient from $C$
12: $e_l = distrGAbs\left(C_{root,l}\right)$
13: $C^*_{root}$ = $C_{root}$ set that yielded the $min\{e_l, e_h\}$
14: FINAL_SYNOPSIS = the produced synopsis for $C^*_{root}$
15: **return** FINAL_SYNOPSIS

*complexity of a DGreedyAbs mapper is* $O\left(Nz\left(S\right) log^2 Nz\left(S\right)\right)$*, the complexity of a reducer is* $O\left(Rmax\{B, Nz\left(S\right)\}\right)$ *and the communication cost is* $O\left(Rmax\{B, Nz\left(S\right)\}\right)$.

*Proof.* The worst-case cost of GreedyAbs is $O\left(Nlog^2 N\right)$ [KM05], thus it follows that the mappers of DGreedyAbs will have the same complexity. A distributed job over an error-tree with a root sub-tree of size $R$ is going to have $R$ partitions. Since each mapper emits discarded coefficients in the form of a list of size $max\{B, Nz\left(S\right)\}$, then it follows that the total number of coefficients transferred over the network is $O\left(Rmax\{B, Nz\left(S\right)\}\right)$. In order to merge the results, the reducer makes a linear pass over the collected sorted lists. Therefore, its complexity is expected to be the same with the communication cost of the job. □

### 3.3.3 Speeding up the Distributed Greedy Solution

While the DGreedyAbs algorithm succeeds in offering a viable solution to the problem, it suffers from a basic drawback: There are multiple distributed jobs that may be required to create the synopsis, and thus, the centralized algorithm needs to run multiple times over the same data. Since we do not know in advance which are the appropriate nodes to retain from the root sub-tree, we run GreedyAbs for many possible $C_{root}$ sets, incurring extra computational and communication overhead.

In order to alleviate this overhead, in this Section we propose BUDGreedyAbs: a modified, bottom-up version of DGreedyAbs that makes only one pass over the dataset and executes the centralized greedy algorithm only once per sub-tree.

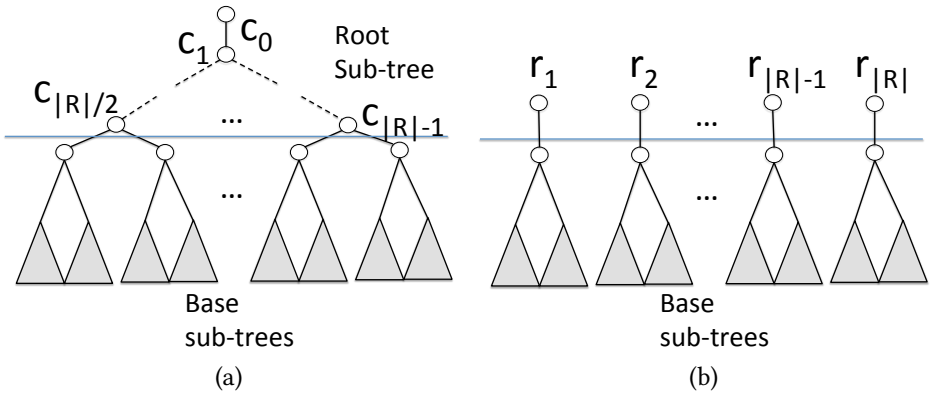



Figure 3.7: Equivalent representations of an error-tree.

In order to avoid the multiple jobs, consider the following strategy. Assume that the size of the root sub-tree is less than $B$ and we a-priori decide to retain it all in the synopsis and then run GreedyAbs to all workers in parallel. One could say that this is the safest choice as we keep a maximal amount of information about the part of the tree that creates dependencies among partitions. However, this is not optimal. Some nodes of the root sub-tree may be of negligible importance and by keeping them, we sacrifice budget space that could be allocated in a smarter way. For example, assume that the harder sub-tree to approximate in Figure 3.5 is $T_i$ and that the blue-filled nodes of the root sub-tree have an absolute value close to zero. Instead of keeping these two nodes, it might be preferable to keep two nodes from within $T_i$. The basic idea behind BUDGreedyAbs is to start from the safest choice of keeping all of the root sub-tree and adaptively refining it.

We start with a Lemma that follows directly from the properties of the wavelet transform. The idea of Lemma 6 is also presented graphically in Figure 3.7.

**Lemma 6.** *An error-tree partitioned to one root-subtree and many base subtrees $S_i, i = 1, .., R$, is equivalent to $R$ independent error-trees $S'_i, i = 1, .., R$, where each $S'_i = S_i$ with an extra coefficient $r_i$ as root. The value of $r_i$ is defined as: $r_i = \Sigma_{c_j \in path_{S_i}} \delta_{ij} \cdot c_j$ and is also equal to the average of all data values in base-subtree $S_i$.*

According to Lemma 6, instead of computing the full wavelet transform of the error-tree, we can compute the transform up to the height of the base-subtrees and also keep the local root-average of each subtree. That is what BUDGreedyAbs does. It first computes a wavelet structure as the one of Figure 3.7b. Then, it triggers a parallel execution of GreedyAbs at each base-subtree. The outputs are merged in the same way as in Section 3.3.2 and a maximum error is computed. As the yielded synopsis may contain some of the $r_i$ coefficients, in order to better exploit the available space budget, in a next step the algorithm examines opportunities for further

compression and computes the root-subtree solely based on these $r_i$ coefficients contained in the synopsis.

As the first part of the algorithm is the same with DGreedyAbs, we discuss the algorithmic details of merging and how more accurate configurations for the root-subtree are explored. For explaining these details, we give the following example:

**Example**. In Figure 3.8 we present two base-subtrees $S_1, S_2$. The lists $L_1, L_2$ show the most important coefficients from the corresponding outputs of GreedyAbs. Thus, in subtree 1, the last nine coefficients that the algorithm would delete are the ones in the array $L_1$ with $c_{a8}$ discarded first. As we have said, in order to create the final synopsis, we need to merge $L_1$ and $L_2$ from left to right and examine the errors of the first $B$ coefficients. Let us assume $B = 16$. The first nine coefficients of the synopsis would be $c_{a1}, ..., c_{a4}$ and $c_{b1}, ..., c_{b5}$. At this point, we check the $r_i$ coefficients. Instead of keeping both of them and waste two slots in the synopsis, we examine the possible merits of increasing compression in the root-subtree. We calculate the wavelet transform of the root-subtree considering as data values the $r_i, i = 1, 2$ coefficients. In our example of Figure 3.8, the transform results in the creation of $c_0$ and $c_1$. According to Lemma 6, keeping both $c_0$ and $c_1$ is completely equivalent to keeping $r_1$ and $r_2$ both in terms of appoximation quality and space overhead. Thus, we also examine the chance of keeping only $c_0$ or $c_1$ or even none of them. In order to preserve correctness, as GreedyAbs has run at each subtree considering all $r_i$ nodes retained in the synopsis, the posterior deletion of coefficients from the root-subtree should be accompanied by some error updates. Let us assume that we first examine the deletion of node $c_1$. Some of the nodes in $L_1$ and $L_2$ should update their observed signed errors by $-c_1$.

But which nodes need to be updated? The errors which are reported by the coefficients in the red box, i.e., the ones in the right side of $r_1$ in Figure 3.8, are calculated taking into account that $r_1$ is retained in the synopsis. Thus, a deletion of a node that contributes to the $r_1$ value must be reflected to the errors observed by these nodes. On the other hand, the nodes in the left side of $r_1$ consider $r_1$ already discarded and as such, nothing more is needed to be done on them.

Which nodes of the root-subtree should we consider to delete? Do we have to try all $R$ nodes in all possible combinations? We treat this issue the same way as we did for DGreedyAbs. We run GreedyAbs on the root-subtree and then execute Algorithm 5. The output of Algorithm 5 represents the candidate combinations for deletion. For each of them, we update the errors in $L_i$ lists and merge them in a final list where the $B$ first nodes are considered for the synopsis. BUDGreedyAbs is formally presented in Algorithm 7.

**Complexity Analysis**. The complexity of the parallel workers of BUDGreedyAbs is the same with that of GreedyAbs, i.e., $O\left(Nlog^2N\right)$. However, in the next stage of BUDGreedyAbs, as we have seen there are $R$ merge operations that take place and in the worst case, each of them needs to process $B$ elements. Furthermore, we also need to run GreedyAbs once on

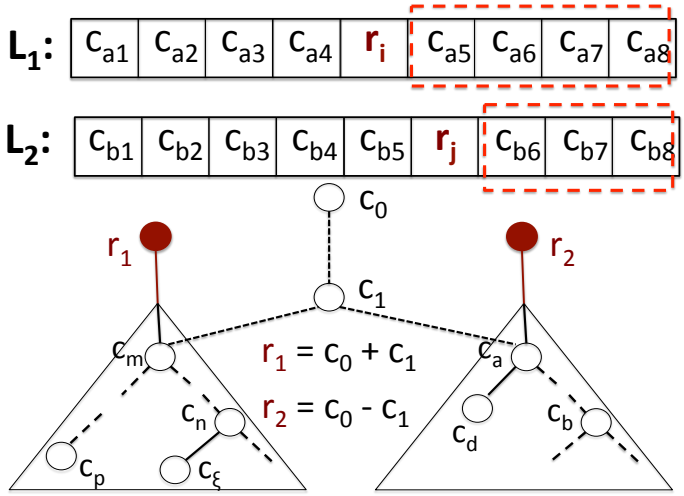



Figure 3.8: Example of merging solutions for BUDGreedyAbs.

the root-subtree. Thus, the complexity of the reduce workers that derive the final synopsis is $O\left(Rlog^2R + RB\right)$.

**Algorithm 7:** BUDGreedyAbs

**Require:** error-tree, space budget $B$

1: **for all** $T_i \in$ base subtrees in parallel **do**
2: $L_i = GreedyAbs\left(T_i\right)$; emit $L_i$
3: $L = merge(L_i$ lists)
4: $synopsis$ =first $B$ elements of $L$; $error = max\_abs(synopsis)$
5: $RA = \{r_i | L\left[j\right] = r_i \wedge 0 \leq j < B\}$ // $r_i$: avg of data values in $T_i$
6: Root-subtree= $WaveletTransform\left(RA\right)$
7: $\Gamma$ = $genRootSets$(Root-subtree,$B$)
8: **for all** $C_{root} \in \Gamma$ **do**
9: **for all** $L_i$ **do**
10: **if** $L_i\left[j\right] = r_i$ **then** update errors at $L_i\left[k\right], k > j$
11: $L = merge(L_j$ lists)
12: **if** $max_abs($first $B$ elements of $L) < error$ **then**
13: $synopsis$ =first $B$ elements of $L$
14: $error = max\_abs(synopsis)$
15: **return** $synopsis$

### 3.3.4 Maximum Relative Error

Minimizing the maximum relative error is arguably more essential compared to absolute error minimization in approximate query processing, as the same absolute error in two different data values may express huge differences in relative error. At the same time, relative error measures

tend to be inordinately dominated by small data values. For instance, returning 2 as the approximate answer for 1 amounts to an 100% relative error, while in fact it is insignificant in a data context dominated by much larger values. In order to overcome such problems, several techniques have been developed for combining absolute and relative error metrics [VW99]. As in earlier approaches ([GG02], [GK04]), we have opted for the relative error metric with a sanity-bound S > 0. Our aim is to produce wavelet synopses in near-linear time and space such that, for each approximation $\hat{d}_i$ of a data value $d_i$, the ratio is kept lower than a feasible bound.

For this problem, in [KM05] the GreedyRel algorithm is presented. GreedyRel follows the greedy paradigm introduced in Section 3.3.1, wherein, instead of using $MA_k$, it chooses to discard the coefficient with the minimum *maximum potential relative* error, defined as follows:

$$MR_k = max_{d_j \in leaves_k}\{\frac{|err_j - \delta_{jk} \cdot c_k|}{max\left(|d_j|, S\right)}\} \tag{3.6}$$

Nevertheless, the four error quantities of Equation 3.5 cannot be used for the calculation or update of the $MR_k$. The reason is the denominator in Equation 3.6, which implies that the effect a coefficient $c_k$ is different in the signed relative error of different data values.

In order to provide a scalable solution to this problem, we use a similar approach with that of DGreedyAbs, but instead of using GreedyAbs at the workers, we use GreedyRel.

## 3.4 CON: Constructing the $L_2$ Synopsis in Parallel

For evaluating the efficiency of the proposed partitioning scheme, I also employ it for constructing the conventional, $L_2$-optimal wavelet synopsis. Then, the results are compared with the ones achieved by the approach used in [JYL11] The algorithms of [JYL11] can be found in Appendix B. In order to compute the conventional synopsis in parallel, we partition the data as described in Section 3.2 (see Figure 3.2). Each mapper reads a portion of the input in the size of a power of two and locally constructs the corresponding sub-tree by pairwise averaging and differencing coefficients, as explained in Chapter 2. As the wavelet transform is of linear complexity and the mapper computes the coefficients only for its local data, the computational complexity of each map task is $O\left(S\right)$. After the construction of the sub-trees is over, each mapper emits all the computed coefficients to the reduce stage. Thus, the communication between the map and reduce phase is $O\left(N\right)$. The reducer reads all the coefficients that are computed in the map phase and inserts them in a priority queue, where only the $B$ largest ones in absolute normalized value are retained. It also computes the wavelet coefficients of the root sub-tree and inserts them in the queue as well. When this process is over, there are $B$ coefficients in the queue which comprise the conventional synopsis. This approach assumes that $B$ coefficients can fit in main memory which is a logical assumption to make.

## 3.5 Experimental Evaluation

In this Section, there is an evaluation of the proposed algorithms in terms of (i) synopsis construction time and (ii) achieved error. The results show that the proposed distributed solutions present linear scalability and we are able to run experiments on bigger datasets than any previous work. All algorithms are implemented in Java 1.8.

**Datasets**. The experiments are conducted using both synthetic and real datasets. Synthetic data (SYN) allows easy testing over different data distributions and value ranges. Distributions utilized are uniform and zipfian (with exponents 0.7 and 1.5). Data values lie between [0, 1000]. For real-life datasets we utilize NYCT [nyc] and WD [lin]. NYCT describes taxi trips in the New York City and contains records for the trip time in seconds. WD consists of observations on wind direction (azimuth degrees) captured during hurricanes in the USA. Table 3.1 gives an overview of NYCT and WD. All datasets are partitioned in order to test scalability over different sizes. The smallest partition comprises the first $1M$ records, while each subsequent partition is $2\times$ the previous one. The largest used dataset consists of $268M$ datapoints.

Table 3.1: Characteristics of NYCT and WD datasets

| Name | #Records | Avg | Stdv | Max |
|---|---|---|---|---|
| NYCT2M | 2M | 672 | 483 | 10800 |
| NYCT4M | 4M | 511 | 519.5 | 10800 |
| NYCT8M | 8M | 255 | 646.6 | 10800 |
| NYCT16M | 16M | 127 | 745 | 10800 |
| NYCT32M | 32M | 63 | 3566.3 | 4293410 |
| NYCT64M | 64M | 31 | 25410.3 | 4294966 |
| WD2M | 2M | 121 | 119.7 | 655 |
| WD4M | 4M | 122 | 119.9 | 655 |
| WD8M | 8M | 138 | 119.4 | 655 |
| WD16M | 16M | 127 | 118.8 | 655 |

**Platform setup**. As a deployment platform, a Hadoop 2.6.5 cluster of 9 machines has been used. Each of the 9 machines features eight Intel Xeon CPU E5405 @ 2.00GHz cores and 8 GB of main memory. One machine is used as the master node and the remaining ones as slaves. Each slave is allowed to run simultaneously up to 5 map tasks and 1 reduce task. Each of these tasks is assigned 1 physical core and 1 GB of main memory. For all the remaining properties, the default Hadoop configuration has been kept.

For experimenting with the centralized algorithms, one machine with the same specifications as the ones listed above has been used. Thus, centralized algorithms may have up to 8 GB of available main memory for their execution.

### 3.5.1 Scalability

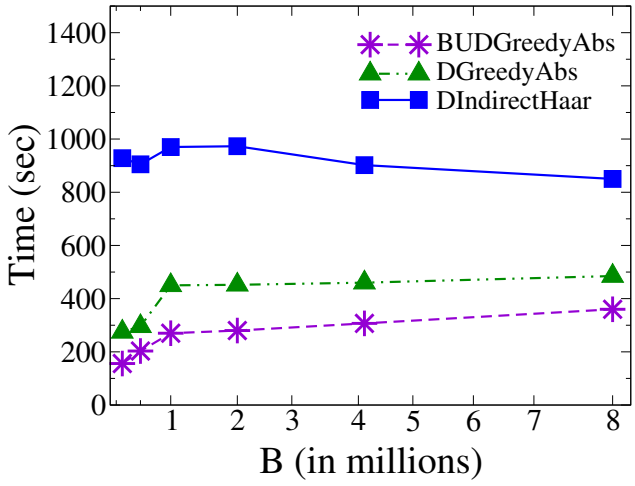



Figure 3.9: Scalability with $B$

In this Section, synthetic data is used to assess the scalability with respect to the available budget for the synopsis $B$, the number of datapoints $N$ and the number of tasks running in parallel. The results show that the algorithms proposed in this dissertation can scale to data sizes that state-of-the-art centralized approaches are incapable of. For all the experiments of this Section, data consists of uniformly distributed values in the range of $[0, 1K]$.

**Varying space budget**. In this experiment I examine the scalability with respect to the space budget $B$. I run DGreedyAbs, BUDGreedyAbs and DIndirectHaar for one-dimensional data of size $N = 17M$ and vary $B$ from $N/64$ to $N/2$. The results of Figure 3.9 show that for DGreedyAbs, running-time is not considerably affected by the size of the synopsis. However, this is not true for DIndirectHaar and BUDGreedyAbs. For DIndirectHaar, a larger $B$ is more probable to lead in a smaller error and decrease the $\frac{\epsilon}{\delta}$ factor of its complexity formula. Thus, a larger budget may lead to faster execution of the algorithm. For BUDGreedyAbs, as the complexity of the reducer is $O\left(Rlog^2R + RB\right)$, running-time can linearly increase with $B$.

**Varying datasize and number of parallel tasks**. Figure 3.10 shows the scalability with respect to the number of datapoints ($N$) and tasks running in parallel for DIndirectHaar, DGreedyAbs and BUDGreedyAbs respectively. We set $B = 1M$ for all the experiments of this subsection and vary the datasize from 2M to 268M datapoints for all the algorithms and the number of parallel map tasks from 10 to 40. Both algorithms are also compared with the corresponding centralized implementations in order to assess the difference in performance. Please note that the y-axis in Figures 3.10-(a), 3.10-(b) and 3.10-(c) follows a logarithmic scale.

All the algorithms scale linearly with the dataset size. The running-time is almost constant at first, when all data can be processed fully in parallel, and is linearly growing as the cluster is fully

utilized and more tasks need to be serialized for execution. Linear scalability is also observed with the number of parallel running tasks. By halving the capacity of the cluster, running-time is almost doubled for all the examined algorithms.

The centralized algorithms were not able to run for datasizes greater than $17M$ datapoints, as their execution demands more than the available main memory. Compared to the centralized GreedyAbs, BUDGreedyAbs appears to be $20\times$ faster for a dataset of 17M datapoints when all of its map tasks can run fully in parallel. In Figures 3.10-(b), 3.10-(c) we also observe that BUDGreedyAbs is twice as fast as DGreedyAbs. This is because DGreedyAbs needed to try two $C_{root}$ sets in order to converge, while BUDGreedyAbs always needs a single MapReduce job. As we notice in Figure 3.10-(a), even if DIndirectHaar scales linearly, it is slower than the greedy algorithms, being $1.5\times$ and $3\times$ slower than DGreedyAbs and BUDGreedyAbs respectively. Moreover, we see that the centralized IndirectHaar is faster than DIndirectHaar when the dataset size is small or few parallel tasks are running. That is because the centralized implementation loads the whole dataset in memory and the required multiple jobs do not need to perform I/O operations. On the other hand, the Hadoop implementation is disk-based and for each job, the algorithm has to read the input and write the output from and to the HDFS respectively.

The main results of this Section are that: (i) all distributed algorithms scale linearly with the datasize, and (ii) greedy algorithms are much faster than the state-of-the-art DP.

### 3.5.2 Comparison for Real Datasets

In this Section, DGreedyAbs, BUDGreedyAbs and DIndirectHaar are compared with each other, as well as with their centralized counterparts using real-life one-dimensional datasets. Furthermore, they are also compared against CON (Section 3.4 ). As CON is less compute-intensive, I wanted to investigate the tradeoffs in running-time and produced maximum error. Note that IndirectHaar is not included in the approximation quality experiments, as it theoretically achieves the same results as DIndirectHaar.

**NYCT dataset**. In Figure 3.11a, the approximation quality results for the NYCT dataset are presented. The utilized space budget is $B = \frac{N}{8}$. The construction of an accurate synopsis for this dataset is a difficult task to accomplish as it contains values of high magnitude and variance. Two important observations are that: (i) scalability does not come at a cost; the distributed greedy algorithms achieve the same error with GreedyAbs and (ii) all algorithms targeting maximum error metrics outperform CON from 2 to 5 times.

Figure 3.11b presents the running-time results for the same dataset. With the maximum absolute error over 550 for all datasizes, the multiplicative factor $\left(\frac{\mathcal{E}}{\delta}\right)^2$ of the complexity formula of DIndirectHaar is equal to 121. As such, for this dataset, the execution of the DP algorithms

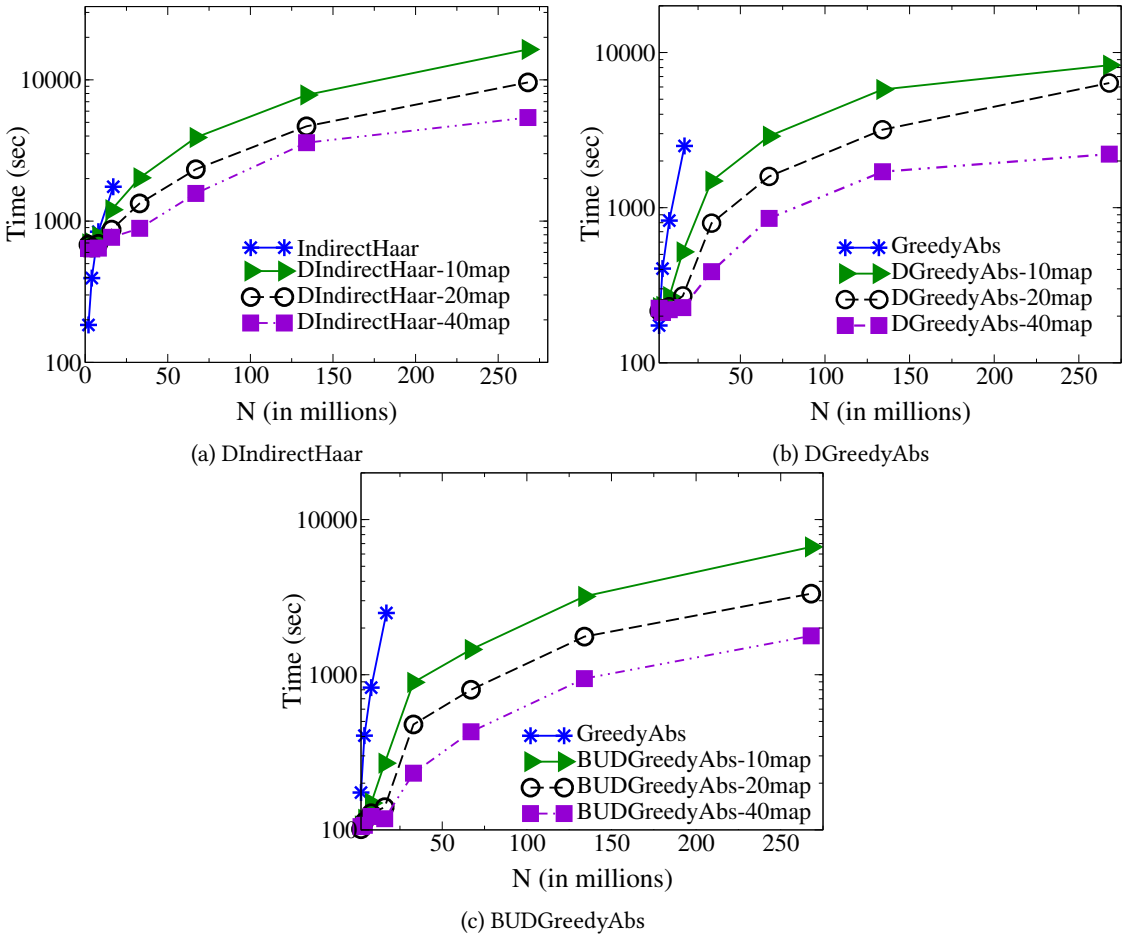
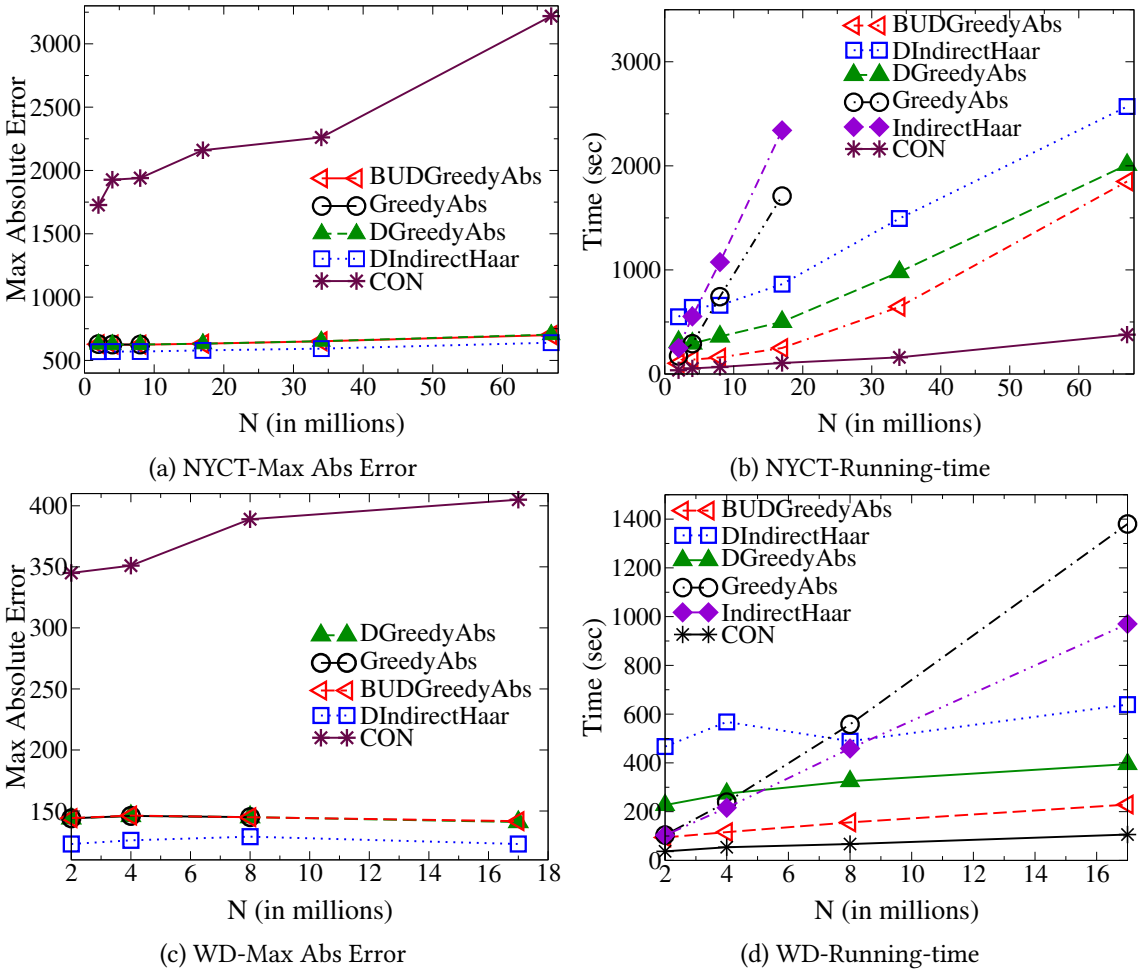



Figure 3.10: Scalability with the dataset size (N) and number of parallel tasks.

is very compute-intensive. We observe that for datasizes smaller than 60M datapoints, BUDGreedyAbs is the most time-efficient algorithm among the ones that target maximum error metrics. In Section 3.5.1, we said that the available budget affects the running-time performance of BUDGreedyAbs. Since we have set $B = N/8$, an increase in the number of datapoints implies an increase in the synopsis' size which in turn increases the running-time of the algorithm. At this point, we observe a trade-off between DGreedyAbs and BUDGreedyAbs. On the one hand, DGreedyAbs needs multiple passes over the data in the map phase of the job, while BUDGreedyAbs needs only one. On the other hand, DGreedyAbs has a lightweight reducer, while the one of BUDGreedyAbs is compute-intensive and can become a bottleneck. Thus, when datasize is large, we suggest BUDGreedyAbs for datasets that can be easily approximated with a small available budget and DGreedyAbs when a higher budget is demanded. As the conventional synopsis is easier to be computed, we observe CON to be much faster than all the other algorithms.




(a) NYCT-Max Abs Error

(b) NYCT-Running-time

(c) WD-Max Abs Error

(d) WD-Running-time

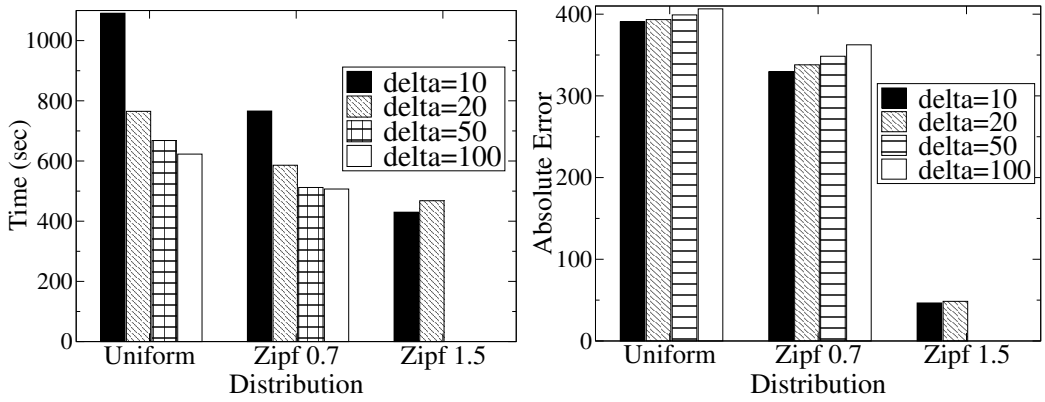
Figure 3.11: Approximation quality and running-time experiments on 1-D real datasets. $B = N/8$

**WD dataset**. Figure 3.11c shows the approximation quality and running-time results for the WD dataset and $B = \frac{N}{8}$. The conclusions are similar to the ones for the NYCT dataset. In Figure 3.11d we see that IndirectHaar outperforms DIndirectHaar for datasizes up to 8M data-points. When data fits in main memory, IndirectHaar avoids the I/O overhead of the multiple MapReduce jobs, that DIndirectHaar requires. Still, the most efficient algorithm, that targets the minimization of maximum error metrics, is BUDGreedyAbs as it outperforms GreedyAbs by a factor of 6.7 and DGreedyAbs by a factor of 2 for a $17M$ dataset.

### 3.5.3 Dataset Impact

In this subsection, synthetic data is used to evaluate the impact of different distributions on both running-time and approximation quality. For all the experiments of this subsection, there have been used datasets of size $N = 17M$ datapoints and a budget of $B = N/8$.

**Varying distribution and $\delta$.** As the parameter $\delta$ of DIndirectHaar provides a "knob" for tuning the tradeoff between resource requirements and solution quality, in Figure 3.12 I show the impact of data distribution on DIndirectHaar when different $\delta$-values are used. The main observation is that biased distributions favor both the synopsis construction time and the approximation quality [PZHM09]. In Figure 3.12a, we see that for the Zipf-0.7 distribution and for all $\delta$-values, the algorithm is about 25% faster compared to the Uniform distribution. Furthermore, the run for the Zipf-1.5 distribution outperforms the one for Zipf-0.7 by 45% when $\delta = 10$ and 20% when $\delta = 20$. Accordingly, in Figure 3.12b we see that when the Zipf-1.5 distribution is the case, the maximum absolute error is 8.4 times smaller than the one achieved for the Uniform data.



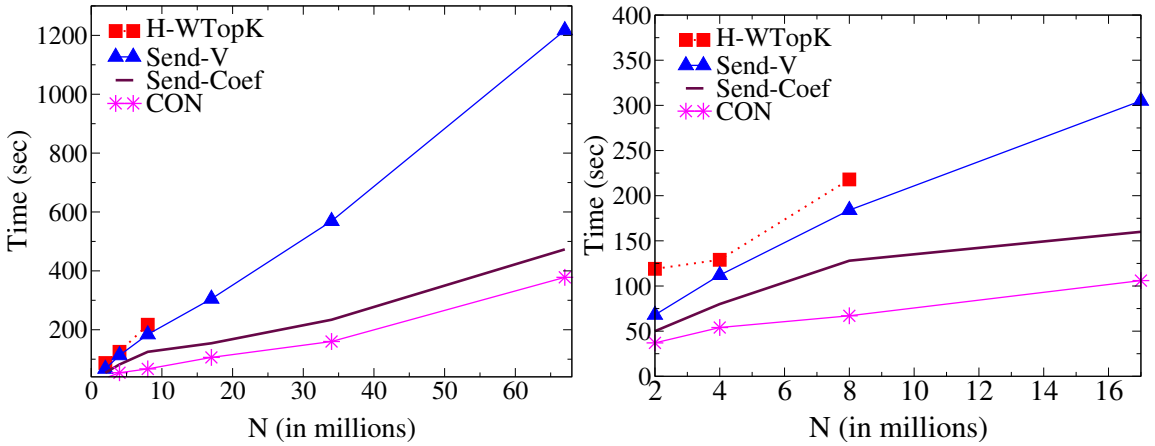

(a) Running-time (b) Approximation quality

Figure 3.12: Impact of data distribution and $\delta$ on the performance and approximation quality of DIndirectHaar.

We also observe that usually the smaller $\delta$ is, the higher is the running-time of the algorithm and the better the approximation quality, since more candidate values are examined for the incoming values and the wavelet coefficients. For values of $\delta$ equal to 50 or 100, the algorithm reaches its lower bound of execution time on this data and thus, higher values for $\delta$ do not affect performance. For the Zipf-1.5, we see that the run for $\delta = 10$ outperforms the one for $\delta = 20$. As we get more approximate results for higher values of $\delta$, DIndirectHaar requires more jobs to converge and provide the final answer. Moreover, the algorithm could not run for Zipf-1.5 and $\delta = 50, 100$ as these values were higher than the space they need to quantize.

### 3.5.4 Constructing the Conventional Synopsis

In this Section, the construction of conventional synopses is evaluated. For the evaluation, the real datasets NYCT and WD have been used and the cluster is configured with 20 map and 1 reduce slots. For any given dataset, all four described algorithms (CON, Send-V, Send-Coef, H-WTopk) produce exactly the same synopses. Thus, there is no need to compare them in terms of approximation quality but only with respect to running time.




(a) NYCT (b) WD

Figure 3.13: Running time comparison for constructing a conventional synopsis with $B = N/8$.

Figure 3.13 shows the running time results for both datasets when a synopsis of size $B = N/8$ is requested. Since Send-V ends up to be a sequential algorithm, it presents much worse running time performance than CON and Send-Coef for both examined datasets.

In Figure 3.13, we also observe CON is the most time-efficient algorithm for computing the conventional synopsis. The performance gain of CON stems from its locality-preserving partitioning, which results in less computational and communication complexity. CON is $1.5\times$ faster on average than Send-Coef, that is the second most efficient algorithm, both for the NYCT and the WD datasets.

For both datasets, we observe that despite the communication optimizations, H-WTopk presents the worst performance. Furthermore, for datasizes larger than 8 millions of datapoints, it runs out of memory. This is because of the selected synopsis size. H-WTopk can be very efficient if $B$ is much smaller than the input size of the mapper. Otherwise, since it needs to emit the $B$ largest and $B$ smallest coefficients, it ends up emitting twice the input size. Furthermore, it also has the extra overhead of three MapReduce jobs. In [JYL11], the wavelet transform was applied to a histogram and thus, data had been already compacted and smaller budget space was needed to achieve accurate results. The impact of $B$ in the communication cost is discussed in [JYL11], where the corresponding values were only chosen in the range $[10, 50]$.

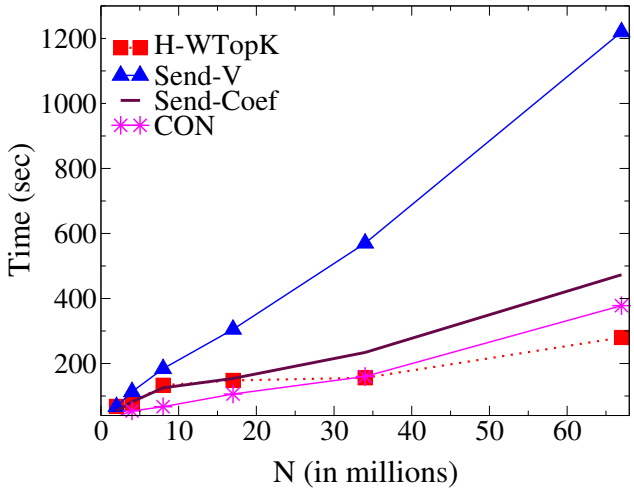



Figure 3.14: Running time results for the NYCT dataset and B=50.

Table 3.2: Testbed details

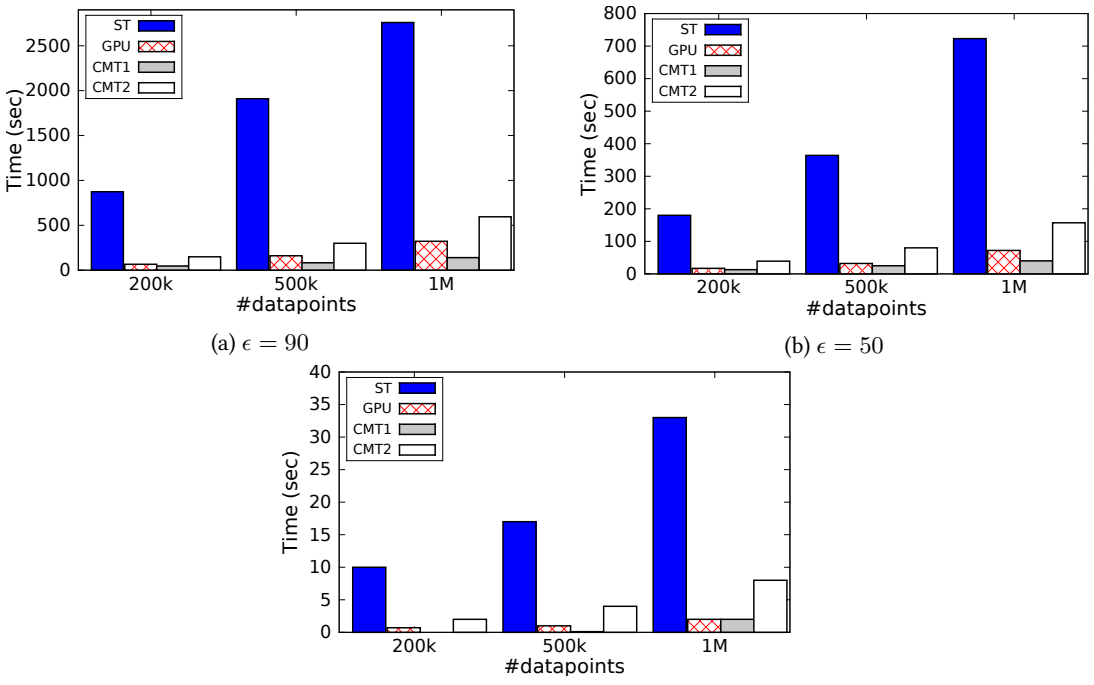
| **Name** | **Model** | **Processor** | **Memory (GB)** |
|---|---|---|---|
| High-end server (CMT1) | Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz | 2 sockets, 14 hyper-threaded cores/socket | 256 |
| Commodity server (CMT2) | Intel(R) Core(TM) i7-4820K CPU @ 3.70GHz | 1 socket, 4 hyper-threaded cores/socket | 64 |
| GPU | NVIDIA Tesla V100-SXM2 | $\#SM = 80$ | 32 |

Figure 3.14 shows the corresponding results for the NYCT dataset when a synopsis of stable size $B = 50$ is used. This figure verifies the results of [JYL11]: H-WTopk dominates the other approaches only when $B$ is very small and the dataset size large enough to not be affected by the overhead of the three MapReduce jobs. Thus, in our case, where the transform is applied directly on the data and not on a histogram, this algorithm is not of practical use as it is very difficult to construct a good quality synopsis with so few coefficients.

### 3.5.5 Evaluating Accelerators

In this Section, I experimentally evaluate the performance gains obtained by parallelizing the construction of a wavelet synopsis. The experiments use synthetic one-dimensional data uniformly distributed in the range $[0 - 1K]$.

**Hardware**. As OpenCL requires devices of massive parallelism, for these experiments different hardware platforms than before have been used. We consider three different architectures: (i) a high-end server (CMT1), (ii) a commodity server (CMT2) and (iii) a GPU device. Details




(a) $\epsilon = 90$

(b) $\epsilon = 50$

(c) $\epsilon = 10$

Figure 3.15: Running time of MinHaarSpace for various input sizes and values of $\epsilon$.

for the testbed can be found in Table 3.2. Henceforth, I denote CMT1 (CPU MultiThreaded) the high-end server and CMT2 the commodity one. As a baseline for the experiments we consider the single-thread (ST) performance of CMT1. For maximizing the utilization of each platform, the multi-threaded executions have available all the threads of a machine. Work group size is set to 256. I also experimented with 512 and 1024 which was the maximum value supported by the device but there was no significant difference noticed in performance.

Figure 3.15 demonstrates running times for different dataset sizes. In MinHaarSpace, the combination of $\epsilon$, $\delta$ parameters determines the size of a single row of the DP matrix and thus, the running time complexity of the algorithm. For experimenting with tasks of different compute intensity, I test various sizes for the DP rows by varying $\epsilon$ and keeping $\delta$ fixed to 0.1.

For all configurations, running time scales linearly to the dataset. This verifies the theoretical complexity of the algorithm which is $O\left(\left(\frac{\epsilon}{\delta}\right)^2 N\right)$. According to this formula and as can be verified in Figure 3.15, high values of $\epsilon$ lead to higher running times. The high-end server CMT1 can reach a speedup of up to $23\times$ compared to the baseline. The corresponding result for the GPU is $15\times$, while CMT2 achieves a more modest performance improvement of $5\times$.

As explained in Section 3.2.1, due to the structure of the error-tree, at each level of execution within a work group, half of the parallelism is lost. Thus, a device that can offer massive data parallelization (e.g., GPU) cannot be exploited to the full of its potential. Even in that case though,

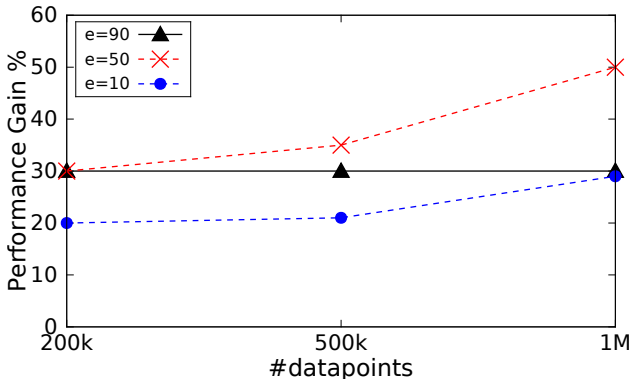



Figure 3.16: Performance gain due to memory coalescing.

the execution over the GPU outperforms the one of CMT2 by more than twice in all examined cases.

We also notice that the smaller the dataset is, the smaller is the performance gap between CMT1 and the GPU. For the compute intensive case of Figure 3.15-(a), CMT1 is 30% faster than the GPU for a dataset of $200K$ datapoints, $49\%$ for $500k$ datapoints and $57\%$ faster when $1M$ of datapoints is the case. Data transfers cause significant delays in the GPU case. Reducing I/O has a direct impact on running-time.

In Figure 3.16 we observe the performance gain in GPU execution when memory coalescing is used. I first run the experiments employing exactly the same kernel as in the CPU case and then repeat them, this time using the kernel that makes memory coalescing and report the relative percentile difference. We notice that the gains stemming from memory coalescing are always higher than $20\%$ and can reach up to $50\%$.

CHAPTER 4

# Extension to Multiple Dimensions

## 4.1 Introduction

The algorithms discussed so far are applicable on one-dimensional datasets. However, datasets with multiple dimensions involved are a common case in real-life applications. In this Chapter, I discuss the modifications required in order to extend both the centralized GreedyAbs and MinHaarSpace [KSM07] to deal with multiple dimensions. The main difference is that now the distributed algorithms of Chapter 3 run over a multidimensional error-tree and instead of using GreedyAbs and MinHaarSpace, they employ the modified algorithms we are going to present in this Chapter.

The structure of a D-dimensional error-tree (Figure 2.3) is somewhat more complex. As opposed to the one-dimensional case, each node of the tree contains many coefficients and thus the terms *node* and *coefficient* should be distinguished. During thresholding, it is not necessary to pick or discard all coefficients of a node at the same time. In Figure 4.1 we see a snapshot where the black-filled coefficients are retained in the synopsis, while the blank ones are discarded. For the navigation in a multidimensional error-tree, we follow the indexing presented in Figure 2.3. The first node at each level has index $2^{D \cdot level}$. The rest of the nodes of the same level maintain index values increased by one each. For our example and the two-dimensional case, in level 1, the first node has index $2^{2 \cdot 1} = 4$, the next node of this level has index 5 and the remaining two have indices 6 and 7 respectively. With that indexing scheme, we can easily navigate the error-tree. Dividing a node's index by $2^D$ leads us to the parent of the node. For the identification

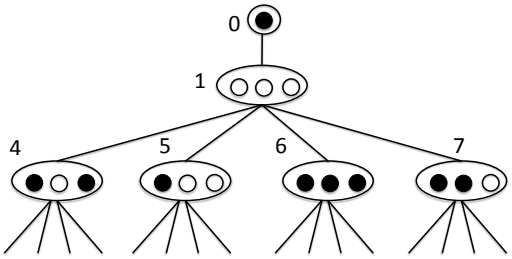



Figure 4.1: Thresholding in a 2-dimensional error-tree.

of an individual coefficient within a node, we apply internal indexing. The internal index of a coefficient $c_{ij}$ belongs in the interval: $[0, 2^D - 1)$. The notation $c_{ij}$ denotes the $j$-th coefficient in node $i$.

## 4.2 MDMSpace: MinHaarSpace for Multiple Dimensions

In order to explain the required modifications for extending MinHaarSpace [KSM07] to multiple dimensions, an understanding of the original algorithm is required. A detailed description of MinHaarSpace can be found in Appendix A. Since the algorithm works with unrestricted wavelets (Section 2.5), Lemma 17 in Appendix A bounds the space we have to explore for selecting coefficient values in the one-dimensional case. For extending this Lemma to multiple dimensions, we have first to understand the basic idea which is based on. The following proof provides an intuition into it.

*Proof.* Let us consider a wavelet node $c_i$, with real coefficient value $z_i$ and incoming value[1] $v_i$. Then, the incoming value to its left child will be $v_{2i} = v_i + z_i$. With $v_i^*$ and $z_i^*$ we denote the incoming value and wavelet coefficient in the unrestricted case, where $z_i^*$ does not come from the Haar wavelet transform but is selected from $\mathbb{R}$. According to Lemma 16 we have:

$$|v_{2i} - v_{2i}^*| \leq \epsilon \Rightarrow |(v_i + z_i) - (v_i^* + z_i^*)| \leq \epsilon \Rightarrow |(v_i - v_i^*) + (z_i - z_i^*)| \leq \epsilon \tag{4.1}$$

If we choose $z_i^*$ values such that $|(z_i^* - z_i^*)| \leq \epsilon - |v_i - v_i^*|$, then the triangular inequality gives us:

$$|(v_i - v_i^*) + (z_i - z_i^*)| \leq |v_i - v_i^*| + |z_i - z_i^*| \leq |v_i - v_i^*| + \epsilon - |v_i - v_i^*| = \epsilon \tag{4.2}$$

which holds according to Inequality 4.1. Thus, the inequality $|(z_i^* - z_i^*)| \leq \epsilon - |v_i - v_i^*|$ appropriately delimits the $z_i^*$ values without violating the constraints of the problem. □

[1]For the notions of incoming value and real wavelet coefficient, look at the MinHaar description in Appendix A

The inequality of Lemma 16 is also satisfied in the multidimensional case and implies that the finite set of possible incoming values we have to examine at node $c_i$ consists of the multiples of $\delta$ in the interval $S_i = [v_i - \epsilon, v_i + \epsilon]$; thus, $|S_i| = \lfloor \frac{2\epsilon}{\delta} \rfloor + 1 = O\left(\frac{\epsilon}{\delta}\right)$.

For bounding the search space for coefficients in multidimensional datasets, we extend the above idea and present the following Lemma:

**Lemma 7.** *Let $v_i$ be the real incoming value to node $c_i$, $z_{i,k}$ the real assigned coefficients at $c_i$, $v_i^* \in S_i$ be a possible incoming value to $c_i$ for which the maximum error bound $\epsilon$ can be satisfied, and $z_{i,k}^*$ be a value that can be assigned for incoming value $v_i^*$ satisfying $\epsilon$. Then, $|z_{i,k} - z_{i,k}^*| \leq \frac{\epsilon - |v_i - v_i^*|}{2^D - 1}$*

*Proof.* A node $c_i$ of a D-dimensional error-tree contains $2^D - 1$ coefficients $z_{i,j}$ that all contribute to its $2^D$ children. Thus, for an incoming value $v_i$ at node $c_i$, the incoming value at its $j$-th child is: $v_{i \cdot 2^D + j} = v_i + \sum_{k=0}^{2^D-2} (z_{i,k} sign\,(k,j))$. Following a similar reasoning as before, we have:

$$|v_{i \cdot 2^D + j} - v^*_{i \cdot 2^D + j}| \leq \epsilon \Rightarrow |(v_i - v_i^*) + \sum_{k=0}^{2^D-2} \left(z_{i,k} - z_{i,k}^*\right) sign\,(k,j)\,| \leq \epsilon \tag{4.3}$$

If we choose $z_{i,k}^*$ values such that: $|z_{i,k} - z_{i,k}^*| \leq \frac{\epsilon - |v_i - v_i^*|}{2^D - 1}$, then from the triangular inequality, we have:

$$|(v_i - v_i^*) + \sum_{k=0}^{2^D-2} \left(z_{i,k} - z_{i,k}^*\right) sign\,(k,j)\,| \leq |v_i - v_i^*| + \sum_{k=0}^{2^D-2} |z_{i,k} - z_{i,k}^*| \leq$$
$$|v_i - v_i^*| + (2^D - 1) \cdot \frac{\epsilon - |v_i - v_i^*|}{2^D - 1} = \epsilon \tag{4.4}$$

which holds according to Inequality 4.3 and thus $|z_{i,k} - z_{i,k}^*| \leq \frac{\epsilon - |v_i - v_i^*|}{2^D - 1}$ effectively delimits the space of candidate values for the wavelet coefficients. □

For a given incoming value $v$ at node $c_i$, the possible assigned values for every coefficient $z_{i,k}$, $k = 0, .., 2^D - 2$ comprise the finite set of the multiples of $\delta$ in the interval: $S_{i,k}^v = \left[z_{i.k} - \frac{\epsilon - |v_i - v_i^*|}{2^D - 1}, z_{i,k} + \frac{\epsilon - |v_i - v_i^*|}{2^D - 1}\right]$. As $|S_{i,k}^v| \leq \frac{2(\frac{\epsilon - |v_i - v_i^*|}{2^D - 1})}{\delta} = O\left(\frac{2}{2^D - 1} \frac{\epsilon}{\delta}\right)$, and each node contains $2^D - 1$ coefficients, the number of examined values for node $c_i$ is $O\left(\left(\frac{2}{2^D - 1} \frac{\epsilon}{\delta}\right)^{2^D - 1}\right)$.

Similarly to MinHaarSpace, the MDMSpace procedure works in a bottom-up left-to-right scan over the error-tree. At each visited node $c_i$ it calculates an array $A$ of size $|S_i|$ from the precalculated arrays of its children nodes. $A$ holds an entry $A[v]$ for each possible incoming value $v$ at $c_i$. Such an entry contains: (i) the minimum number $A[v].s = S(i, v)$ of non-zero coefficients that need to be retained in the sub-tree rooted at $c_i$ with incoming value $v$, so that the

resulting synopsis satisfies the error bound $\epsilon$, (ii) the $\delta$-optimal values $A[v].\left(z^v_{i,0},..,z^v_{i,2^D-2}\right)$ to assign at $c_i$, for incoming value $v$, and (iii) the actual minimized maximum error $A[v].e$ obtained in the scope of $c_i$. $S(i,v)$ is recursively expressed as:

$$S(i,v) = \min_{z_{i,k} \in S^v_{i,k}} \left( \sum_{j=i2^D}^{i2^{D+1}-1} S(j, v + \sum_{k=0}^{2^D-2} z_{i,k} sign(j,k)) + \sum_{k=0}^{2^D-2} (z_{i,k} \neq 0) \right)$$

$$S(0,0) = \min_{z_{0,0} \in S^0_{0,0}} (S(1, z_{0,0}) + (z_{0,0} \neq 0))$$

The above equations compute the smallest between (i) the minimum required space if a non-zero coefficient value $z_{i,k}$ is assigned at $c_{i,k}$; and (ii) the required space if a zero value is assigned at it. The latter case applies only if $0 \in S^v_{i,k}$. Let $S^v_{i,k}$ denote the set of those assigned values at $c_{i,k}$ for incoming value $v$ that require the minimum space in order to achieve the error bound $\epsilon$: The $\delta$-optimal value to select is the one among these candidates that also minimizes, in a secondary priority, the obtained maximum absolute error in the scope of $c_i$. So, we also need the equations:

$$E(i,v) = \min_{z_{i,k} \in S^v_{i,k}} \left( \max_{j=i2^D}^{i2^{D+1}-1} E(j, v + \sum_{k=0}^{2^D-2} z_{i,k} sign(j,k)) \right)$$

$$E(0,0) = \min_{z_{0,0} \in S^0_{0,0}} (E(1, z_{0,0}))$$

**Complexity Analysis**. The result array $A$ on each node $c_i$ holds $|S_i|$ entries, one for each possible incoming value, hence its size is $O\left(\frac{\epsilon}{\delta}\right)$. Moreover, at each node $c_i$ and for each $v \in S_i$, we loop through all $\prod_{k=0}^{2^D-2} |S^v_{i,k}| = O\left(\left(\frac{2}{2^D-1}\frac{\epsilon}{\delta}\right)^{2^D-1}\right)$ possible assigned values. Thus, the runtime of MDMSpace(0, $\epsilon$) is $O\left(\left(\frac{2}{2^D-1}\right)^{2^D-1} \cdot \left(\frac{\epsilon}{\delta}\right)^{2^D} N\right)$.

## 4.3 MGreedyAbs: Extending GreedyAbs to Multiple Dimensions

For extending the algorithms of Section 3.3 to multiple dimensions, we first need to modify the centralized GreedyAbs algorithm.

As in the one-dimensional case, the greedy algorithm picks each time the coefficient $c_{jk}$ with the lowest $MA$ and discards it. According to Equation 3.5, the computation of $MA_k$ for a node $c_k$ demanded four values ($max^l_k$, $min^l_k$, $max^r_k$, $min^r_k$): the maximum and minimum error for each of the two subtrees of $c_k$. A node of a D-dimensional error-tree has $2^D$ children. Thus, in order to compute $MA_{jk}$, we need to know the maximum and minimum error in each of the $2^D$ subtrees of $c_{jk}$, thus $2^{D+1}$ values are required. We can see that all the coefficients of a node

support the same region of the original data, and so they should observe the same errors in the reconstruction of the corresponding data values. In that way, we do not need to store at each coefficient the maximum and minimum error observed in each sub-tree, but all the coefficients of a node can share the same $2^{D+1}$ values. The equation for the computation of $MA_{jk}$ is:

$$MA_{jk} = \max_{0 \leq s \leq 2^D - 1} \{|max_j^s - sign\,(s)\, c_{jk}|, |min_j^s - sign\,(s)\, c_{jk}|\} \tag{4.5}$$

where $s$ is the index of each sub-tree of node $j$ and $sign\,(s)$ is the sign of the error caused in sub-tree $s$ when deleting coefficient $c_{jk}$. Similarly to the one-dimensional case, when a coefficient $c_{jk}$ is discarded, its maximum and minimum errors need to be updated, as well as the $MA$-values of all coefficients in the sub-trees of node $j$ and if needed the coefficients in the ancestors of node $j$. Furthermore, this time we also need to update the $MA$-values of the remaining coefficients in node $j$ that are not yet discarded. Algorithm 8 formally presents MGreedyAbs, the modified algorithm for handling multidimensional data.

**Complexity Analysis**. The initial heap $H$ can be constructed in $O(N)$ time. The algorithm performs $O(N)$ discarding operations. A dropped coefficient $c_{jk}$ at height $h$ of the error-tree has at most $2^{Dh}$ descendant nodes and each of them at most $2^D - 1$ non-deleted coefficients. Thus, each coefficient at height $h$ has at most $2^{Dh}(2^D - 1)$ non-deleted descendant coefficients which must be updated. Moreover, at height $h$ of the error-tree, there are $2^{D(log_{2^D} N - h)}(2^D - 1)$ coefficients. As all of them will eventually be discarded, the total number of updates in descendants for all coefficients is:

$$\sum_{h=1}^{log_{2^D} N} [2^{Dh}(2^D - 1) \cdot 2^{D(log_{2^D} N - h)}(2^D - 1)] = (2^D - 1)^2 N log_{2^D} N \tag{4.6}$$

However, as it holds $log_{2^D} x = \frac{1}{D} log x$, the above quantity becomes: $\frac{(2^D-1)^2}{D} N log N$. A discarded coefficient $c_{jk}$ has at most $log_{2^D} N$ ancestor nodes with at most $2^D - 1$ non-deleted coefficients each, and thus the total number of updates in ancestors for all deleted coefficients is also $O(\frac{(2^D-1)^2}{D} N log N)$. Furthermore, for each dropped coefficient $c_{jk}$, we need to update at most $2^D - 2$ coefficients in node $j$, i.e., in the same node of the discarded coefficient. As there are $O(N)$ discarded nodes, the cost of updates in the same node is: $O((2^D - 2)N)$ in total. Thus, the total update operations of the algorithm are: $O(N log N + N)$. Moreover, each update in a coefficient costs its re-positioning in $H$ which is an $O(log N)$ operation. The complexity of the algorithm is thus: $O(\frac{(2^D-1)^2}{D} N log^2 N + (2^D - 2) N log N)$, that asymptotically remains to be $O(N log^2 N)$ as in the one-dimensional case. Please also note, that in contrast to MDMSpace, the term $2^D$ does not have exponential impact on the running-time complexity.

**Algorithm 8:** MGreedyAbs

**Input:** $W_A$ vector of N Haar wavelet coefficients
H := create_heap($W_A$)
**while** H not empty **do**
  **discard** $c_{jk}$ := H.top // coefficient with smallest $MA_{jk}$
  **for** $s = 0; s \leq 2^D - 1; s++$ **do**
    $max_j^s = max_j^s - sign(s)\, c_{jk}$; $minx_j^s = minx_j^s - sign(s)\, c_{jk}$
  **for** $i = 0; i \leq 2^D - 2; i++$ **do**
    **if** $c_{ji}$ not discarded **then**
      recalculate $MA_{ji}$;update $c_{ji}$'s position in H
  **for** each subtree $s \in [0, 2^D - 1]$ of node $j$ **do**
    **for** each coefficient $c_{mn} \in s$ **do**
      **if** $c_{mn}$ not discarded **then**
        Update all error measures in $c_{mn}$ by $c_{jk}$
        recalculate $MA_{mn}$;update $c_{mn}$'s position in H
  $max_{err} := \max_{0 \leq s \leq 2^D - 1} \left( max_{jk}^s, max_{jk}^s \right)$;
  $min_{err} := \min_{0 \leq s \leq 2^D - 1} \left( min_{jk}^s, min_{jk}^s \right)$; $node_i = node_j$.parent
  **while** $node_i \neq NULL$ **do**
    $max_i^l := max_{err}$; $\min_i^l := \min_{err}$
    **if** any of $\{max_i^s, min_i^s\}, s \in [0, 2^D - 1]$ changed **then**
      **if** $c_i$ not discarded **then**
        recalculate $MA_i$;update $c_i$'s position in H
      $max_{err} := \max_{0 \leq s \leq 2^D - 1} \left( max_{jk}^s, max_{jk}^s \right)$
      $min_{err} := \min_{0 \leq s \leq 2^D - 1} \left( min_{jk}^s, min_{jk}^s \right)$
      $node_i = node_i$.parent
      **else break**

## 4.4 Discussion

From an algorithmic perspective, the main difference in the construction of one- and D-dimensional wavelet synopses is the structure of the error-tree. All the modifications on the proposed algorithms aim at handling error-trees in which each node can have an arbitrary number of children.

We observe that the complexity of MDMSpace becomes prohibitive even for low dimensionalities. A dimension of $D = 4$ can lead to billions of iterations for the algorithm. On the other hand, the complexity analysis of the greedy algorithms shows that an error-tree of a D-dimensional dataset incurs a computational overhead in the order of $2^{2D}$. The conducted experiments in Section 4.5 show that the synopsis construction for a 4-dimensional dataset is only 1.5 times

slower than the synopsis construction for a one-dimensional same-sized dataset when the greedy algorithms are employed.

As the experiments indicate, it is more difficult to yield accurate wavelet synopses for datasets of high dimensionality. Intuitively, the higher the number of dimensions, the higher is the number of neighbors for a data-value in the input array. Depending on the distribution, this can lead to an increased number of discontinuities that should be captured by the synopsis.

Table 4.1 summarizes the discussed algorithms. IndirectHaar and DIndirectHaar can handle the multidimensional case only if they use MDMSpace instead of MinHaarSpace. Similarly, DGreedyAbs and BUDGreedyAbs should use MGreedyAbs for handling multiple dimensions.

Table 4.1: Summary of presented algorithms

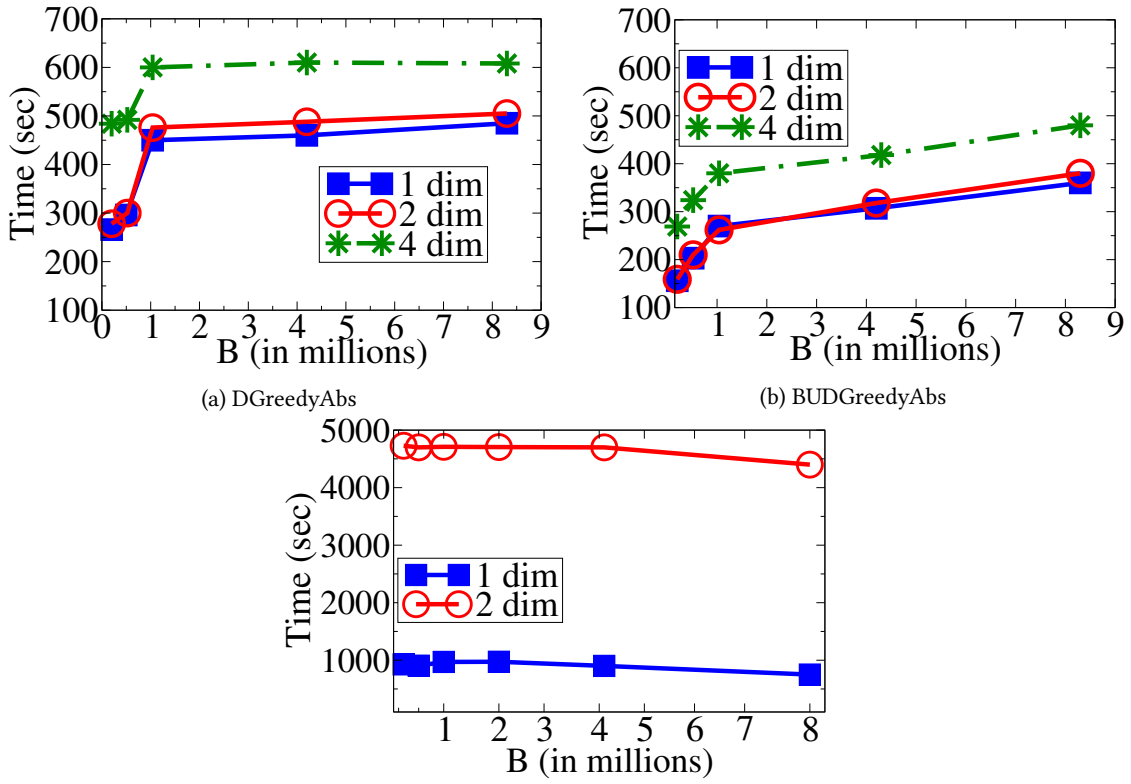
| Algorithm | Distributed | Multidimensional |
|---|---|---|
| MinHaarSpace | no | no |
| MDMSpace **(this dissertation)** | no | **yes** |
| IndirectHaar | no | **yes** |
| DIndirectHaar **(this dissertation)** | **yes** | **yes** |
| GreedyAbs | no | no |
| MGreedyAbs **(this dissertation)** | no | **yes** |
| DGreedyAbs **(this dissertation)** | **yes** | **yes** |
| BUDGreedyAbs **(this dissertation)** | **yes** | **yes** |

## 4.5 Experimental Evaluation

The proposed algorithms are evaluated in terms of (i) synopsis construction time and (ii) achieved maximum absolute error. All algorithms are implemented in Java 1.8.

**Datasets**. The experiments are conducted using both synthetic and real datasets. As synthetic data, uniform and zipfian distributions are used, with data values that lie between [0, 1000]. As multidimensional real-life datasets, we utilize NOAA [noa] and NYCT2D [nyc]. NYCT2D is a 2-dimensional dataset of $1.5$ billion records that contains trip distances and total fares for the taxi rides. For NOAA, the following four dimensions are considered: *Wind Direction*, *Wind speed*, *Temperature* and *Dew point*. All datasets are partitioned in order to test scalability over different sizes. The smallest partition comprises the first $1M$ records, while each subsequent partition is $2^D$ times the previous one, where $D$ is the dataset's dimensionality. The largest dataset consists of $268M$ datapoints.

**Platform setup**. As a deployment platform, a Hadoop 2.6.5 cluster of 9 machines is used. Each machine features eight Intel Xeon CPU E5405 @ 2.00GHz cores and 8 GB of main memory. One machine is used as the master node and the remaining ones as slaves. Each slave is allowed




(a) DGreedyAbs

(b) BUDGreedyAbs

(c) *DIndirectHaar*

Figure 4.2: Scalability with the space budget $B$.

to run simultaneously up to 5 map tasks and 1 reduce task. Each of these tasks is assigned 1 physical core and 1 GB of main memory. For all the remaining properties, we keep the default Hadoop configuration.

For experimenting with the centralized algorithms one machine with the same specifications as the ones listed above is employed. Thus, centralized algorithms may have up to 8 GB of available main memory for their execution.

### 4.5.1 Scalability

In this Section, we use synthetic data to assess the scalability with respect to the available budget for the synopsis $B$ and the number of datapoints $N$. The dataset consists of values uniformly distributed in the range: $[0, 1K]$.

**Varying space budget**. We run DGreedyAbs, BUDGreedyAbs and DIndirectHaar for $N = 17M$ datapoints and vary $B$ from $N/64$ to $N/2$. Figures 4.2-(a), 4.2-(b) and 4.2-(c) show the results for DGreedyAbs, BUDGreedyAbs and DIndirectHaar respectively. As expected, for all algorithms, the higher the dataset dimension is, the higher is the running-time of the algorithm. For the greedy algorithms and for budget sizes smaller than the partition size of the distributed

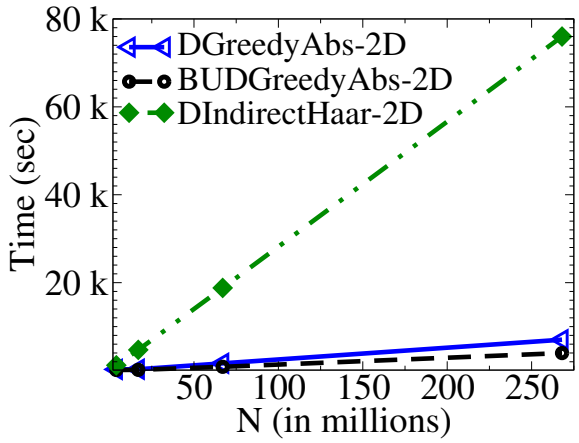



Figure 4.3: Scalability for 2-dimensional datasets

job (1M datapoints), better running-time is observed. This is due to an optimization where each worker emits only the $B$ most important coefficients to the reduce stage. When the budget is over 1M datapoints, the performance of DGreedyAbs is not affected by $B$, while BUDGreedyAbs is linearly affected. For Figure 4.2-(c), we note that the complexity of DIndirectHaar for a 4-dimensional dataset is too high and the algorithm is not able to run.

**Varying datasize**. Figure 4.3 presents scalability results when two-dimensional datasets of different sizes are used. Once again, all examined algorithms scale linearly with the dataset size. The important observation here is that the running-time gain of the greedy algorithms compared to *DIndirectHaar* increases along with the dimensionality. For the 2-dimensional datasets, the greedy algorithms present almost the same performance with the 1-dimensional case while *DIndirectHaar* becomes considerably slower.

The main results of this Section are that: (i) dimensionality positively affects running-time and (ii) the higher the dimensionality, the higher is the benefit of using a greedy algorithm.

### 4.5.2 Data Dimensionality and Maximum Absolute Error

In this Section, we investigate how dimensionality affects maximum absolute error and what trade-offs DIndirectHaar offers for the high computational overhead. For this experiment, synthetic datasets of 1, 2 and 4 dimensions are considered and budget is set to $B = \frac{N}{16}$. All datasets follow a zipfian-1.5 distribution, with a size of $N = 17M$ datapoints. The choice of distribution is inline with previous research [GK05], as it has been shown that wavelets can better capture skewed distributions.

In Figure 4.4, we notice that the achieved accuracy is negatively affected by an increase in dimensionality. The higher the dimensionality, the higher is the observed error. This is probably

an effect of the enhanced locality in high-dimensional spaces. Furthermore, DIndirectHaar compensates for its high computational complexity with an error 30% smaller than the one achieved by the greedy algorithms for both multidimensional datasets.

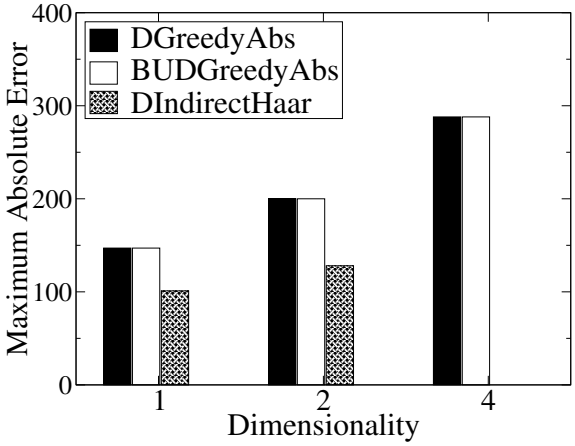


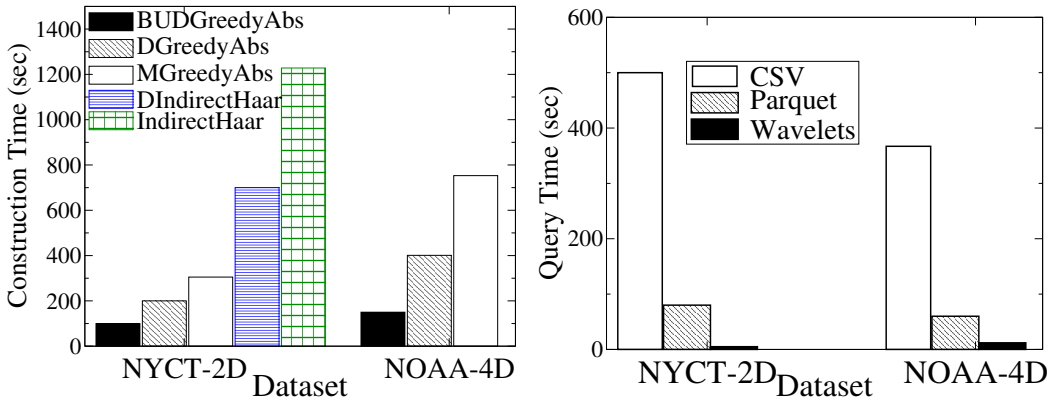

Figure 4.4: Maximum Absolute Error for Zipfian data and $B = N/16$.

### 4.5.3 Comparison for Real Datasets

For the multidimensional experiments, the NYCT2D and NOAA datasets are used. Furthermore, in order to demonstrate the merits of wavelet thresholding in exploratory analysis tasks, a query-time evaluation for the constructed synopses is also presented. For answering queries on wavelet synopses, I have implemented the work of [CGRS01]. As proposed there, instead of applying the wavelet transform directly on the data, we first construct a datacube of joint frequencies. After the synopsis is constructed, it can be loaded in main memory and provide in-memory query answering.

Figure 4.5a presents the results of the construction time comparison when $B = N/16$. For both datasets, BUDGreedyAbs is the most time-efficient algorithm. DP algorithms are able to run only for the NYCT2D dataset, where IndirectHaar is $12\times$ and DIndirectHaar $7\times$ slower than BUDGreedyAbs. Despite the dimensionality of these datasets, we observe that all algorithms achieve lower running-times than the ones achieved in Figure 3.11b. This may seem counter-intuitive, but the explanation lies behind the distribution of the wavelet transform. The transforms of NOAA and NYCT2D are sparse enough and the data that the thresholding algorithms actually process are fewer than the original dataset.

Regarding quality guarantees, all greedy algorithms produced a maximum absolute error of 1.8 and 0.9 for the NYCT2D and NOAA datasets respectively. As the errors are already small enough, the DP algorithms could not yield an interesting trade-off for the high-running time they present.




(a) Synopsis Construction Time (b) Query time

Figure 4.5: Synopsis Construction and Query Time for real-life datasets.

Figure 4.5b shows the results of the query time experiment. We consider queries of the form ***select*** $\{sum, count, avg\}$ ***from T where*** $p_1 \wedge ... \wedge p_k$, where $p_i$ is an inequality predicate. For each dataset, we run a workload of 10 random queries of that form and present the average query time. We compare query time on wavelet synopses against SparkSQL [spa] queries on raw csv and Parquet [par] files. The csv text files do not fit in the aggregate memory of the cluster we have configured and thus they produce the worst query latencies. As Parquet enables lossless compression mechanisms, the corresponding Parquet files fit in our cluster's memory and improve a lot on the observed query time. However, our wavelet synopses with $B = N/16$ can fit in a single machine's main memory and thus present the best query times.

The main conclusions from the comparisons in this Section are that: (i) The proposed distributed approaches scale to datasizes that the traditional centralized algorithms are unable to process. (ii) The most time-efficient algorithms are BUDGreedyAbs and DGreedyAbs and each of these algorithms can be the most appropriate choice in a different use-case; when $B$ is not too large, the BUDGreedyAbs algorithm is suggested. (iii) DIndirectHaar produces results of better quality but it presents the worse running-time and ends up to be impracticable in higher dimensions.

CHAPTER 5

# Online Synopses for Sliding Window Aggregates

## 5.1 Introduction

In this Chapter, efficient algorithms are proposed for the computation and online maintenance of wavelet synopses. The construction process should be constrained to a limited memory budget, that is usually much smaller than the window size ($B << W$). This is a realistic requirement in many real-life applications. For example, embedded devices such as Arduinos[1], that are often met in IoT scenarios, possess memory in the order of KB [arda]. Thus, a space budget $B$ should be defined and cap the number of retained wavelet coefficients. In the analysis of this Section, synopses logarithmic in the window size are considered, i.e., $B = O\left(logW\right)$.

The goal is to evaluate the ability of wavelets to accurately compute point queries and basic range statistics (SUM, COUNT, AVG) in a data stream that follows the time-based sliding-window model and where data elements are expected to arrive in the stream-order. Such a stream is formally defined in Definition 1. Henceforth, the term *stream* is used to describe such a data sequence.

**Definition 1** (Ordered Time-based Stream)**.** *An ordered, time-based data stream is an infinite sequence of tuples in the form:* $S = \{(t_1, v_1), (t_2, v_2), ...\}, t_1 \leq t_2 \leq ...$*, where* $t_i$ *denotes the arrival time of tuple* $i$ *and* $v_i$ *its value.*

[1] https://www.arduino.cc/

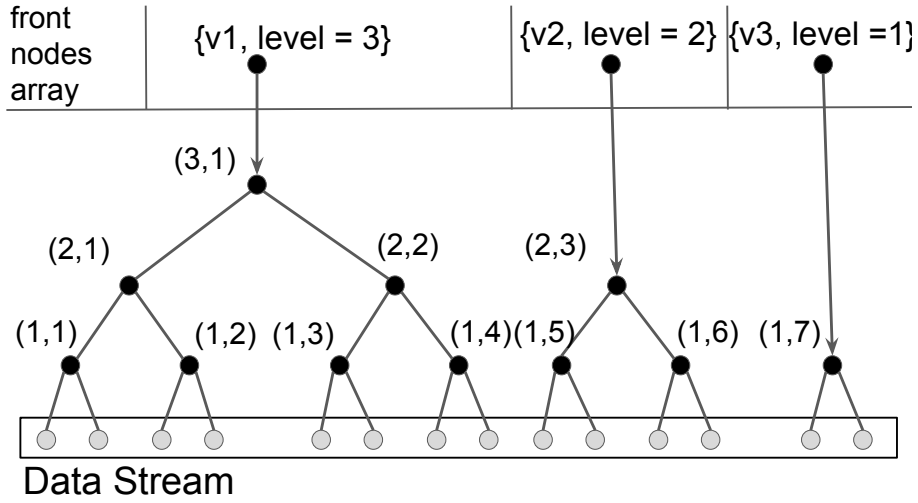



Figure 5.1: Error-tree for streaming data.

Both the sliding-window point and range queries are defined in Definition 2. A point query can ask for the stream value at any time moment lying within the active window. Similarly, a range query has always the current time as the end of its interval, while the start of it can be any time moment within the window.

**Definition 2.** *Let $S$ be a stream, $t$ the current time and $W$ the window size.*

- *A **sliding-window point query** $P(t_q)$ on $S$ returns an estimation for the value $v_q$ that arrived at time $t_q$, $t_q \in [t - W, t]$.*
- *A **sliding-window range query** $AGG(t_q)$ on $S$ returns an estimation for an aggregate $AGG \in \{SUM, COUNT, AVG\}$ computed over the time range: $[t_q, t]$, where $t_q \in [t - W, t]$.*

While this Chapter mainly discusses range queries of the described form, in order to demonstrate the general applicability of the proposed approach, in Section 5.7, queries of the form $[s, e]$, where $t - W \leq s \leq e \leq t$ are also investigated.

## 5.2 Dynamic Synopsis Maintenance

### 5.2.1 Streaming Error-Tree

Similarly to previous works, we operate on the streaming version of an error-tree [LTC10, KM05]. Each pair of newly arrived items is subjected to the wavelet transform and inserted into the error-tree. During this construction process, at some time $t$, the number of stream data that have arrived may be unequal to a power of two. Hence, the error-tree has not formed a full binary tree as in the static case and unconnected sub-trees of different heights may exist. That means

that there can be at most one such sub-tree rooted at each error-tree level (thus, $\lfloor logW \rfloor$ sub-trees). Figure 5.1 depicts an example, where there are three unconnected sub-trees of heights: one, two and three respectively. In order to avoid information loss and be able to continue the decomposition process, we need to keep track of all sub-trees in the active window. For this purpose, the *front nodes array* structure is used. For each sub-tree, that we want to track, we create a *fnode* (i.e., a new element of the front nodes array) annotated with: (i) the timestamp of the first supported item, (ii) the level of the sub-tree and (iii) the average value of its data. We then set the created fnode to point to the sub-tree and append it in the front nodes array, as shown in Figure 5.1.

**Indexing Coefficients.** In the streaming error-tree, a wavelet coefficient $c_i$ is indexed by a tuple $(l_i, o_i)$, where $l_i$ is the level of the coefficient in the error-tree and $o_i$ its order in the specific level. Figure 5.1 illustrates the indexing scheme for our example. Given two coefficients $c_i$, $c_j$, where $c_i$ is an ancestor of $c_j$, $c_j$ belongs to the left sub-tree of $c_i$ if: $2 \cdot o_j - 1 < (2 \cdot o_i - 1) \cdot 2^{l_i - l_j}$.

This dissertation exploits the sliding-window and proposes an efficient representation that minimizes the space overhead for a coefficient. The key observation is that we do not have to index an infinite stream but, at any given time, the synopsis approximates a single window of size $W$. As the level of a coefficient can be at most $logW$, for $l_i$ we need at most $loglogW$ bits. For reducing the size of the $o_i$ values, which are infinite in an unbounded stream, we use a wrap around counter $o'_i = \left[(o_i - 1)\, mod \frac{W}{2^{l_i}} + 1\right]$ that uses $log\frac{W}{2^{l_i}}$ bits for a coefficient in level $l_i$. With this scheme and for a window of size 1 billion, a coefficient needs at most 35 bits for storing both $l_i$ and $o_i$.

### 5.2.2 Algorithm Outline

Algorithm 9 shows the outline of the streaming algorithm for the construction of a wavelet synopsis. Each pair of newly arrived data is transformed into a wavelet coefficient and inserted into the error-tree. The addition of a new coefficient may trigger the creation of more coefficients in higher levels. In Figure 5.1, when two more items arrive, a new wavelet coefficient will be inserted in the first level of the error-tree. As there is already one node in the first level, the two coefficients will be averaged and differenced and create a new coefficient in level two. The process will be recursively repeated and new wavelet coefficients are expected to be also added in levels three and four. In general, every new item in the stream can fire up to $\lceil logW \rceil$ insert-updates in the wavelet structure.

In line 4, we first check whether there are coefficients that lie outside the active window and thus have expired. If such coefficients exist, we can safely discard them releasing this way space without compromising accuracy (they support a range we are no longer interested in).

**Algorithm 9:** Streaming Algorithm for Constructing a Sliding-Window Wavelet Synopsis

```
   input: Stream S, Budget B, Window size W
1  currTime = 0; wSynopsis = new WaveletSynopsis();
2  for data items in S do
3  |  currTime = currTime + 2; d1, d2 = read(S);
4  |  wSynopsis.deleteExpired(currentTime, W);
5  |  wSynopsis.insert(currTime,W,d1, d2);
6  |  while wSynopsis.size > B do
7  |  |  wSynopsis.discardNext();
```

Next, we insert the new elements. Depending on the data distribution, the wavelet transform may produce some zero coefficients. These coefficients are never inserted in the structure we maintain. If after the insert-step, the size of the synopsis still exceeds $B$, we discard coefficients according to a greedy criterion (will be later discussed) until the size of the synopsis respects the available budget.

We now delve into the internals of each of the *insert*, *deleteExpired* and *discardNext* functions.

**Insert.** The algorithm for the insertion of new coefficients in the synopsis is presented in Algorithm 10. For each pair of arrived items $d_1$, $d_2$, we perform averaging and differencing (line 7) and create a new wavelet coefficient $c_i$. If $c_i$ is non-zero, we add it to a min-heap (line 16) in order to specify its order of deletion. In line 18, we check if $c_i$ is the only node at level $l$. If this is the case, we create a new fnode (line 19) that points to $c_i$, else we continue the process at the next level of the error-tree, as explained in the example of Figure 5.1.

According to the proposed algorithm, all fnodes that support a part of the active window are retained in the synopsis. This is the reason why fnodes are not inserted into the min-heap. As we will explain in Section 5.3, this design choice improves the approximation quality of range queries.

Moreover, in line 5 of the algorithm, we notice that a cap is enforced on the maximum level of a sub-tree; the wavelet decomposition is not allowed to continue further than *maxLevel* levels. This decision permits the existence of more than one fnodes with *maxLevel* levels. We store these fnodes in a separate structure called *topLevelFnodes* (line 22). We claim that a limit on the maximum level of the error-tree offers two advantages: i) lower bounded update times, and ii) allows for the more accurate computation of range queries.

The first claim can be trivially verified. From the *while* condition of Algorithm 10, we can see that an insert operation can trigger up to $logW$ updates. For a *maxLevel* $< logW$, we directly restrict the number of updates at every time unit. The impact of *maxLevel* in the accuracy of range queries will be discussed in Section 5.3, where the query answering mechanism is described.

**Algorithm 10:** Insert

**input:** Number of arrived items $N$, window size $W$, item $d_1$,item $d_2$

```
1  f = fnode with lowest level; tmp = null; l = 0
2  maxLevel = log (W / logW)
3  while N > 0 and N mod 2 = 0 do
4      N = N / 2; l = l + 1
5      if l > maxLevel then break
6      if tmp = null then
7          avg = (d1 + d2) / 2; v = (d1 - d2) / 2
8          minCf = maxCf = v
9      else
10         avg = (avg + tmp) / 2; v = tmp - avg
11         minCf = min (prevFnode.minCf, tmpMin, v)
12         maxCf = max (prevFnode.maxCf, tmpMax, v)
13     c_i = new WaveletCoef(l_i = l, o_i = N, value = v)
14     c_i.maxCoefInSubtree = maxCf
15     c_i.minCoefInSubtree = minCf
16     if c_i ≠ 0 then put c_i in min-heap
17     delete fnode below f
18     if no fnode in level l then
19         f = new Fnode(level = l, value = avg)
20         f.minCf = minCf; f.maxCf = maxCf
21         if l < maxLevel then frontNodesArray.add(f)
22         else topLevelFnodes.add(f)
23     else
24         tmp = f.value
25         tmpMin = f.minCf; tmpMax = f.maxCf;
26     if f.pointer = null then f.pointer = c_i
27     f = fnode at next level
```

Now, we are going to investigate what is an appropriate value for *maxLevel*. A small value offers the advantages we just mentioned. Nevertheless, as all fnodes are retained in the synopsis, a cap on the maximum level increases the space we need to dedicate to the front nodes array. Thus, we need to set a value such that we enjoy the benefits of a short tree without significantly increasing space complexity. The value we select is $\lceil log\left(\frac{W}{logW}\right)\rceil$. The following Lemma shows that with this choice we only require poly-logarithmic space in the window size for storing the front nodes array.

**Lemma 8.** *Consider a wavelet error-tree $T$ built over $W$ data points. Setting the constraint that each sub-tree of $T$ cannot have more than $\lceil log\left(\frac{W}{logW}\right)\rceil$ levels, results in storing at most $O\left(logW\right)$ fnodes.*

*Proof.* Let $k$ denote the maximum permitted size for a sub-tree. Thus, within a window of size $W$ there can be up to $\lceil\frac{W}{k}\rceil$ such sub-trees, and thus $\lceil\frac{W}{k}\rceil$ fnodes. As the given budget $B$ is usually poly-logarithmic in $W$, we want to store at most $O\left(logW\right)$ fnodes. So, it should hold: $\frac{W}{k} \leq c \cdot logW, c \geq 1 \Rightarrow k \geq \frac{W}{c \cdot logW}$. Thus, the minimum sub-tree size we can tolerate without violating the constraint of $O\left(logW\right)$ fnodes is the first power of 2 that is larger than $\frac{W}{c \cdot logW}$ and has $M = \lceil log\left(\frac{W}{c \cdot logW}\right)\rceil$ levels. However, the construction process of a wavelet tree is such that we may have more than $\lceil\frac{W}{k}\rceil$ sub-trees in the window. As it is known that $\Sigma_{i=0}^{n-1} 2^i = 2^n - 1$, we can substitute a sub-tree of size $k$ with up to $M-1$ sub-trees of levels $l = 1, .., M-1$. This way, there are at most $\lceil\frac{W}{k}\rceil - 1 + M - 1 = \lceil\frac{W}{k}\rceil + \lceil log\left(\frac{W}{c \cdot logW}\right)\rceil - 2$ sub-trees and thus fnodes in the window. As we want to save space, we set $c = 1$ and in the worst case we have $logW + log\left(\frac{W}{logW}\right) = O(logW)$ fnodes. □

The cost for inserting new elements in the wavelet synopsis is given by Lemma 9.

**Lemma 9** (Insertion Time)**.** *Considering a synopsis size of $B = O\left(logW\right)$, an arriving pair of data items leads to a worst case insertion time of $O\left(log\frac{W}{logW} \cdot loglogW\right)$ and $\Theta\left(log\frac{W}{logW}\right)$ in the average case.*

*Proof.* The cost of an insert-update consists of the cost of creating new coefficients and the cost of re-configuring the binary heap. The proof for the worst-time case is straightforward: As we discussed, an insert-update can lead to the creation of $L$ new wavelet coefficients, where $L$ is the size of the tree. Since our algorithm permits only sub-trees of height up to $\lceil log\left(\frac{W}{logW}\right)\rceil$, it follows that this is also the maximum number of operations that an insert-update can cause. Moreover, since the synopsis should occupy only poly-logarithmic space, we assume a min-heap of size $B = O\left(logW\right)$. Thus, the worst-case insertion in the heap is $O\left(loglogW\right)$. It follows that the total needed worst-case time for updating the synopsis when two new data items arrive is $O\left(log\frac{W}{logW} \cdot loglogW\right)$.

We now compute $\Theta$ complexity. The insertion in a binary heap needs $\Theta(1)$ time on average. The question is how many wavelet coefficients are created with every new arrival in the average case. Without loss of generality, we assume a tree of size $N$, where $N$ is a power of two. Each arriving item can trigger the creation of $1 \leq i \leq logN$ coefficients. Since there are $N$ items within the window, we first compute how many of them create 1 coefficient, how many 2, etc. Let $a(j)$ denote the number of coefficients within a window that lead to the creation of paths of length $logN - j$. We observe that only the last element can create a path of length $logN$, i.e., $a(0) = 1$. The same holds for a path of length $logN - 1$. There are two paths in the window that have length at least $logN - 1$. However, the one of them has length $logN$ and thus, $a(1) = 1$. With similar reasoning, we observe that the following recursion holds: $a(0) = 1$ and $a(j) = \sum_{i=0}^{j-1} a(i)$. As the first two elements of the $a(j)$ sequence add up to 2, it is easy to derive that:

$$a(j) = \begin{cases} 1 & j = 0 \\ 2^{j-1} & j \neq 0 \end{cases}$$

Since it is known that $\sum_{i=0}^{n-1} 2^i = 2^n - 1$, we observe that:

$$\sum_{j=0}^{logN-1} \frac{a(j)}{N} = \frac{1 + \sum_{j=0}^{logN-1} 2^j}{N} = \frac{1 + 2^{logN} - 1}{N} = 1$$

and thus the term $\frac{a(j)}{N}$ can represent the probability of creating a path of length $logN - j$. Let the random variable $X$ express the number of updates a newly arriving data pair yields. The expected value of $X$ can be expressed as:

$$\mathbb{E}(X) = \sum_{j=0}^{logN-1} \frac{a(j)}{N} \cdot (logN - j) =$$

$$\frac{logN}{N} + \frac{logN}{N} \sum_{j=1}^{logN-1} 2^{j-1} - \frac{1}{N} \sum_{j=1}^{logN-1} j \cdot 2^{j-1} =$$

$$\frac{logN}{N} \left( 1 + \sum_{j=1}^{logN-1} 2^{j-1} \right) - \frac{1}{N} \sum_{j=1}^{logN-1} j \cdot 2^{j-1} \tag{5.1}$$

We use again the fact that $\sum_{i=0}^{n-1} 2^i = 2^n - 1$ in order to compute the first term. For $k = j - 1$, we have: $\sum_{j=1}^{logN-1} 2^{j-1} = \sum_{k=0}^{logN-1-1} 2^k = 2^{logN-1} - 1 = \frac{N}{2} - 1$ and the first term of Equation 5.1 is equal to $\frac{logN}{2}$. For the second term, it is easily proven that when $n$ is a finite number, it

holds:
$\sum_{j=1}^{n} j \cdot x^{j-1} = 1 - \frac{x^n}{(1-x)^2} + \frac{nx^n}{1-x}$. For $x = 2$ and $n = logN - 1$, we get that:

$$\sum_{j=1}^{logN-1} j \cdot 2^{j-1} = 1 - 2^{logN-1} - (logN - 1) \cdot 2^{logN-1} = 1 - \frac{NlogN}{2}$$

Thus, Equation 5.1 becomes:

$$\mathbb{E}(X) = \frac{logN}{2} - \frac{1}{N}\left(1 - \frac{NlogN}{2}\right) = logN - \frac{1}{N}$$

Thus, the total update time for every arrived pair in the stream is $\Theta(1) \cdot \Theta\left(logN - \frac{1}{N}\right)$. As for the sub-trees there is the constraint that the maximum size $N$ is the first power of 2 that is greater than $\frac{W}{logW}$, the complexity becomes: $\Theta\left(log\frac{W}{logW} - \frac{1}{N}\right) = \Theta\left(log\frac{W}{logW}\right)$. □

**Delete Expired.** We first check if all fnodes still support the active window. As an fnode $f$ supports $2^{f.level}$ data points beginning from $f$.start, we have to discard all fnodes with: $f$.start + $2^{f.level} < currTime - W$. If a fnode is deleted, so is the whole sub-tree underneath it.

We then scan all the remaining elements to check if there are coefficients that also need to be removed. The criterion for removing a coefficient $c_i$ is: $o_i \cdot 2^{l_i} - 1 < currTime - W$. As we require $B = O(logW)$, the cost of this scan operation is also $O(logW)$.

**Discard Next.** When budget is exceeded, we need to discard some coefficients. The heuristic for selecting coefficients to discard depends on the error metric we need to optimize. If $L_2$-norm is the targeted metric, we should always keep the $B$ largest coefficients in normalized value. If the minimization of $L_\infty$ is required, we select each time the coefficient $c_k$ with the minimum *maximum potential absolute error* $MA_k$ [KM05]. The $MA_k$ value is defined as: $max_{d_j \in leaves_k}\{|err_j - \delta_{jk} \cdot c_k|\}$, where $err_j$ is the signed error for item $j$, and shows the maximum error that the removal of $c_k$ would produce. In either case, for efficiently identifying the node that should be discarded and assist the greedy selection, the synopsis is organized as a min-heap structure. In this work, the $L_\infty$ norm is used and the min-heap is implemented as a binary heap.

Lemma 10 gives the cost of deletions either due to expiration or budget excess.

**Lemma 10** (Deletion Time)**.** *The time spent in delete operations every time the synopsis is updated is $O(logW)$ in both worst and average case.*

*Proof.* Delete operations occur due to either window sliding or a manual coefficient removal in order to respect the budget constraint. We observe that in the permanent state of the algorithm (more than $B$ data items have already arrived) the synopsis size increases by at most two elements with every new arrival. Thus, there are at most two deletions that we need to make. As

the *deleteExpired* function can delete at most one coefficient, the *discardNext* function is called at most twice. The manual removal of a coefficient results in the extraction of the minimum element of a binary heap. Considering $B = O(logW)$, this operation has a worst-case complexity $O(loglogW)$ and average time $\Theta(1)$. As for identifying an expired coefficient we need to scan the whole synopsis, a $O(logW)$ operation is needed for both the worst and average case. □

### 5.2.3 Error Guarantees

Regardless of which error-metric is optimized, the constructed synopsis should be able to provide queries with deterministic guarantees. As shown in [KM05], providing guarantees for point queries demands each node to maintain the maximum and minimum signed errors of its left and right sub-trees.

This dissertation also provides deterministic guarantees for range queries. As mentioned in Chapter 2, the value of a SUM query over a range $[t_1, t_2]$ can be exactly reconstructed, by only using the coefficients $c_j \in path_{[t_1,t_2]}$, according to Equation 2.1. Here, we observe that under the sliding-window model, the sum can be computed solely based on the coefficients $c_j \in path_{t_1}$, i.e., the ones that belong to the left path of the queried interval. As it is explained in detail in Section 5.3, in the sliding-window model, we expect some sub-trees to be fully-contained in the query-range and one last sub-tree to partially overlap with it. Let us consider that $[t_1, t_2]$ is the range of overlap with the last sub-tree. Thus, by definition, $path_{t_2}$ is the rightmost path of a full binary tree. As such, every coefficient $c_j$ in $path_{t_2} \backslash path_{t_1}$ is expected to have $x_j = 0$ and does not contribute to the sum, either it is contained in the synopsis or not. Thus, $SUM_{[t_1,t_2]} = \sum_{c_j \in path_{t_1}} c_j x_j$.

For providing error guarantees, we need to bound this sum. No matter if we have deleted a coefficient $c_j$ or not, the $x_j$ value is always known since it only depends on the coefficient's position in the error-tree and the query range. So, if we had some bounds for the deleted (and thus, unknown) coefficients $c_j$, such that $l_j \leq c_j \leq h_j$, it would hold:

- $x_j \geq 0 \Rightarrow l_j x_j \leq c_j x_j \leq h_j x_j$
- $x_j < 0 \Rightarrow h_j x_j \leq c_j x_j \leq l_j x_j$

By summing up these inequalities for all deleted coefficients $c_j$, we obtain deterministic guarantees for the $SUM_{[t_1,t_2]}$. The idea for bounding $c_j$ values is to keep track of the minimum and maximum coefficients in each sub-tree. In Algorithm 10, it is annotated with blue color all required modifications for tracking minimum/maximum coefficients in each sub-tree.

## 5.3 Query Answering

Point queries $P(t_q)$ are answered as explained in Chapter 2, i.e., $P(t_q) = \Sigma_{c_j \in path_q} \delta_{qj} \cdot c_j + f.value$, where $f$ is the corresponding fnode of the sub-tree where $t_q$ belongs. We are now going to focus on the query answering mechanism for range queries.

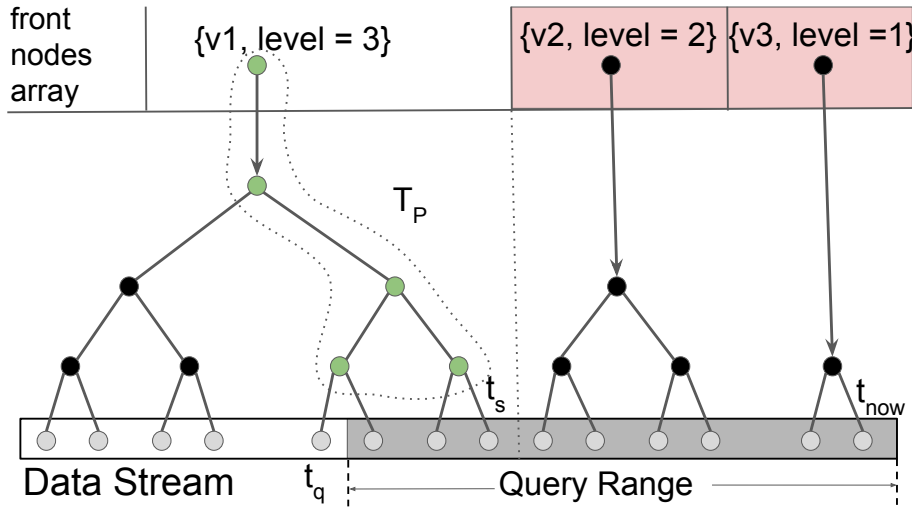



Figure 5.2: Range query answering

Figure 5.2 depicts a range query $AGG(t_q)$. The range of interest $[t_q, t_{now}]$ is highlighted with grey color. We observe that there are sub-trees which are fully-contained in the range and a last sub-tree $T_p$ that partially overlaps with it. Let us denote $t_s$ the moment in time that separates $T_p$ with the leftmost fully-contained sub-tree.

For the part of the query that corresponds to fully-contained sub-trees we can provide an exact answer. Thus, $AGG(t_q) = AGG_{approx} \oplus AGG_{exact} = AGG_{[t_q,t_s]} \oplus AGG_{t>t_s}$, where $\oplus$ is a function that combines partial aggregates. This function is a simple addition for the case of COUNT and SUM queries, while for AVG Lemma 11 holds.

**Lemma 11.** *Let $avg(\cdot)$ and $n(\cdot)$ denote the averaging and counting functions respectively. The average value of region $X = \bigcup x_i$, $i = 1, 2, .., k$ with $x_i \cap x_j = \emptyset$ can be computed as:*

$$AVG(X) = \oplus(avg(x_1), ..., avg(x_k)) = \sum \frac{n(x_i) \cdot avg(x_i)}{n(X)}$$

We first show how to compute the exact part of the aggregate and then discuss how to approximate the range that intersects with the last sub-tree $T_p$. Recall that each fnode $f_i$ keeps information about the level of its sub-tree $T_i$ and the average value of the corresponding data elements. Thus, an aggregate of $T_i$ can be computed solely based on $f_i$. Considering that a data item arrives at each time unit, a COUNT query can be computed as $2^{f_i.level}$, the answer to

an AVG query is $f_i.value$ and the SUM can be derived by $f_i.value \cdot 2^{f_i.level}$. So, $AGG_{t>t_s} = \oplus\left(AGG_{T_i}, ..., AGG_{T_j}\right)$, where $\{T_i, ..., T_j\}$ are all the sub-trees that are fully-contained in the range query $AGG\,(t_q)$.

For approximating $AGG_{[t_q,t_s]}$ we use the wavelet coefficients that lie in $path_{t_q}$. We remind that for coefficients $c_j$ in $path_{t_s} \backslash path_{t_q}$ we expect $x_j = 0$. As there is exactly one item that arrives at each time unit, we know that there are $t_s - t_q + 1$ items in the range. A SUM query can be approximated as: $SUM_{[t_q,t_s]} = \sum_{c_j \in path_{t_q}} c_j x_j + f_p.value \cdot (t_s - t_q + 1)$ and an AVG query can then be easily answered as: $\frac{SUM_{[t_q,t_s]}}{(t_s - t_q + 1)}$. Guarantees for the approximate $AGG_{[t_q,t_s]}$ are provided as follows: we traverse $path_{t_q}$ in a bottom-up fashion. For each position $j$ of the error-tree, we check if coefficient $c_j$ exists in the synopsis. If it does, we compute its contribution $c_j x_j$. If it does not, we buffer the $x_j$ value that corresponds to the missing coefficient until we find the next coefficient that exists in the synopsis. Then, we use the minimum and maximum coefficients stored in this node, in order to bound the contribution of the missing coefficients.

Thus far, we have assumed that an item arrives at each time unit. However, in reality, streams may be bursty and arrival rates do not follow a regular pattern. In order to handle the general case and be able to answer all COUNT, SUM and AVG queries, we maintain two distinct wavelet structures. The first one keeps track of a bit-stream $\{(t, b)\,, b \in \{0, 1\}\}$ that indicates whether a tuple has appeared at time $t$. The second one approximates the value distribution of the actual input stream. Let $BW$ denote the wavelet synopsis of the bit-stream and $VW$ the synopsis of the value-stream. The procedure for updating $BW, VW$ is presented in Algorithm 11. Every time $t$ a data item $(t, v)$ appears, we insert it in $VW$ exactly as explained in Section 5.2. Moreover, we insert the tuple $(t, 1)$ in $BW$ and note the time when the update takes place (line 11). While the stream is inactive and no data arrives, we keep the system idle. The next time a tuple arrives after an inactivity period, we insert $t - lastTimeActive - 1$ zero values to both $BW$ and $VW$ (line 6). This mechanism ensures that a direct mapping between the time and wavelet domains always exists. Let us also note that keeping two structures does not constitute a deficiency of the proposed approach. Exponential histograms and waves do the same in order to support both COUNT and SUM queries.

Answering COUNT queries on the stream is translated into SUM queries on the $BW$ structure. For instance, if we need to know the number of measurements that a sensor produced between times $t_1$ and $t_2$, we have to add the 1-bits that exist in the corresponding time range. SUM queries on the input stream are answered by the $VW$ structure. Since in the absence of arrived data we insert zero-values to $VW$, we do not affect the result of additive operations. For AVG and point queries, we have to "touch" both structures. For an AVG query, we compute the sum from $VW$, the count from $BW$ and divide the results.

**Algorithm 11:** BW-VW updates

```
Initialize BW, VW;
lastTimeActive = 0;
for every time unit t do
    (t, v) = listenToStream();
    if (t, v) ≠ null then
        for t* in [lastTimeActive + 1, t) do
            BW.insert((t*, 0));
            VW.insert((t*, 0));
        BW.insert((t, 1));
        VW.insert((t, v));
        lastTimeActive = t;
```

### 5.3.1 Discussion

Having described the query answering mechanism of the proposed algorithm, we now discuss the impact of limiting the maximum level of a sub-tree. We saw that an error is introduced only due to the range $[t_q, t_s]$. Intuitively, the higher is the $T_P$ wavelet sub-tree, the larger this range can be. By keeping sub-trees short, we increase the possibility to have more sub-trees fully-contained in the query-range and thus, increase the exact part of the answer $AGG_{t>t_s}$. The following Lemma shows how the maximum level we allow for sub-trees affects the relation between the $[t_q, t_s]$ and $[t_{s+1}, t_{now}]$ ranges.

**Lemma 12.** *Let $Q$ a range query, $E$ =$[t_{s+1}, t_{now}] \subseteq Q$ the sub-range of $Q$ for which our structure provides an exact result and $A$ the sub-range of $Q$ that we need to approximate. It holds that: $\frac{|A|}{|E|} \geq \frac{1}{2logW}$.*

*Proof.* We distinguish two cases depending on whether $A$ overlaps with a sub-tree of height $\lceil log\frac{W}{logW} \rceil$ or not. Let us initially assume that $A$ overlaps with a sub-tree of size $2^k$, with $k < \lceil log\frac{W}{logW} \rceil$. The maximum length of the range we need to approximate is $|A| = 2^{k-1}$. By the wavelet construction, it is guaranteed that there can be up to $k-1$ trees in $E$ of sizes $2, 4, .., 2^{k-1}$ and thus $|E| \leq \sum_{i=2}^{k-1} 2^i = 2^k$. It follows: $\frac{|A|}{|E|} \geq \frac{1}{2}$. We now consider the case where $A$ overlaps with a sub-tree of size $M$, where $M$ is the first power of 2 which is greater than $\frac{W}{logW}$. In that case, it holds that $|E| \leq W$ and $|A| = \frac{M}{2}$, and so we have $\frac{|A|}{|E|} \geq \frac{M}{2W} \geq \frac{\frac{W}{logW}}{2W} = \frac{1}{2logW}$. □

Lemma 12 implies that for range queries of length near to $W$, the proposed method has to approximate only the $\frac{1}{2logW}$ of the query. The larger the window size, the larger the portion of the query we can exactly compute. For windows larger than 1 million items, we have to approximate less than 3% of the queried range. This is a direct consequence of limiting the maximum level a sub-tree can have. According to the proof, the corresponding ratio in classic wavelets is $\frac{1}{2}$ in the best case.

As factor $\frac{1}{2logW}$ bounds the range we have to approximate but does not contain information on data values distribution, it favors mostly COUNT queries but no theoretical guarantees can be given for SUM and AVG. However, the experiments of Section 5.7 show that the proposed approach is very robust and that for queries of length $W$ high quality results are achieved for all examined datasets, both real and synthetic.

Other methods, such as exponential histograms (EH), provide theoretical guarantees by tracking query results over time. Instead of approximating the data distribution of the stream, as this work does, they approximate the distribution of a query over time. For example, in the case of a SUM query, they maintain a structure that tracks the SUM at different time intervals. The benefit of wavelet-based techniques compared to such approaches is flexibility to handle more generic query types and underlying data distributions. EH-like techniques are restricted to only handle streams of positive integers and answer a single query. While due to Lemma 12, our method performs better when applied to positive numbers, in Section 5.7, it is shown that it can also be efficiently applied to streams of arbitrary numerical data. Moreover, the same structure can be used to also answer point queries and more general range queries, where the end of the query range is not equal to the current time.

## 5.4 Distributed Wavelets For Streams

This thesis also addresses the problem of tracking basic sliding-window aggregates over the union of local streams in a large-scale distributed system. By union, we mean a linear combination (e.g., average) of the remote streams. In the described setting, the remote sites are not allowed to exchange information with each other but communicate through the network with a centralized coordinator node. Let us consider a linear function $F$ applied on a set of $N$ distributed streams $S_i, i = 1, .., N$. Our goal is to answer COUNT, SUM and AVG queries on $F$, i.e., $AGG\left(F\left(S_1, .., S_N\right)\right)$, while minimizing communication; collecting all streaming data is too costly to afford in many real use-cases. Therefore, similarly to [GKMS07], each remote site computes a wavelet synopsis ($WS$) on its local stream ($S$) and it is only the synopses that are sent to the coordinator. This way, the communication cost is reduced.

The coordinator computes the requested aggregate directly in the wavelet domain. As Haar wavelets are linear functions of the original streams and $F$ is also a linear function, if we apply $F$ on the individual synopses $WS_i$, we are going to get a wavelet synopsis of $F\left(S_1, .., S_N\right)$. Thus, $WS\left(F\left(S_1, .., S_N\right)\right) = F\left(WS_1, .., WS_N\right)$ and we can approximate the query $AGG\left(F\left(S_1, .., S_N\right)\right)$ as $AGG\left(F\left(WS_1, .., WS_N\right)\right)$.

Figure 5.3 illustrates an example. Sites $1, 2$ monitor their local streams $s_{1i}, s_{2i}$ and construct the corresponding wavelet synopses. At the coordinator node, we want to track the stream $F\left(s_{1i}, s_{2i}\right)$. Instead of collecting the $s_{1i}, s_{2i}$ values, applying $F$ on them, computing the wavelet

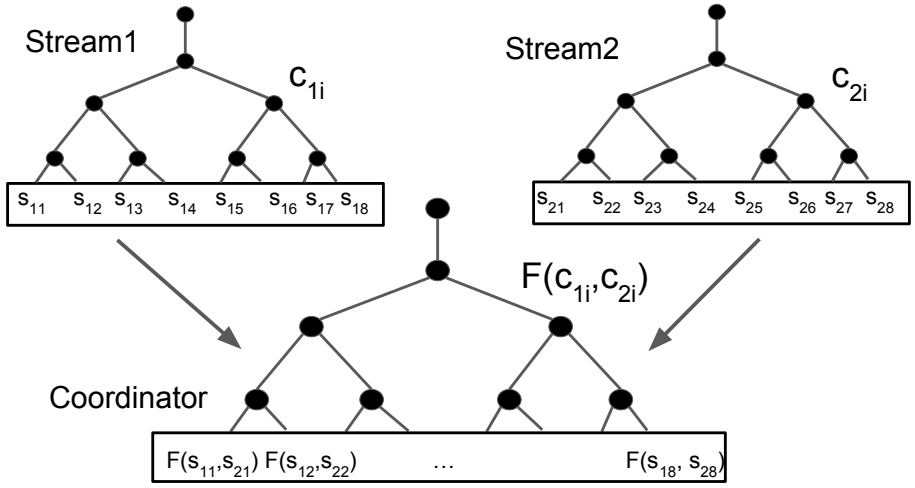



Figure 5.3: Composition of individual wavelet synopses.

transform and constructing the synopsis, we observe that for each coefficient with index $i$, it holds that $cm_i = F(c_{1i}, c_{2i})$, where $cm_i$ is the corresponding coefficient in the error-tree of the coordinator. Thus, it suffices to aggregate the coefficients by index and compute the $F$ function. The following Lemma shows that the maximum error guarantees in the wavelet synopsis of the coordinator also follow the $F$ function. Therefore, we are able to provide deterministic guarantees to queries on the union of the streams.

**Lemma 13.** *Let $S_1, S_2, .., S_N$ be $N$ streams and $\epsilon_{1k}, \epsilon_{2k}, .., \epsilon_{Nk}$ the corresponding maximum absolute errors for the reconstruction of the data value at $t = k$. The corresponding error in the stream $F(S_1, .., S_N)$, where $F$ is a linear function, is $F(\epsilon_{1k}, \epsilon_{2k}, .., \epsilon_{Nk})$.*

*Proof.* Since the reconstruction error of stream $S_i$ for $t = k$ is $\epsilon_{ik}$, it holds: $|\sum \delta_{kj} c_{ij} - d_{ik}| \leq \epsilon_{ik}$, where $c_{ij}$ are the wavelet coefficients of $S_i$ that have been retained in the synopsis. Let $F = a_1 x_1 + .. + a_N x_N$. By applying $F$ on the above inequalities we get: $-a_i \epsilon_{ik} \leq \sum \delta_{kj} c_{ij} a_i - a_i d_{ik} \leq a_i \epsilon_{ik}$. Summing up for all streams yields:

$$|\sum \delta_{kj} F(c_{1j}, .., c_{Nj}) - F(d_{1k}, .., d_{2k})| \leq F(\epsilon_{1k}, .., \epsilon_{Nk})$$

□

## 5.5 Out-of-Order Arrivals

While so far we have considered time-based streams where items arrive in order, this is not a real restriction of the algorithm. In favor of completeness, in this Section it is described how the scheme can be generalized to handle out-of-order arrivals. In Algorithm 11, we saw that in case

arrivals are in order but a discontinuity in time exists, i.e., the next arrived value has a timestamp $t = t_{now} + k, k > 1$, we pad the stream with zero-values. This way it is ensured that the wavelet transform is performed over a continuous time domain and the error-tree contains a path for each possible time $t$.

**Algorithm 12:** Out-of-Order Updates

```
   input: Synopsis S, update tuple (t_p, v), t_now − W ≤ t_p ≤ t_now
 1 // find the sub-tree where t_p belongs fnode = null;
 2 for f in fnodes do
 3     if f.start ≤ t_p ≤ f.start + 2^{f.level} then
 4         fnode = f;
 5         break;
 6 // compute new coefficients;
 7 level = 1;
 8 order = ⌈t_p/2⌉;
 9 while level ≤ fnode.level do
10     if (level, order) in S then
11         c = S.get((level, order));
12         c.value+=δ_ij · v/2^level;
13         update the MA-value of c;
14         update max/min coefficients in sub-tree rooted at (level, order);
15     else
16         c_j = findDirectAncestor(level, order);
17         c = ((level, order), δ_ij · v/2^level);
18         c.errorInfo = c_j.errorInfo;
19         update the MA-value of c;
20         update max/min coefficients in sub-tree rooted at (level, order);
21     S.add(c);
22     level += 1;
23     order = ⌈order/2⌉;
```

It is now described how we handle the case where a tuple $(t_p, v)$ with $t_p < t_{now}$ arrives. The only restriction we have is: $t_p \geq t_{now} - W$, i.e., the tuple should lie within the active window. We first have to find the sub-tree where this tuple belongs. This can be accomplished by a linear scan over the fnodes. It is reminded that each fnode $f$ maintains the start point $f.start$ of its coverage in time as well as its level $f.level$. Thus, the range it spans in time is $[f.start, f.start + 2^{f.level}]$. Then, for the $f.level$ levels of the sub-tree, we compute the contribution of value $v$ to the wavelet nodes in $path_{t_p}$. The contribution of $v$ to a wavelet node $c_i$ with index $(l_i, o_i)$ is $\delta_{ij}\frac{v}{2^{l_i}}$, where $\delta_{ij} = 1$ if $t_p \in$ leftleaves$_{c_i}$ and $-1$ otherwise. Each of the newly computed coefficients $((l_i, o_i), \delta_{ij}\frac{v}{2^{l_i}})$ has to be inserted into the synopsis. If a coefficient $((l_i, o_i), v_{old})$ already exists for the index $(l_i, o_i)$, then we just update its value and the quantities that help us provide error

guarantees (e.g., $MA$-value). If the node that corresponds to index $(l_i, o_i)$ has been deleted, the newly computed node is directly inserted into the synopsis. Nevertheless, in the latter case, the new coefficient misses some information (maximum/minimum errors and coefficients in sub-tree) for providing error-guarantees. For dealing with this issue, we can find its first available ancestor $c_j$ in the error-tree and inherit that information from there. However, as the ancestor $c_j$ covers a larger part of the time domain than $c_i$ does, the max/min values it maintains are derived not only from the sub-tree rooted at $c_i$ but from other sub-trees too. Hence, the error-guarantees will still hold but are expected to be loosened compared to the in-order case. Algorithm 12 describes the process in more detail.

## 5.6 Workload Aware Synopses

Section 5.3.1 provides an intuition on why the proposed scheme works well in a variety of cases but also indicates some of its weaknesses: it does not provide theoretical guarantees and it is not expected to present a good behavior when the query range is significantly smaller than $W$. In this Section, we are going to demonstrate how we can boost performance in these cases too, assuming we have knowledge of the workload.

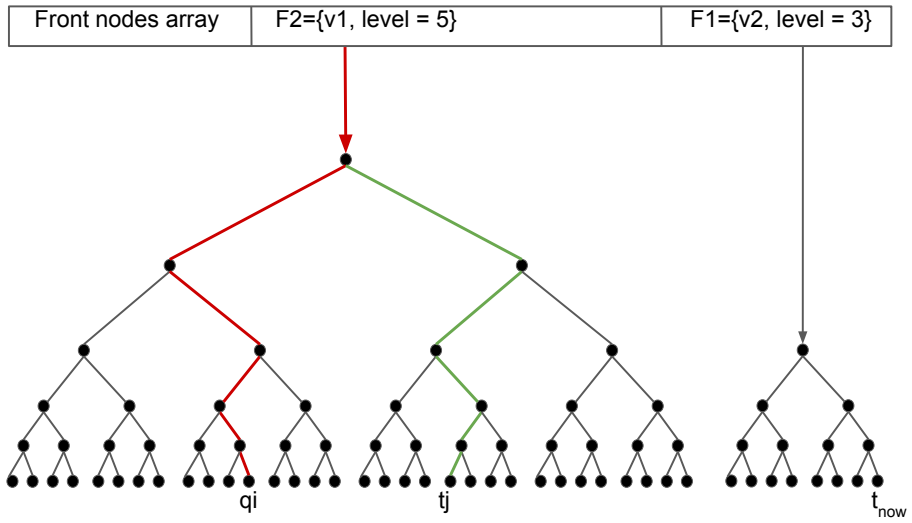



Figure 5.4: Example demonstrating the pitfalls in workload-aware sliding-window synopses: If $q_i$ is a query of interest, eventually all coefficients in paths $t_j > t_{now} - q_i$ will be requested. Hence, we have to delete coefficients that we know they will be important in the future.

We consider workload to be a set of fixed queries in the form $Q = \{q_1, q_2, ..., q_k\}$ which are known a priori and can be asked at any time. Each $q_i$ represents a range query $[t_{now} - q_i, t_{now}]$ and thus it should be $0 \leq q_i < W$. The problem of constructing an optimal wavelet synopsis with respect to a set of range queries has been extensively studied [GPS08]. Guha et al. propose

both DP and heuristic algorithms not only for prefix queries[2], which is our case, but also for the more general case of hierarchical range queries. However, they examine the static version of the problem where data is fixed and does not change over time. The real-time requirements we have, and the sliding-window model render the approach of [GPS08] inapplicable to our case.

In Section 5.2 we observed that in order to compute a range sum over $[t_j, t_{now}]$ we only need the coefficients $c_i \in path_{t_j}$. Then, the answer is derived by the fnodes of the sub-trees that are fully contained in the query and the term $\sum_{c_i \in path_{t_j}} c_i x_i$, where $t_j$ belongs to the last sub-tree, that partially overlaps the query range. If we knew the coefficients $c_i \in path_{q_j}$ for all $q_j \in Q$ then the provided answers would always be exact.

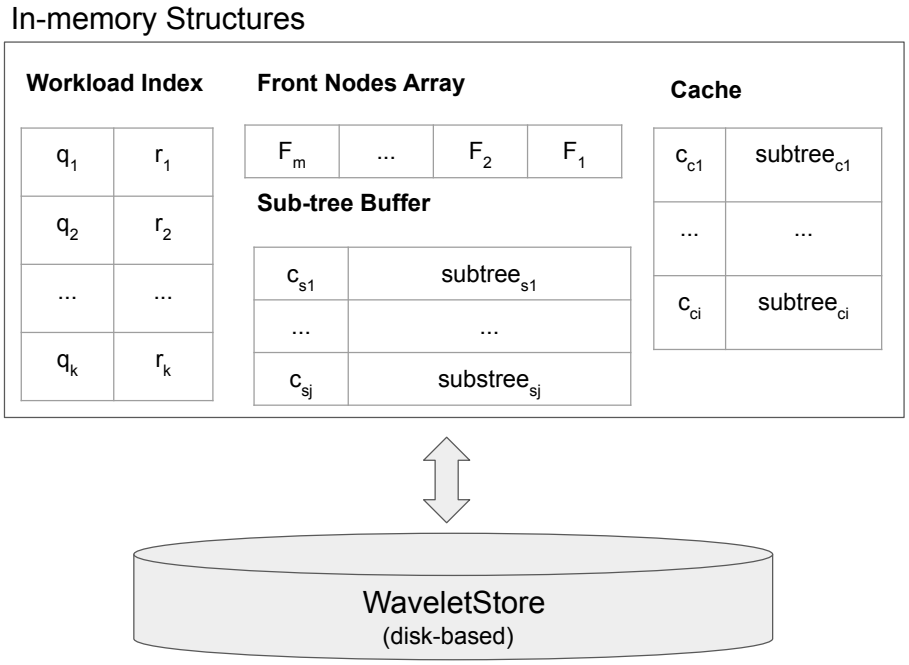



Figure 5.5: Architecture of the proposed system for workload-aware range queries in sliding-window streams.

For understanding the extra difficulties the streaming case introduces, we consider the example of Figure 5.4. Let us assume a budget of $B = 5$ coefficients and a workload $Q = \{q_i\}$. If we keep in the synopsis the whole $path_{q_i}$, then at $t_{now} = t_1$ we will be able to provide an exact answer. Nevertheless, in order to achieve this, we have discarded all the other coefficients in the active window. Hence, since we always care for $q_i$, in a later time (e.g., $t_{now} = t_1 + 9$), the coefficients of $path_{t_j}$ (annotated with green color) will be of interest but they will have already been deleted.

[2]In prefix range queries, the start (or end) of a range is always the same for all queries of the workload

In order to overcome this obstacle, this thesis introduces a system design that violates the "one-pass" over the data feature but offers very interesting trade-offs between accuracy and real-time responsiveness. Besides the limited memory, many IoT devices are also equipped with a secondary storage (e.g., SD card) [ardb] with larger capacity but which is more "expensive" to access. Based on this observation, the system of Figure 5.5 is proposed. According to the presented design, we do not keep in-memory the whole wavelet synopsis as before, but only the front nodes array and some helping structures that are going to be explained. Moreover, there is an analysis that shows that the helping structures we maintain do not incur a memory overhead larger than $logW$ and thus, the memory constraints still hold.

The main idea of the system is the following: as data items arrive, the Haar wavelet transform is dynamically computed. However, as soon as a new coefficient is created, it is persisted into the *WaveletStore*: a disk-based storage device. This way we can retrieve in the future coefficients that have been discarded. Please also note that as *write* operations are performed asynchronously, the ingestion of data into the disk does not delay the construction of the synopsis. In order to answer a query, we perform a lookup in the *Workload Index*. This is a structure that contains the materialized results for the queries of interest. For having fresh data in the Workload Index, we need to continuously update it. An update consists of computing the answer for every query $q_i \in Q$. As usually, the computation of a query $q_i$ consists of two parts: (i) one fully contained in range and (ii) a sub-tree that partially overlaps with it. For the part of the query that is fully contained in the range, we derive the answer by using the *front nodes array*. But for the last sub-tree, we can now retrieve the coefficients $c_j \in path_{q_i}$ from the WaveletStore. A naive solution would require $O\left(\lceil log\frac{W}{logW}\rceil\right)$ *GET* operations, i.e., as many as the coefficients in a maximal path. Nevertheless, such an approach would result in excessive accesses of the considerably more expensive secondary storage and would defeat the purpose of the fast, in-memory approximate query processing.

In the following, I present how we can limit disk accesses and create a fast system that can accurately answer workload-aware range queries under the sliding-window model.

### 5.6.1 Disk Access Patterns

The data organization on disk plays a crucial role on the system's performance. In this Section, we discuss how data should be stored and retrieved in order to obtain better response times compared to the naive solution where a logarithmic number of *GET* requests is required for each query. In the following discussion, we assume the WaveletStore to be any disk-based lightweight key-value store.

### Path-based Organization

The first approach for limiting the number of issued *GET* requests per query is to store in a single value all the coefficients that have been created at a specific time. Let us denote $P(t)$ the set of coefficients that are created at $t$. Thus, at each time $t$, we persist a key-value of the form:

$$(key, value) = (t, P(t))$$

As all $|P(t)|$ coefficients must have been created before they are persisted on disk, a buffer of size $|P(t)|$ should exist. According to Section 5.2, the arrival of two data items can trigger the creation of up to $O\left(\lceil log\frac{W}{logW}\rceil\right)$ new coefficients, and thus the memory overhead of this approach is $O\left(\lceil log\frac{W}{logW}\rceil\right)$.

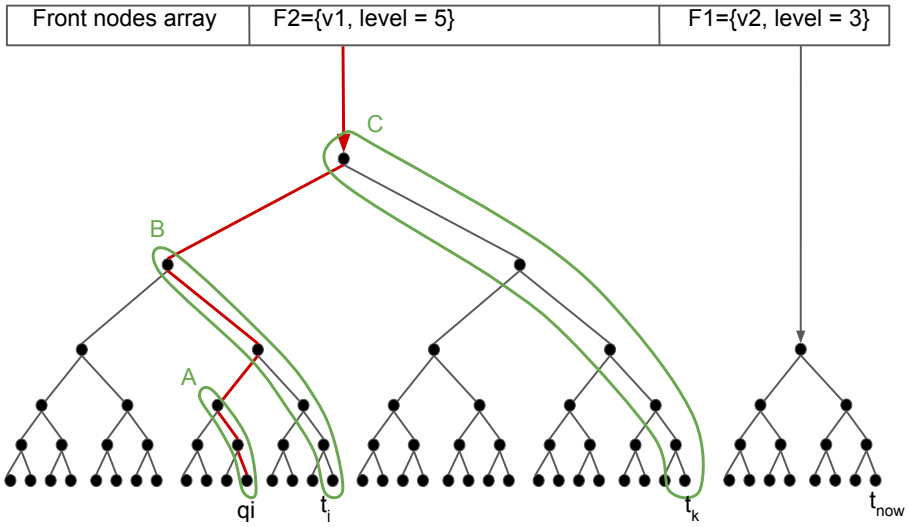



Figure 5.6: Example of the path-based data organization.

For the query answering, consider the example of Figure 5.6. The construction process of the error-tree implies that all coefficients that are surrounded by curve $A$ have been created at $t_{now} - q_i$, and thus are stored in a single key-value $(t_{now} - q_i, P(t_{now} - q_i))$. Similarly, the coefficients surrounded by $B$ have been created at $t_j$ and by $C$ at $t_k$. For computing $q_i$, we need to fetch from disk $P(t_{now} - q_i), P(t_j), P(t_k)$ and filter in memory the coefficients that belong to $path_{q_i}$. Algorithm 13 presents the exact procedure for achieving this task. This way, for the example of Figure 5.6, we perform 3 *GET* requests instead of the 5 that the naive approach requires. Lemma 14 places the bounds for the improvements this approach can bring.

**Lemma 14.** *For reconstructing the exact answer, the path-based organization needs* 1 GET *in the best case and has the same behavior as the naive approach in the worst case.*

*Proof.* The worst case is observed when the start of the query range is near to the leftmost path of a sub-tree. In that case, each coefficient of the path has been created at a different time and thus, $O\left(\lceil log_{\frac{W}{logW}} \rceil\right)$ *GETs* are required. The best case is observed, if we query the rightmost path of a sub-tree. The rightmost path is wholly created at a single time moment and can be fetched with a single request. □

**Algorithm 13:** Query Answering under the Path-based Data Organization

**input:** $t_{now}$, $q_i$, $maxLevel$
1 $time = t_{now} - q_i$;
2 $level = 1$;
3 $result = 0$;
4 $order = \lceil \frac{time}{2} \rceil$;
5 **while** $level \leq maxLevel$ **do**
6 $time = order * 2^{level}$; // a coefficient with index (l, o) is created at $time = o * 2^l$
$fetchedPath = waveletStore.get(time)$;
7 **for** *coefficient* $c_i$ *in* $fetchedPath$ **do**
8 **if** $c_i.level = level$ **then**
9 $result$ += $x_i c_i$;
10 **if** $\lceil \frac{order}{2} \rceil \cdot 2^{level+1} > time$ **then**
11 break;
12 $level$ += 1;
13 $order = \lceil \frac{order}{2} \rceil$;
14 $level$ += 1;
15 $order = \lceil \frac{order}{2} \rceil$;
16 return $result$;

### Subtree-based Organization

Similarly to Chapter 3, the subtree-based organization partitions the error-tree into sub-trees of fixed size $s$[3]. For persisting a sub-tree into the WaveletStore, we use as key the index of its root coefficient and as a value the sub-tree itself. Hence, we have key-values of the form:

$$(key, value) = ((r(S).level, r(S).order), S)$$

where $S$ denotes a sub-tree and $r(S)$ its root coefficient. This approach is more time-efficient as it achieves better locality and retrieves more "useful" coefficients with a single *GET* request. However, this is accomplished at the cost of a higher memory overhead. Figure 5.7 depicts an example where partitions of size 7 are annotated. We observe that this partitioning scheme divides

[3]The size of a partition is of the form $s = 2^k - 1, k > 0$

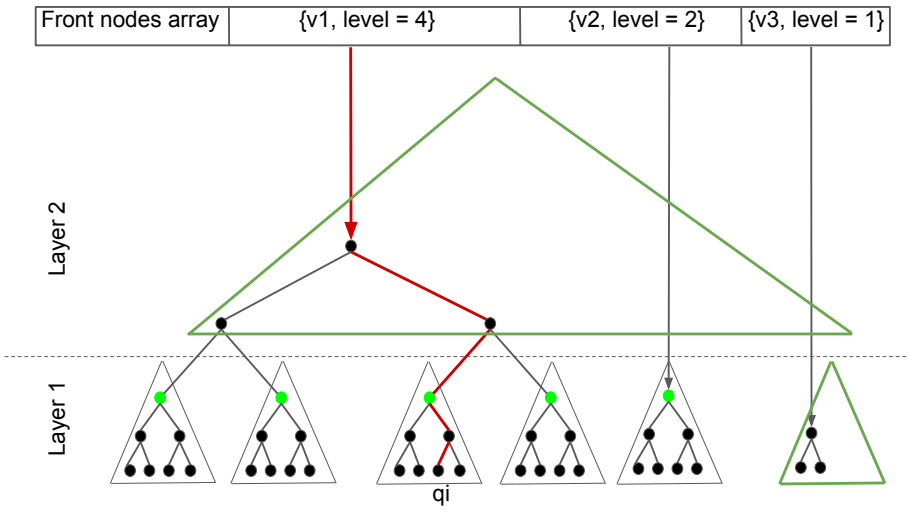



Figure 5.7: Example of the subtree-based data organization.

the error-tree into layers and at any time there can be at most one semi-completed partition at each layer. Partitions that are not yet fully completed should be buffered in memory. Buffering sub-trees until they are flushed to disk is responsibility of the *Sub-tree Buffer* component which is illustrated in Figure 5.5.

Since we can have at most one semi-completed partition at each sub-tree layer, the memory overhead the *Sub-tree Buffer* incurs is:

$$O(s \cdot \#Layers) = O(s \cdot \lceil log\frac{W}{logW} \rceil / \lfloor log(s) \rfloor)$$

A question that naturally arises is what is a proper value for $s$. The conducted experiments in Section 5.7 indicate that the larger the value of $s$ the higher is the memory consumption and the better is the query response time. Moreover, the experiments suggest that the optimal $s$-value is dependent on the window size $W$. Setting $s$ equal to a sub-linear function of the window size, such as $logW$, leads to $log^2W$ space complexity and thus, the constraint for poly-logarithmic memory is not violated.

For answering queries under this model, we traverse $path_{q_i}$ in a top-down manner and fetch from disk the sub-trees that intersect with the path. As some sub-trees of interest may reside in the *Sub-tree Buffer* and have not been persisted on disk yet, we also check if the query-path intersects with any of the sub-trees contained in memory. Algorithm 14 presents the technical details of this process.

The improvements on disk-accesses that can be achieved with the subtree-based approach are presented in Lemma 15.

**Algorithm 14:** Query Answering under the Subtree-based Data Organization

**input:** Partition size $s$, $t_{now}$, $q_i$, $maxLevel$

```
1  time = t_now − q_i;
2  result = 0;
3  level = 1;
4  rootLevel = log(s);
5  order = ⌈time/2⌉;
6  rootOrder = ⌈time/s⌉;
7  // compute path and subtree roots' indices;
8  path = new map();
9  subtreeRoots = new array();
10 subtreeRoots.add((rootLevel, rootOrder));
11 for cnt = 0; cnt < ⌈maxLevel/log(s)⌉; cnt++ do
12     while level ≤ min(rootLevel, maxLevel) do
13         path.put(level, order);
14         level += 1;
15         order = ⌈order/2⌉;
16     rootLevel+=log(s);
17     rootOrder = ⌈rootOrder/s⌉;
18     subtreeRoots.add((rootLevel, rootOrder));
19 // compute answer;
20 for i = 0;i < subtreeRoots.size() − 1;i++ do
21     index = subtreeRoots.size() − 1 − i;
22     rootLevel = subtreeRoots.get(index).level;
23     rootOrder = subtreeRoots.get(index).order;
24     level = rootLevel − log(s) + 1;
25     order = path.get(level);
26     if rootLevel in subtreeBuffer then
27         fetchedSubtree = subtreeBuffer.get(rootLevel);
28     else
29         fetchedSubtree = waveletStore.get((rootLevel, rootOrder));
30     for coefficient c_i in fetchedSubtree do
31         if c_i.level = level && c_i.order = order then
32             result += x_i c_i;
33             level += 1;
34             order = ⌈order/2⌉;
35 return result;
```

**Lemma 15.** *The subtree-based organization can reconstruct the exact answer without any disk access in the best case and with at most* $\lceil log(\frac{W}{logW}) \rceil / \lfloor log(s) \rfloor$ GET *requests in the worst case.*

*Proof.* The best case occurs when the query asks for one of the $s$ rightmost paths of the sub-tree. In that case, it can be answered solely based on the *Sub-tree Buffer* and no disk access is needed. In the worst case, the query asks for a path of a maximum height sub-tree ($\lceil log \frac{W}{logW} \rceil$) that does not have any overlap with the *Sub-tree Buffer*. Thus, there are $\lceil log(\frac{W}{logW}) \rceil / \lfloor log(s) \rfloor$ partitions/sub-trees that need to be fetched from the secondary storage. □

### 5.6.2 Maximizing Throughput

Selecting a good data placement in the secondary storage helps improving performance but there are still too many disk accesses that need to be made. The experiments of Section 5.7 show that even with the subtree-based organization the throughput of maintaining the wavelet structure is $8\times$ lower than the one achieved by the algorithm of Section 5.2 that works completely in-memory. For speeding-up construction, two key-ideas are used: (i) AQP and (ii) caching.

#### AQP

Algorithm 14 computes the contribution of the last sub-tree, that partially overlaps with the query range, to the final answer. To achieve that, the algorithm traverses the whole $path_{q_i}$ and computes an exact answer. However, this is too costly to afford as very frequent disk accesses take place. Thus, it is suggested to fetch only the $g$ topmost partitions that intersect with $path_{q_i}$. Retrieving from disk only a part of the path ($g \cdot s$ coefficients) leads to an approximate answer but favors performance. Intuitively, loading the topmost coefficients of a path yields better quality results, since these coefficients contribute to a larger part of the query range. Error guarantees are provided in exactly the same way as described in Section 5.2.3. The evaluation of Section 5.7 shows that there are very interesting speed-accuracy trade-offs to explore by experimenting with different $g$ values.

#### Caching

For further boosting the synopsis' construction throughput, a small cache is also used, as shown in Figure 5.5. Similarly to *Sub-tree Buffer*, the cache is allowed to contain at most a logarithmic number of sub-trees/partitions.

By observing Figure 5.7, we notice that each partition that intersects with $path_{q_i}$ for a given query $q_i$, is going to be present in many consecutive *GET* requests. That means that each partition will be fetched from disk multiple times. Retrieving over and over the same data incurs a significant overhead that we can mitigate with caching. Having available memory space for

$c$ partitions, the idea is to cache the $c$ partitions that are requested by the workload and will be "active" for the longest period of time. The time a partition stays active depends on its layer; a partition of size $s$ in layer $L$ has a time coverage of length: $(s+1)^L$. In the simple example of Figure 5.7, by considering a cache capable of storing a single partition, we can avoid 63 *GET* requests. For many distributions, retrieving from disk even only the topmost partition for a query can lead to very accurate results. This fact in combination with the caching mechanism create a fast and accurate system where the secondary storage is merely accessed for retrieving data values.

## 5.7 Experimental Evaluation

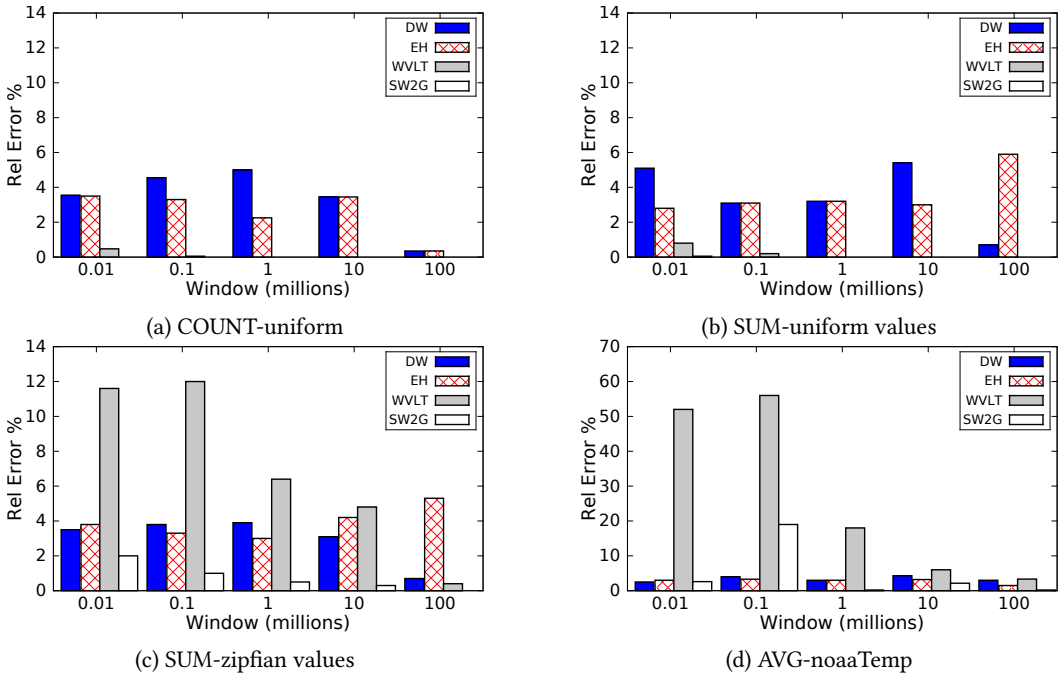
In this Section, the experimental evaluation of the proposed streaming algorithms is presented. Algorithms are compared in terms of accuracy and memory consumption. As accuracy we measure the real observed relative error, i.e.,

$$\text{Real Error} = \frac{|\text{precise answer} - \text{approximate answer}|}{\text{precise answer}} \cdot 100\%$$

For the disk-based approach of Section 5.6, the goal is to explore the speed-accuracy trade-off that the secondary storage incurs. Thus, only for this case, synopsis construction throughput experiments are also considered.

**Algorithms.** Henceforth, SW2G (Sliding Window Wavelets with Guarantees) denotes the in-memory, approximate algorithm that is presented in Section 5.2. SW2G is compared to the following techniques: (i) Exponential Histograms (*EH*) [DGIM02], (ii) Deterministic Waves (*DW*) [GT02] and (iii) the classic wavelet structure (*WVLT*) as discussed in [LTC10] for sliding-windows. EH and DW are deterministic structures that provide theoretically $\epsilon$-approximate results in COUNT and SUM queries for positive integers. However, it is proven [DGIM02] that for general SUM queries that also include negative numbers, providing theoretical guarantees requires $\Omega(W)$ bits and these methods cease to work. As the guarantees of the proposed method of this thesis are computed while constructing the synopsis and are not theoretical, we demonstrate the results of the proposed approach even for the case of arbitrary data values.

All single-stream algorithms are implemented in Java 8, except for the exponential histograms where the Scala implementation of [alg] is used. For the distributed algorithms, the Apache Flink 1.6 stream processing framework is employed. The Flink implementation for distributed exponential histograms is based on the Java code of [PGD12]. For the workload-aware case, where a disk-based secondary storage is required, a port of the LevelDB [lev] key-value store in Java has been used.




(a) COUNT-uniform

(b) SUM-uniform values

(c) SUM-zipfian values

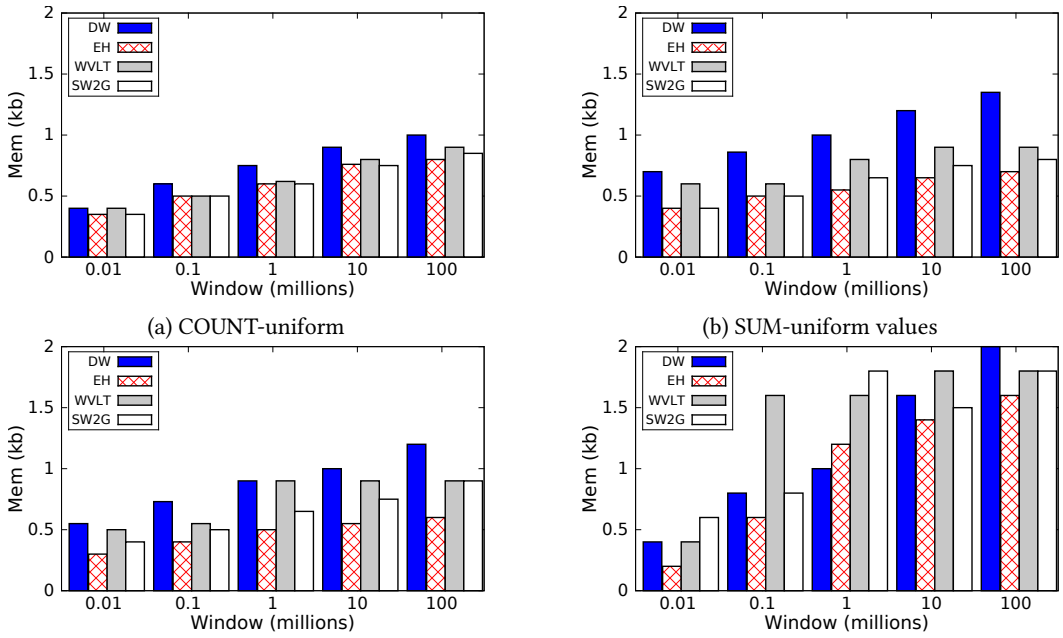
(d) AVG-noaaTemp

Figure 5.8: Relative error in streams of positive integers (query length = $W$).

**Queries.** The considered workloads are mainly range queries (COUNT, SUM, AVG) in the form described in Section 5.1. This is the most common query type in the sliding-window context. Moreover, the performance of wavelets in sliding-window aggregates has not been studied before. In order to demonstrate the generality of the proposed approach, in Section 5.7.4, aggregates over arbitrary ranges are also considered as well as point queries.

**Datasets.** For the assessment of the proposed algorithms, both synthetic and real data is used. Synthetic data is used for experimenting with various data distributions. The generated data values lie in the range $[0 - 1000]$ and follow a uniform, normal or highly biased ($s = 2$) zipf distribution. As real data, we use the sensor measurements provided by NOAA [noa]. From the various attributes contained in NOAA, the temperature (noaaTemp) and wind-speed (noaaSpeed) time-series are selected. NOAA time-series consist of both positive and negative numerical data.

**Platform.** All single-stream algorithms are executed on top of a server with 8 Intel(R) Xeon(R) CPU E5405 @ 2.00GHz processors and 8 GB of main memory. For the experiments on distributed streams, a cluster of 4 machines with the same processing and memory capabilities is used.




(a) COUNT-uniform

(b) SUM-uniform values

(c) SUM-zipfian values

(d) AVG-noaaTemp

Figure 5.9: Memory consumption in streams of positive integers (query length = $W$).

### 5.7.1 Positive Integers

In the first experiment, SW2G is evaluated over a single stream of positive integers. As this is the only case where EH and DW can be applied, a direct comparison among the various methods can be performed.

Figure 5.8 presents accuracy results for various data distributions and window sizes. We consider streams of $400$ millions data points and window sizes in the range of $[10k, 100M]$. At random times, we query each structure for the COUNT, SUM or AVG of the stream elements over the last $W$ time units. In the case of the noaaTemp dataset, a more complex query is computed: we filter the stream *on the fly* and compute the average temperature only for tuples having a temperature larger than $86F$. In favor of a fair comparison, algorithms are tuned to use approximately the same amount of memory. In EH and DW, the tuning knob of memory consumption is the guaranteed error $\epsilon$ and for the wavelet-based techniques, the available budget $B$.

EH and DW respect the theoretical guarantees and both achieve an average error near $4\%$ for all datasets. The vanilla wavelet method, while performing well in uniform distributions, it presents considerably large errors for the other two datasets. Particularly for noaaTemp, as WVLT can reach up to a $60\%$ relative error, it cannot provide an acceptable solution to the problem. By being near precise in all demonstrated cases, SW2G appears to be the best alternative for approximating the examined datasets.

Please recall that in sliding window range queries, an error is introduced only due to the overlap of the query range with the last bucket of the active window. Techniques like EH and DW control the size of the last bucket in a way that provides theoretical guarantees. By putting a constraint on the maximum level of a sub-tree, SW2G also controls the size of the last bucket. WVLT is not designed with range queries in mind; the whole window can be covered by a single tree of size $W$. Thus, WVLT presents an unstable behavior where quality highly depends on the current state and structure of the error-tree.

The overlap with the last bucket is also the cause for the high quality results of SW2G compared to EH and DW. Both these techniques assume that half of the last bucket's items lie in the range of interest. On the other hand, wavelet-based techniques can more accurately approximate the number of items that should be considered. By combining the powerful wavelet structure and the idea of limiting the maximum size of an error-tree, SW2G manages to present the best results in all cases.

Figure 5.9 illustrates the corresponding memory consumption. We observe that as window size increases, we need to consume more memory in order to preserve error guarantees. We see that DW is the most expensive among the evaluated methods. Moreover, we observe that COUNT queries use slightly less memory than SUM ones and AVG queries need the largest amount of memory since we have to maintain two structures for each algorithm: one that keeps track of counts and one for sums. However, in all cases, memory consumption is negligible. In the case of $W = 100M$, the footprint of the exact solution is $400$ MB, while all approximation techniques need only around a single kilobyte. Especially in the case of SW2G, $1$ Kb is enough for achieving a relative error lower than $1\%$ in all demonstrated cases.

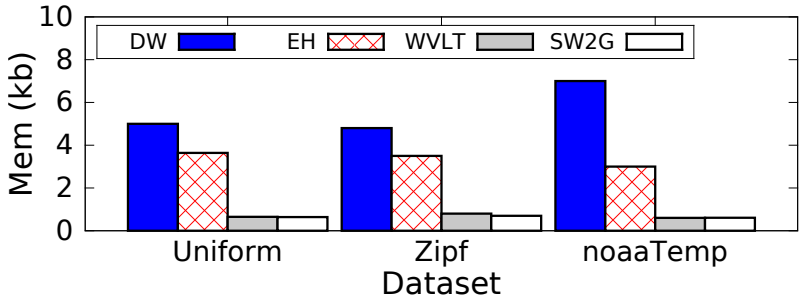


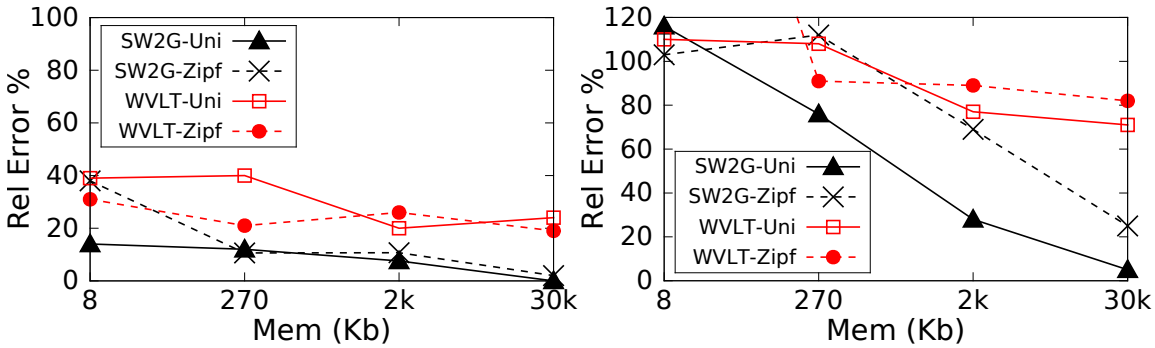

Figure 5.10: Memory for $\epsilon = 0.01$

Figure 5.10 illustrates the memory EH and DW need in order to achieve the same performance as SW2G when $W = 10M$. For this purpose, we set $\epsilon = 0.01$ for both EH and DW and issue SUM queries to all datasets. In the case of noaaTemp, we notice that DW needs $7\times$ and EH $4\times$ the memory that SW2G requires.

As window size does not affect accuracy, in all subsequent experiments we set $W$ to $10M$.

### 5.7.2 Streams of Generic Numerical Data

In the case of positive numbers, we demonstrated that the proposed approach outperforms existing techniques. In this Section, the applicability and efficiency of SW2G are examined in more general cases, where the stream also includes negative values. We experiment with SUM queries in real and synthetic data. Uniform and zipf synthetic distributions are used, where each data point $d_i$ is drawn from range $[0, 1000]$ and is converted to the corresponding negative value $-d_i$ with a probability of $\frac{1}{2}$. Since EH and DW do not work for negative numbers, they are not considered for these experiments.




(a) Query length = $W$ (b) Query length =$W/8$

Figure 5.11: Relative error in streams of arbitrary numerical data.

First, queries of length $W$ and $W/8$ are computed on the noaaTemp and noaaSpeed datasets. As the value distribution of the NOAA datasets does not present a great variance, it can be easily approximated by wavelets. As such, both SW2G and WVLT achieved relative errors less than $1\%$ in both workloads.

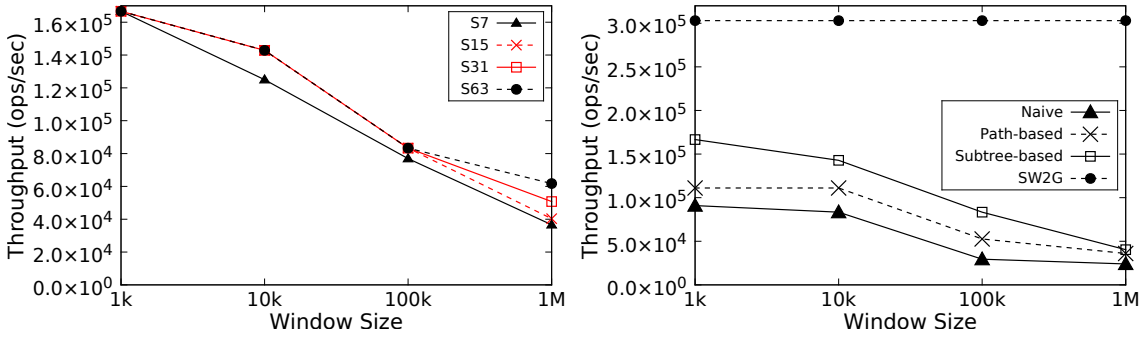
In order to stress wavelet algorithms, the described synthetic distributions are used. As each subsequent data point can vary from $-1000$ to $1000$ large discontinuities appear and the distribution becomes hard to approximate.

Figure 5.11 illustrates relative error with respect to the consumed amount of memory. We observe that for both distributions and query lengths, SW2G converges better than WVLT as memory increases. In the case where the query is applied over the whole window, a budget size of $W/10$ is enough to achieve an error less than $10\%$ both for the uniform and the zipfian data.

### 5.7.3 Evaluating Workload Aware Synopses

In this Section we evaluate the described system of Section 5.6 and explore the trade-offs it can achieve between construction throughput and accuracy. For all the experiments of this Section, we consider a workload of a single query, i.e., $Q = \{q_i\}$. The query $q_i$ is randomly selected in

the range $[1, W]$. As it is shown, even a single query is enough to showcase the implications of the secondary storage as well as the worst- and best-case performance of the proposed system.




(a) Impact of partition size

(b) Compare various disk access patterns

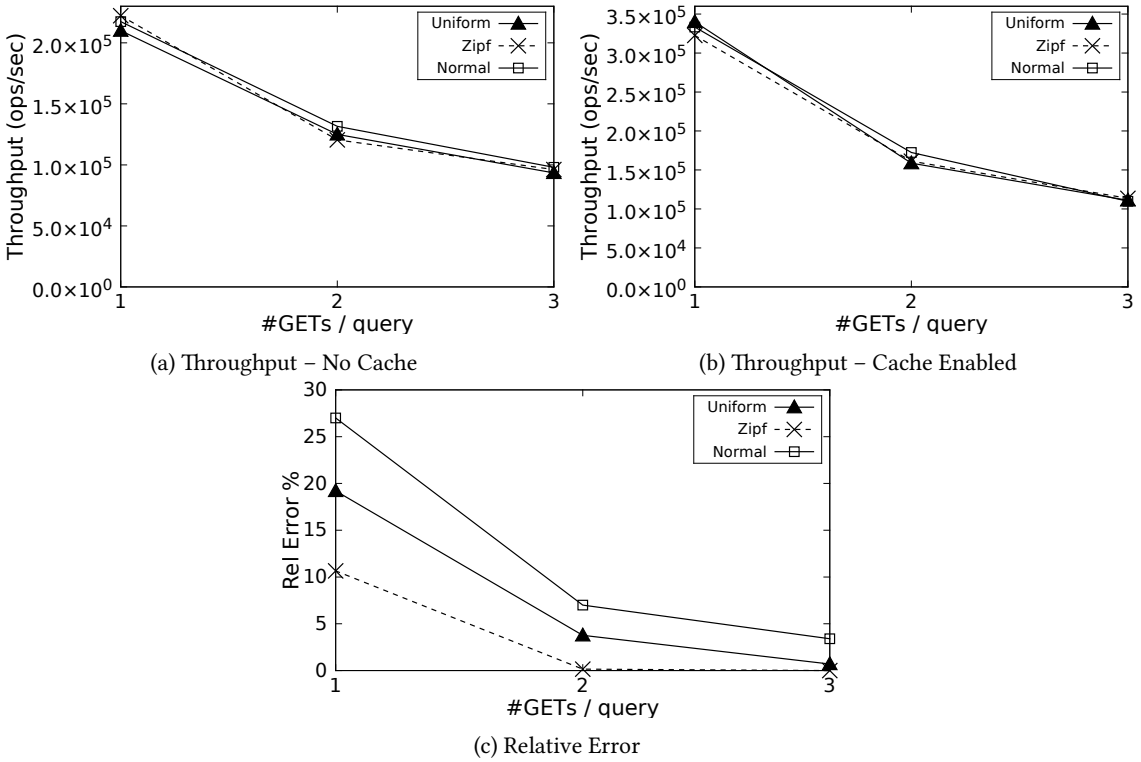
Figure 5.12: Experiments on disk placement parameters.

### Disk Organization Parameters

First, we assess the proposed methods for organizing data in the secondary storage. As here we want to investigate the pure impact of disk, for the experiments of this Section, we compute an exact answer by fetching the whole $path_{q_i}$ and the caching mechanism is turned off.

Figure 5.12-(a) illustrates the throughput of constructing a synopsis when the subtree-based organization is used and for various partition sizes. In the Figure, we denote with $S_k$ a partition that contains $k$ wavelet nodes. The results suggest that in general larger partitions achieve better throughput. We observe that the smallest partition $S_7$ is always outperformed and in the case of a window $W = 1M$, performance results strictly follow the order of partition sizes; the run for the largest partition $S_{63}$ is the fastest one, the run for $S_{31}$ comes second, etc. However, we also notice that performance-wise, the optimal partition size is dependent on the window. For windows smaller than $W = 1M$, further increasing the partition size does not have an impact on throughput. For the remainder of this Section, partitions of 15 wavelet coefficients are considered. Larger partitions may achieve better running-time results but consume more memory. As we want the *Sub-tree Buffer* to be of poly-logarithmic space in the window size, partitions have been selected in a way to achieve a good trade-off between running-time and memory consumption.

Figure 5.12-(a) presents a comparative analysis among the various data organizations on disk. The subtree-based organization is $1.5\times$ as fast as the path-based and $2\times$ as fast as the naive one for all evaluated window sizes. Nevertheless, we notice that as the window size increases, throughput drops. When the secondary storage is involved, computing the exact answer for




(a) Throughput – No Cache

(b) Throughput – Cache Enabled

(c) Relative Error

Figure 5.13: Impact of # GETs/query on throughput and relative error.

a window $W = 1M$ is 7.5× more expensive, with respect to throughput, than using the in-memory SW2G.

### Exploring the Time-Accuracy Trade-off

In Figure 5.11, we noticed that under some circumstances SW2G does not behave well. More specifically, when the stream contains negative numbers, the budget space is small and the query range is much smaller than $W$, the relative error increases considerably.

In this Section, it is demonstrated how the proposed system comes to the rescue when workload information exists. Based on the above observations, first, we create adversarial conditions for SW2G and test its running-time and accuracy for various distributions. The results for SW2G are shown in Table 5.1. Figure5.13 shows the corresponding results when the proposed workload-aware system is employed.

Table 5.1: Running-time and accuracy performance of SW2G for various distributions

| Metric | Uniform | Zipf | Normal |
|---|---|---|---|
| Throughput (ops/sec) | 400K | 385K | 345K |
| Relative Error % | 60 | 390 | 159 |

For all examined distributions, Figure 5.13-(c) shows that even a single *GET* query to the secondary storage can ensure a relative error lower than $30\%$; that is a $90\%$ improvement compared to SW2G. When two *GETs* are issued per query, the corresponding error drops to lower than $10\%$, while three *GETs* provide an almost exact result.

Figure 5.13-(a) presents the corresponding throughput results when caches are disabled. The first *GET* request to the disk has a cost of $40\%$ performance degradation compared to SW2G. However, the situation is much better when caches are enabled. In that case, the cost in throughput is less than $15\%$. Thus, for all distributions, the proposed system can achieve an error lower than $30\%$ with minimal performance overheads.

### 5.7.4 General Range and Point Queries

SW2G and WVLT are also evaluated in other query types such as aggregates over arbitrary ranges and point queries.

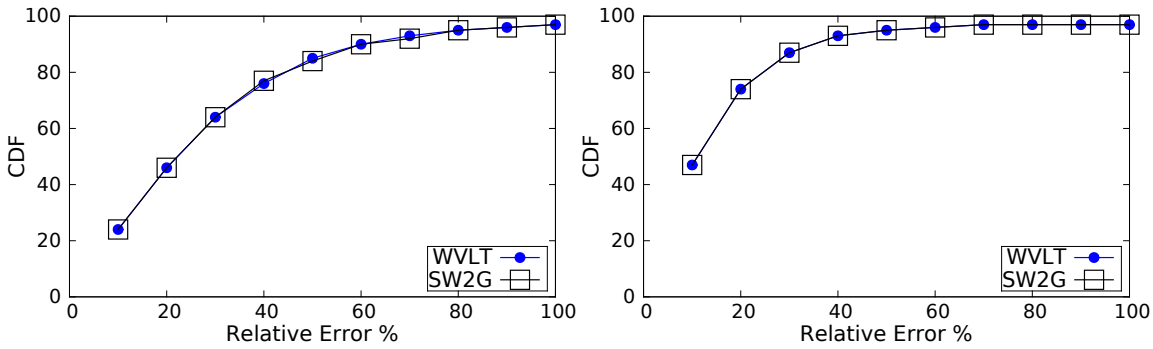
Table 5.2 shows the results for a workload of random AVG queries where the limits of the queried ranges are selected at random. For this experiment, a $200Kb$-sized synopsis is used. Moreover, all datasets contain both positive and negative values.

Table 5.2: Relative error for AVG queries with random ranges

| Dataset | SW2G | WVLT | % Gain |
|---|---|---|---|
| uniform | 27 | 126 | 78 |
| zipf | 53 | 61 | 13 |
| noaaTemp | 0.12 | 1.04 | 88 |
| noaaSpeed | 0.75 | 5.4 | 86 |

Depending on the data distribution, the performance of both algorithms varies. However, SW2G consistently outperforms WVLT, demonstrating this way the contribution of this thesis to the wavelet structure for tackling range queries.

Figure 5.14 demonstrates the results for point queries. The applied workload in this case is the following: Every $W$ time units, we ask for the value of every item in the range $[t - W, t]$, where $t$ is the current time. Both algorithms achieve the same accuracy in all examined cases. Thus, while optimizing for range queries, the performance of the proposed algorithm in point queries is not compromised. As noticed in Figure 5.14a, the distribution of the noaaSpeed dataset needs more space than $W/100$ in order to be accurately represented. However, error drops as space budget is increased. Having available $W/10$ of memory leads to an error less than $20\%$ for the $70\%$ of the workload.

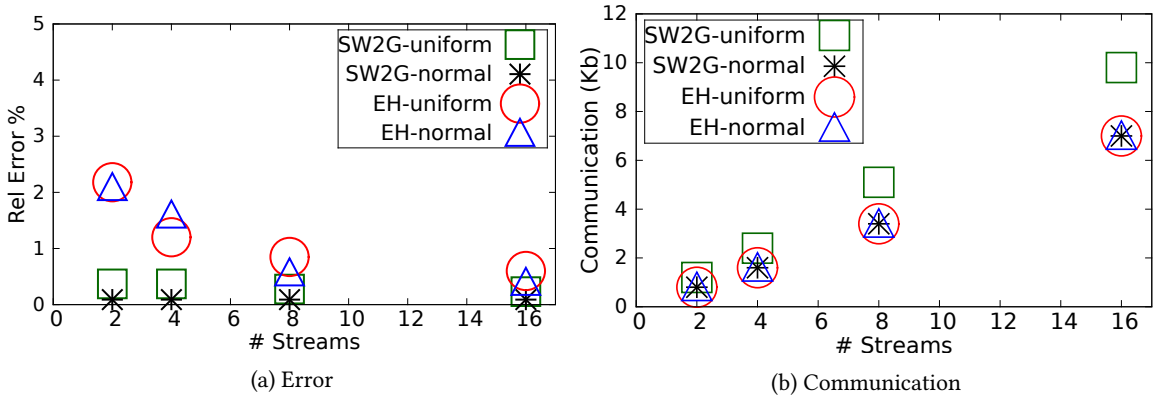



(a) noaaSpeed, $B = W/100$ (b) noaaSpeed, $B = W/10$

Figure 5.14: CDF of relative error in point queries.

### 5.7.5 Distributed Streams

This Section examines the behavior of SW2G in a distributed environment of multiple streams. In this scenario, we track range queries in the average of the streams. Each stream maintains a local synopsis; a coordinator node collects wavelet coefficients from all streams and composes a global synopsis which is used to answer queries. SW2G is compared with the distributed version of EH which is described in [PGD12]. For distributed exponential histograms we set an error of $\epsilon = 0.1$ both for the coordinator and all remote streams.




(a) Error (b) Communication

Figure 5.15: Relative error and communication cost in distributed streams.

Figure 5.15 shows the real relative error and the communication cost for synthetic data of uniform and normal distributions. Results are presented for 2 up to 16 streams. For each setup we plot the average error of the issued workload and the total bytes sent over the network each time the streams emit their local synopses. Although EH are configured with $\epsilon = 0.1$ and according to [PGD12] are expected to have an error up to $2\epsilon+\epsilon^2 = 21\%$, they present a maximum error of only 2%. SW2G performs even better and is almost exact in all cases. Furthermore, the guarantees

it provides do not exceed 9%. As expected, communication increases linearly to the number of streams for both techniques.

CHAPTER 6

# Related Work

This Chapter surveys the related work to this thesis. I present and discuss various techniques and systems for building approximate synopses over massive data. While this dissertation focuses on wavelets, in favor of completeness and in order to give an overview of the whole AQP landscape, a short description of different techniques such as sampling and sketches is also provided. The techniques are compared to each other and the pros and cons of each of them are discussed.

For the wavelet case, a complete review of the related database literature is presented. There is a discussion about all major approaches for building wavelet synopses over batch and streaming data. For the batch processing case, we put emphasis on techniques that optimize non-Euclidean errors, since they can be directly compared to the proposed algorithms of this thesis (Chapters 3, 4). For streaming synopses, apart from wavelet-based techniques, we also consider algorithms that work under the sliding-window model and thus are related to the ones presented in Chapter 5.

While up to this point, the discussion is mainly restricted to the algorithmic level, Section 6.4 presents a list of research prototype systems that enable AQP in practice.

## 6.1 The AQP Landscape

The main four families of data synopses are: random samples, histograms, sketches and wavelets. Since the wavelet bibliography is going to be extensively discussed in a separate Section, here the discussion is restricted to the remaining three categories.

### 6.1.1 Sampling

Random samples are perhaps the most fundamental synopses for AQP, and the most widely implemented [AMP+13, RMW+16, PMSW18]. The use of random samples as synopses for AQP has an almost 35 year history in the database research literature, with the earliest major-venue database sampling paper being published in 1984 [PSC84]. This arguably makes sampling the longest studied method for AQP. Many different methods of extracting and maintaining samples [CDN07, BDM02] of data have been proposed, along with multiple ways to build an estimator for a given query.

For small tables, drawing a sample can be done straightforwardly. For larger relations, which may not fit conveniently in memory, or may not even be stored on disk in full, more advanced techniques are needed to make the sampling process scalable. For disk-resident data, sampling methods that operate at the granularity of a block rather than a tuple may be preferred. Existing indices can also be leveraged to help the sampling. For large streams of data, considerable effort has been put into maintaining a uniform sample as new items arrive or existing items are deleted. Finally, "online aggregation" algorithms [HHW97, LWYZ16, CCA+10] enhance interactive exploration of massive datasets by exploiting the fact that an imprecise sampling-based estimate of a query result can be incrementally improved simply by collecting more samples.

In the exceptional survey of Cormode et al. [CGHJ12], the advantages and disadvantages of sampling are summarized. The sampling benefits include:

- Simplicity: Conceptually, it is very simple to understand the idea of drawing items at random from a dataset, then scaling up the result of a query over the sample to guess the result of applying the query to the whole dataset.
- Pervasiveness: Sampling is widely supported by database systems (Section 6.4), and support for sampling is part of the current SQL standard (SQL-2016).
- Extensive theory: Almost 100 years of prior research in survey sampling can be applied directly to sampling massive data.
- Flexibility: A sample is a very general-purpose data structure and as such, the same sample can be used to answer a wide variety of arbitrary queries.
- Insensitivity to dimension. The accuracy of sampling-based estimates is usually independent of the number of attributes in the data.
- Ease of implementation: Because sampling commutes with many of the common query operations, it is possible to use a database engine itself to evaluate a query over a sample.

However, sampling also has its own drawbacks. Specifically:

- Poor performance to highly selective queries. Sampling relies on having a reasonable chance of selecting some of the data items that are relevant for answering a query. If only ten tuples out of one million contribute to the answer, then a 1% sample of the data is unlikely to select any of them.

- Worse running time performance than other types of synopses. Since the size of a sample is proportional to the size of the original data, in the case of large datasets, even a 1% sample can be considerably larger than a histogram or a sketch over the same data.

- Sensitive to skew and outliers.

### 6.1.2 Histograms

The histogram is a fundamental object for summarizing the frequency distribution of an attribute or combination of attributes. As such, they have been an integral component of query optimizers since at least the mid-1990s [Cha98], and are often used by the information management and statistical communities for purposes of data visualization. As discussed in the review paper of Ioannidis [Ioa03], use of histograms for data summarization and visualization goes back to the 18th century, and the term "histogram" itself was coined by statistician Karl Pearson in the 1890s. Histograms have been studied in the database literature for over 30 years, starting with the paper of Piatetsky-Shapiro and Connell [PSC84].

The most basic histograms are based on a fixed division of the domain (*equi-width*), or using quantiles (*equi-depth*), and simply keep statistics on the number of items from the input which fall in each such bucket. Nevertheless, many more complex methods have been designed, which aim to provide the most accurate summary possible within a limited space budget [BLS+08, BSS07]. At query time, the summary and bucket information is used to approximately reconstruct the data in the bucket in order to answer the query. Schemes differ in: (i) how the buckets are chosen, (ii) what statistics are stored, (iii) how estimates are extracted, and (iv) what query classes are supported. They are quantified based on their space and time requirements, as well as on the provided accuracy guarantees.

The one-dimensional case is at the heart of histogram construction, since higher dimensions are typically handled via extensions of one-dimensional ideas. Beyond equi-width and equi-depth, end biased, high biased, maxdiff and other generalizations have been proposed. For a variety of approximation-error metrics, DP methods can be used to construct an optimal histogram, subject to an upper bound on the allowable memory space. An important class of histograms that are constructed with DP and are really popular for several selectivity estimation problems are the "V-optimal histograms" [IP95]. When the quadratic cost of DP is not practical, approximate methods can be used [GMP02, GGI+02, DGR01].

Histograms most naturally answer range–sum queries. They can also be used to approximate more general classes of queries, such as aggregations over joins. Various negative theoretical and empirical results indicate that one should not expect histograms to give accurate answers to arbitrary queries [CGHJ12]. Nevertheless, due to their conceptual simplicity, they can be effectively used for a broad variety of estimation tasks, including set-valued queries, real-valued data, and aggregate queries over predicates more complex than simple ranges.

The advantages of histograms are the following:

- Easy to interpret: which makes it easier for system developers to construct and analysts to query them.
- Well-established theory: Histograms have been studied in the database literature for over 30 years and there is a strong notion of optimality for various query classes.

Histogram drawbacks include:

- They do not adapt well to high dimensional data.
- They do not fit well in the streaming model.
- Many histogram techniques have several parameters which have to be set a priori, such as the number of buckets, statistics to keep within each bucket, and other parameters that determine when to split or merge buckets.

### 6.1.3 Sketches

While having the shortest history, sketching techniques have undergone extensive development over the past few years. They are especially appropriate for streaming data, in which large quantities of data flow by and the sketch summary must continually be updated quickly and compactly. Sketches are designed so that the update caused by each new piece of data is largely independent of the current state of the summary [CGHJ12]. This design choice makes them faster to process, and also easy to parallelize.

The basic properties that characterize a sketching algorithm are:

- The supported queries. Unlike samples, we cannot simply execute a query on the sketch. Instead, we need to perform a specific procedure to obtain an approximate answer.
- The sketch size. A sketch has one or more parameters which determine its size. A common case is where parameters $\epsilon$ and $\delta$ are chosen by the user to determine the accuracy (approximation error) and probability of exceeding the accuracy bounds, respectively.

- Update/Query speed. Dense transforms [AMS96] affects all entries in the sketch, and so takes time linear in the sketch size. But typically the sketch transform can be made very sparse [CM05], and consequently the time per update may be much less than updating every entry in the sketch.

Similarly to histograms and wavelets, sketches present a fundamental difference with sampling regarding on how the data is observed. A sample "sees" only those items which were selected to be in the sample whereas the sketch "sees" the entire input, but is restricted to retain only a small summary of it. Therefore, to build a sketch, we must be able to perform a single linear scan of the input data (in no particular order).

As mentioned, each sketching algorithm targets specific query types. Some popular sketch categories are: (i) the "Set sketches" (e.g., Bloom Filters [Blo70]), (ii) the "Frequency based sketches" (e.g., Count-Min Sketch [CM05]) and (iii) sketches for COUNT-DISTINCT queries (e.g., Flajolet-Martin Sketches [FM85, FFGM07]).

Set sketches answer membership queries. Specifically, the Bloom Filter guarantees no false negatives, but may report false positives. Frequency based sketches are concerned with summarizing the observed frequency distribution of a dataset. From these sketches, accurate estimations of individual frequencies can be extracted. This leads to algorithms for finding approximate "heavy hitters" – items that account for a large fraction of the frequency mass – and quantiles such as the median and its generalizations. The same sketches can also be used to estimate the sizes of joins between relations, self-join sizes, and range queries. Such sketching algorithms can also be used as primitives within more complex mining operations [PGD12], and to extract wavelet and histogram representations of streaming data [GKMS03, CGS06, GKMS01]. Finally, problems relating to estimating the number of distinct items present in a sequence have been heavily studied in the last three decades. The Flajolet–Martin (FM) sketch is probably the earliest, and perhaps the best known method for approximating the distinct count in small space [FM85].

The main advantage of sketches is that they are really efficient for high speed streams of data. However, this comes at the cost of less flexibility. The main limitation of sketching techniques – especially in contrast to the general-purpose sampling paradigm – is that each sketch tends to be focused on answering a single type of query.

## 6.2 Wavelets for AQP

Wavelets have some commonalities with histograms as both techniques partition the input space and compute simple statistics for sub-regions of the input domain. A theoretical difference is that whereas histograms excel at capturing the local structure of contiguous data values, wavelets are

particularly well suited to capturing non-local structures. Moreover, due to the linearity of the Haar wavelet transform, wavelets are easier to maintain than histograms in streaming scenarios.

In the following of this Section, we make a literature review on existing techniques that construct wavelet synopses for maximum error metrics both for one- and multi-dimensional data. In addition, algorithms that use wavelets for online synopses in streams are also discussed.

### 6.2.1 Wavelets for One-Dimensional Data

In [GG02], a probabilistic DP algorithm is proposed. The running-time of the algorithm is $O\left(N\delta^2 Blog\left(\delta B\right)\right)$. However, as there is always a possibility of a "bad" sequence of coin flips, this approach can lead to a poor quality synopsis. As an improvement, a deterministic DP approach is proposed in [GK04]. Unfortunately, the optimal solution provided has a high time complexity of $O\left(N^2 BlogB\right)$. These solutions are very expensive in terms of time and space and such requirements render them impracticable for the purpose of quick and space-efficient data summarization.

In order to decrease space complexity, Guha introduces a generally applicable, space efficient technique [Guh05] for all these DP-based approaches, that needs linear space for the synopsis construction but at the cost of a $O\left(N^2\right)$ running time.

A more recent and sophisticated approach is presented in [KM07]. Karras and Mamoulis devise Haar+: a modified error-tree, whose structure gives more flexibility on choosing which coefficients to keep. For the thresholding, a DP algorithm with running-time complexity $O\left(\left(\frac{\Delta}{\delta}^2\right) NB\right)$ is presented.

A different approach is proposed in [KSM07]. The authors design a solution that tackles the space-bound problem (Problem 1 defined in Chapter 2) by running multiple times a DP algorithm for the dual problem [KSM07, Mut05, PZHM09]. The resulting complexity is $O((\frac{\mathcal{E}}{\delta})^2 N(log\epsilon^* + logN))$, where $\mathcal{E}$ is the minimum maximum error that can be achieved with $B-1$ coefficients and $\epsilon^*$ is the real maximum error. This algorithm is considered to be the current state-of-the-art for the problem, as it provides the optimal data reconstruction for the given budget and has the best running-time complexity among the corresponding DP algorithms.

Similar DP algorithms have been also proposed for the minimization of general distributive errors like the $L_p$ norm [GK05, GH05]. The proposed framework of Chapter 3 for the parallelization of DP algorithms can be seamlessly used to speedup the execution of these algorithms too.

In order to decrease running-time, greedy algorithms have been proposed [KM05, MP03] for the minimization of the maximum absolute and relative error with worst-case running-time complexities of $O\left(Nlog^2 N\right)$ and $O\left(Nlog^3 N\right)$ respectively. These algorithms present almost linear

behavior in practice and require less memory capacity than most of the DP-based ones. Nevertheless, since they run in a centralized fashion, as data scales close to the memory constraints of the machine, their performance significantly deteriorates. Moreover, they have inherent difficulties in their parallelization and thus, the decomposition to local sub-problems is not an easy task to accomplish.

In the literature, there is a lack of parallel implementations for wavelet synopses. Only the work of Jestes et al. [JYL11] considers MapReduce-based algorithms for the wavelet decomposition. However, the algorithms of [JYL11] target only the $L_2$-error minimization, which is a considerably easier task to accomplish.

### 6.2.2 Synopses for Multidimensional Data

Although there is a lot of research for the one-dimensional case, few attempts have been made to approach the multidimensional version of the problem. In [CGRS01], algorithms for multidimensional wavelet decomposition and thresholding are presented. However, only conventional thresholding is studied and there is no proposed algorithm for maximum-error metrics.

In [GK04] the authors present deterministic, exact and approximate DP-based algorithms for the problem. The most time-efficient algorithm is a $(1+\epsilon)-$approximation algorithm with running time: $O(\frac{logR_Z}{\epsilon}2^{2^D+3D}Nlog^2NBlogB)$ for a D-dimensional dataset. Despite the optimal quality, the running-time of these algorithms is prohibitive for real-world scenarios even for small data dimensionalities (i.e., $D \in [2,5]$, where wavelet-based data reduction is typically employed[1]).

A multi-dimensional extension of the DP algorithm that targets the Haar+ tree is presented in [KM08]. The algorithm has a running-time complexity of $O\left(2^{2^D}\left(\frac{\Delta}{\delta}^{2^D}\right)NB\right)$. While this result improves a lot upon the work of Garofalakis et al. [GK04] , the constant term $2^{2^D}\left(\frac{\Delta}{\delta}^{2^D}\right)$ can easily explode even for small dimensionalities. Moreover, as $\Delta$ denotes the difference between the minimum and maximum values of the data, the performance of this algorithm is very sensitive to data distribution.

For dealing with multi-dimensional datasets, the authors of [MP03] propose mapping all data to one dimension by using a space filling curve. Then, a one-dimensional algorithm can be applied. The drawback of this approach is that it destroys data locality and thus can lead to sub-optimal quality results.

In [ZPH09], an algorithm of $O(N)$ time complexity is proposed for solving the dual problem. Once again, a space-bound synopsis can be constructed by employing the technique in [KSM07].

[1]Due to the "dimensionality curse", wavelets and other space-partitioning schemes become ineffective above 5-6 dimensions.

However, the algorithm of [ZPH09] is presented in a centralized setting and there is no evidence of its performance on large scale datasets.

A similar algorithm is also presented in [LHZ+16] for image compression and thus only covers two-dimensional datasets. However, the algorithm is still applicable on small datasets.

### 6.2.3 Wavelets on Streams

All approaches discussed thus far refer to batch jobs, where algorithms are applied to static data. In [GKMS01, GKMS03], the authors compute $L_2$-optimal wavelets on streams. As they find it more challenging, they put more emphasis on handling the unordered cash register stream model. In [CGS06], a similar sketching technique, that allows more efficient updates, is presented for the same problem. Streaming techniques have also been proposed for the optimization of the $L_\infty$ norm. In [GH05, GH08], the authors present optimal algorithms for computing the optimal error in a streaming way for a broad category of non-Euclidean errors. Nevertheless, as dynamic programming needs a recursive top-down procedure in order to construct the final synopsis, these algorithms are not suitable for the scenario of an unbounded stream where inactive elements are permanently discarded. For $L_\infty$-minimization, a greedy streaming algorithm has appeared in [KM05]. However, opposed to the algorithms of Chapter 5, the work of [KM05] does not support sliding-window queries.

The only wavelet-based algorithm that exists in the literature and considers the sliding-window model is the work presented in [LTC10]. This work mainly covers point queries and it does not take into account range queries such as COUNT and SUM, which are the most basic and common queries in sliding-window streams.

## 6.3 Sliding-Window Streams

The bulk of existing work on the sliding-window model has focused on algorithms for efficiently maintaining simple statistics, such as COUNT and SUM. By *efficiently*, we mean sub-linear space and time (typically, poly-logarithmic) in the window size $W$. Exponential histograms [DGIM02] are a state-of-the-art deterministic technique for maintaining $\epsilon$-approximate counts and sums over sliding windows, using $O\left(\frac{1}{\epsilon} log^2 W\right)$ space. Deterministic waves [GT02] solve the same basic aggregates problem with the same space complexity as exponential histograms, but improve the worst-case update time complexity to $O(1)$. In the same work [GT02], Gibbons also presents randomized waves to tackle COUNT-DISTINCT queries. Randomized waves, as most randomized sketching techniques, are easily parallelizable and composable (in distributed settings), but come with increased space requirements. In [XTB08], the authors describe a randomized, sampling-based synopsis, very similar to randomized waves, for tracking sliding-window

COUNT and SUM queries with out-of-order arrivals. As in randomized waves, the space requirements are also quadratic in the inverse approximation error. To address the high cost associated with randomized data structures, Busch and Tirthapura propose a deterministic structure for handling out-of-order arrivals in sliding windows [BT07]. Similar to other deterministic structures, this structure does not allow composition and focuses only on basic counts and sums. Finally, Chan et al. [CLLT12] investigate continuous monitoring of exponential-histogram aggregates over distributed sliding windows. The main contribution of their work lies in the efficient scheduling of the propagation of the local exponential-histogram summaries to a coordinator, without violating prescribed accuracy guarantees.

Work has also been done on sketching techniques that are suitable to answer more complex queries such as k-medians [BDMO03], heavy hitters, inner products and self-joins [PGD12, SMTZ17, RBM15]. However, as the majority of these techniques employ under the hood algorithms for computing basic aggregates, this dissertation focuses only on point queries and basic aggregates like COUNT, SUM and AVG.

## 6.4 Systems

Perhaps the introduction of approximation technology into DBMSs dates back to the first cost-based query optimizers in 1970s. A query optimizer needs to quickly evaluate the size of intermediate query results, in order to evaluate competing query plans. Importantly, these result sizes need to be determined only to a degree of accuracy sufficient for query-plan comparison. Initial estimation schemes were rather crude, and usually assumed that the frequency distribution was uniform. However, over time, systems began to employ more elaborate techniques that are efficient even in the presence of highly non-uniform distributions. A discussion of statistics and query optimizers can be found in [HILM09].

While there is a large adoption of synopses in commercial DBMSs for query optimization, there have only been a few successes [RMW$^+$16, SZB$^+$16] in the AQP area with the majority of them being sampling-based. Historically, probably the first AQP research prototype is the Aqua system [AGPR99a], developed at Bell Labs in the 1990s. Since then, a plethora of academic systems have appeared and recently there is also an interest from the industry world.

BlinkDB [AMP$^+$13] and its successor SnappyData [RMW$^+$16] use an optimization framework based on Mixed Integer Linear Programming (MILP) to precompute a set of multidimensional, multi-resolution samples. Then, they employ a dynamic sample selection strategy that selects an appropriately sized sample based on a query's accuracy and/or response time requirements. As SnappyData can also handle streaming data, it also uses Count-Min sketches to approximate frequencies. SnappyData has recently entered the industry world through a spin off company.

VerdictDB [PMSW18] is a sampling-based engine that follows a different logic. The intuition behind VerdictDB is that the adoption of AQP is hindered by the fact that each of the available AQP engines are tied to specific platforms and require users to completely abandon their existing databases. Thus, VerdictDB operates at the client-level, intercepts analytical queries issued to the database and rewrites them into another query that, if executed by any standard relational engine, will yield sufficient information for computing an approximate answer.

Quickr [KSV+16] approximates complex ad-hoc queries in big-data clusters by injecting samplers on-the-fly. BigData queries may need several passes over the data. The idea is that when samplers are injected at the appropriate location in the query plan, there can be substantial I/O improvements.

In Section 6.1.1, where we discussed the pros and cons of sampling techniques, we saw that one of the drawbacks of sampling is its poor performance in highly selective queries. Sample+Seek [DHC+16] is a sampling-based system that employs indices especially designed to handle queries of high selectivity.

The DBO [JAPD08] and XDB [LWYZ16] systems are two AQP engines based on *Online Aggregation* [HHW97]. Apart from queries over a single table, both these systems can also efficiently approximate multi-way JOIN queries. DBO's join algorithm is based on ripple join [HH99], while the work of [LWYZ16] formulates the problem as random walks over a graph.

ApproxHadoop [GBNN15] and Sapprox [ZWY16] are two Hadoop-based systems that use multi-stage sampling theory [Loh19] and extreme value theory [CBTD01] in order to speed up arbitrary user defined functions of MapReduce programs.

Traditionally, In sampling-based techniques, in order to quantify the error there are two main approaches: (i) analytic error quantification and (ii) the bootstrap method. The first approach is extremely efficient but lacks generality, whereas the second is quite general but suffers from its high computational overhead. ABS [ZGG+14] is a system that bridges the gap between the two through the *analytical bootstrap* method [ZGMZ14].

CHAPTER 7

# Conclusions

In this thesis, I have examined the role of wavelets in modern AQP. In the first part of the dissertation, batch processing scenarios are discussed and emphasis is given on the wavelet thresholding problem when non-Euclidean errors are optimized. Traditionally, the optimal solution of the problem comprises a DP-based approach. Having established that DP techniques do not scale for big datasets when executed over a single machine, we opt for designing algorithms with linear scalability over scale-out infrastructures. I first present a novel technique that allows the parallel execution of all the existing DP algorithms for the problem and show that it works for both one- and multi-dimensional datasets. The results indicate that we can scale DP algorithms to data sizes that their centralized counterparts are incapable of processing. Moreover, in order to further improve on the running-time for the synopsis construction, distributed heuristic algorithms are proposed. These greedy algorithms are more time-efficient than the state-of-the-art DP solutions and it is also shown that the performance gain they offer increases along with the dimensionality.

The advent of IoT has put increasing emphasis on edge computing. Scaling to the edge of the network relieves stress on centralized data stores and allows organizations to perform more effective analysis of data by shortening the time between taking in the data and acting on it. Furthermore, as for most applications there is more value in real-time information, recent data tend to be prioritized. In order to enable real-time processing, I also investigate the problem of constructing wavelet synopses under the sliding-window streaming model. Specifically, an approximate algorithm for answering range queries such as COUNT, SUM and AVG is proposed.

The experimental evaluation shows that the proposed approach outperforms many well-known techniques for a variety of data distributions and query workloads. Moreover, accuracy is further improved in case the workload is known. In order to achieve that, this thesis proposes a workload-aware system that trade-offs accuracy with synopsis construction time. The results indicate that near exact results can be obtained regardless of data distribution, while the construction throughput penalty is minimal.

# Bibliography


[AFTU97] Laurent Amsaleg, Michael J Franklin, Anthony Tomasic, and Tolga Urhan. Improving responsiveness for wide-area data access. In *IEEE Data Engineering Bulletin*. Citeseer, 1997.

[AGPR99a] Swarup Acharya, Phillip B Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. The aqua approximate query answering system. In *ACM Sigmod Record*, volume 28, pages 574–576. ACM, 1999.

[AGPR99b] Swarup Acharya, Phillip B Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. Join synopses for approximate query answering. In *ACM SIGMOD Record*, volume 28, pages 275–286. ACM, 1999.

[alg] Abstract algebra for scala. https://twitter.github.io/algebird/.

[amaa] Amazon ec2 elastic gpus. https://aws.amazon.com/ec2/elastic-graphics/.

[amab] Amazon ec2 f1 instances. https://aws.amazon.com/ec2/instance-types/f1/.

[AMP+13] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 29–42. ACM, 2013.

[AMS96] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 20–29. ACM, 1996.

[arda] Arduino memory. https://www.arduino.cc/en/tutorial/memory.

[ardb] Arduino sd library. https://www.arduino.cc/en/reference/SD.

[BDM02] Brian Babcock, Mayur Datar, and Rajeev Motwani. Sampling from a moving window over streaming data. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 633–634. Society for Industrial and Applied Mathematics, 2002.

[BDMO03] Brain Babcock, Mayur Datar, Rajeev Motwani, and Liadan O'Callaghan. Maintaining variance and k-medians over data stream windows. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 234–243. ACM, 2003.

[big] The definition of big data. https://www.oracle.com/big-data/guide/what-is-big-data.html.

[Blo70] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[BLS+08] Francesco Buccafurri, Gianluca Lax, Domenico Saccà, Luigi Pontieri, and Domenico Rosaci. Enhancing histograms by tree-like bucket indices. *The VLDB Journal*, 17(5):1041–1061, 2008.

[BSS07] Chiranjeeb Buragohain, Nisheeth Shrivastava, and Subhash Suri. Space efficient streaming algorithms for the maximum error histogram. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 1026–1035. IEEE, 2007.

[BT07] Costas Busch and Srikanta Tirthapura. A deterministic algorithm for summarizing asynchronous streams over a sliding window. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 465–476. Springer, 2007.

[CBTD01] Stuart Coles, Joanna Bawa, Lesley Trenner, and Pat Dorazio. *An introduction to statistical modeling of extreme values*, volume 208. Springer, 2001.

[CCA+10] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Nsdi*, volume 10, page 20, 2010.

[CÇC+02] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams: a new class of data management applications. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 215–226. VLDB Endowment, 2002.

[CD97] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and olap technology. *ACM Sigmod record*, 26(1):65–74, 1997.

[CDN07] Surajit Chaudhuri, Gautam Das, and Vivek Narasayya. Optimized stratified sampling for approximate query processing. *ACM Transactions on Database Systems (TODS)*, 32(2):9, 2007.

[CFPR00] Corinna Cortes, Kathleen Fisher, Daryl Pregibon, and Anne Rogers. Hancock: a language for extracting signatures from data streams. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 9–17. ACM, 2000.

[CGHJ12] Graham Cormode, Minos Garofalakis, Peter J Haas, and Chris Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1–3):1–294, 2012.

[CGRS01] Kaushik Chakrabarti, Minos Garofalakis, Rajeev Rastogi, and Kyuseok Shim. Approximate query processing using wavelets. *The VLDB Journal–The International Journal on Very Large Data Bases*, 10(2-3):199–223, 2001.

[CGS06] Graham Cormode, Minos Garofalakis, and Dimitris Sacharidis. Fast approximate wavelet tracking on streams. In *Advances in Database Technology-EDBT 2006*, pages 4–22. Springer, 2006.

[Cha98] Don Chamberlin. *A complete guide to DB2 universal database*. Morgan Kaufmann, 1998.

[CLLT12] Ho-Leung Chan, Tak-Wah Lam, Lap-Kei Lee, and Hing-Fung Ting. Continuous monitoring of distributed data streams over a time-based sliding window. *Algorithmica*, 62(3-4):1088–1111, 2012.

[CM05] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[CS03] Edith Cohen and Martin Strauss. Maintaining time-decaying stream aggregates. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 223–233. ACM, 2003.

[CW04] Pei Cao and Zhe Wang. Efficient top-k query calculation in distributed networks. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 206–215. ACM, 2004.

[Dau92] Ingrid Daubechies. *Ten lectures on wavelets*, volume 61. Siam, 1992.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[DGIM02] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *SIAM journal on computing*, 31(6):1794–1813, 2002.

[DGR01] Amol Deshpande, Minos Garofalakis, and Rajeev Rastogi. Independence is good: Dependency-based histogram synopses for high-dimensional data. *ACM SIGMOD Record*, 30(2):199–210, 2001.

[DHC+16] Bolin Ding, Silu Huang, Surajit Chaudhuri, Kaushik Chakrabarti, and Chi Wang. Sample+ seek: Approximating aggregates with distribution precision guarantee. In *Proceedings of the 2016 International Conference on Management of Data*, pages 679–694. ACM, 2016.

[fbr] Scaling the facebook data warehouse to 300 pb. https://code.facebook.com/posts/229861827208629/scaling-the-facebook-data-warehouse-to-300-pb/.

[FFGM07] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Discrete Mathematics and Theoretical Computer Science*, pages 137–156. Discrete Mathematics and Theoretical Computer Science, 2007.

[fli] Approximate calculation of frequencies in data streams. https://issues.apache.org/jira/browse/FLINK-2147.

[FM85] Philippe Flajolet and G Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences*, 31(2):182–209, 1985.

[GBNN15] Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D Nguyen. Approxhadoop: Bringing approximations to mapreduce frameworks. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 383–397. ACM, 2015.

[GG02] Minos Garofalakis and Phillip B Gibbons. Wavelet synopses with error guarantees. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 476–487. ACM, 2002.

[GGI+02] Anna C Gilbert, Sudipto Guha, Piotr Indyk, Yannis Kotidis, Sivaramakrishnan Muthukrishnan, and Martin J Strauss. Fast, small-space algorithms for approximate histogram maintenance. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 389–398. ACM, 2002.

[GGRS07] Sumit Ganguly, Minos Garofalakis, Rajeev Rastogi, and Krishan Sabnani. Streaming algorithms for robust, real-time detection of ddos attacks. In *Distributed Computing Systems, 2007. ICDCS'07. 27th International Conference on*, pages 4–4. IEEE, 2007.

[GH05] Sudipto Guha and Boulos Harb. Wavelet synopsis for data streams: minimizing non-euclidean error. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 88–97. ACM, 2005.

[GH08] Sudipto Guha and Boulos Harb. Approximation algorithms for wavelet transform coding of data streams. *IEEE Transactions on Information Theory*, 54(2):811–830, 2008.

[GK04] Minos Garofalakis and Amit Kumar. Deterministic wavelet thresholding for maximum-error metrics. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 166–176. ACM, 2004.

[GK05] Minos Garofalakis and Amit Kumar. Wavelet synopses for general error metrics. *ACM Transactions on Database Systems (TODS)*, 30(4):888–928, 2005.

[GKMS01] Anna C Gilbert, Yannis Kotidis, S Muthukrishnan, and Martin Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *VLDB*, volume 1, pages 79–88, 2001.

[GKMS03] Anna C Gilbert, Yannis Kotidis, S Muthukrishnan, and Martin J Strauss. One-pass wavelet decompositions of data streams. *Knowledge and Data Engineering, IEEE Transactions on*, 15(3):541–554, 2003.

[GKMS07] Anna C Gilbert, Ioannis Kotidis, Shanmugavelayutham Muthukrishnan, and Martin J Strauss. Method and apparatus for using wavelets to produce data summaries, November 13 2007. US Patent 7,296,014.

[GM98] Phillip B Gibbons and Yossi Matias. New sampling-based summary statistics for improving approximate query answers. In *ACM SIGMOD Record*, volume 27, pages 331–342. ACM, 1998.

[GMP97] Phillip B Gibbons, Yossi Matias, and Viswanath Poosala. Fast incremental maintenance of approximate histograms. In *VLDB*, volume 97, pages 466–475, 1997.

[GMP02] Phillip B Gibbons, Yossi Matias, and Viswanath Poosala. Fast incremental maintenance of approximate histograms. *ACM Transactions on Database Systems (TODS)*, 27(3):261–298, 2002.

[GPS08] Sudipto Guha, Hyoungmin Park, and Kyuseok Shim. Wavelet synopsis for hierarchical range queries with workloads. *The VLDB Journal—The International Journal on Very Large Data Bases*, 17(5):1079–1099, 2008.

[GS16] Max Grossman and Vivek Sarkar. Swat: A programmable, in-memory, distributed, high-performance computing platform. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 81–92. ACM, 2016.

[GT02] Phillip B Gibbons and Srikanta Tirthapura. Distributed streams algorithms for sliding windows. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 63–72. ACM, 2002.

[Guh05] Sudipto Guha. Space efficiency in synopsis construction algorithms. In *Proceedings of the 31st international conference on Very large data bases*, pages 409–420. VLDB Endowment, 2005.

[HH99] Peter J Haas and Joseph M Hellerstein. Ripple joins for online aggregation. *ACM SIGMOD Record*, 28(2):287–298, 1999.

[HHW97] Joseph M Hellerstein, Peter J Haas, and Helen J Wang. Online aggregation. In *Acm Sigmod Record*, volume 26, pages 171–182. ACM, 1997.

[HILM09] Peter J Haas, Ihab F Ilyas, Guy M Lohman, and Volker Markl. Discovering and exploiting statistical properties for query optimization in relational databases: A survey. *Statistical Analysis and Data Mining: The ASA Data Science Journal*, 1(4):223–250, 2009.

[Ioa03] Yannis Ioannidis. The history of histograms (abridged). In *Proceedings 2003 VLDB Conference*, pages 19–30. Elsevier, 2003.

[IP95] Yannis E Ioannidis and Viswanath Poosala. Balancing histogram optimality and practicality for query result size estimation. In *Acm Sigmod Record*, volume 24, pages 233–244. ACM, 1995.

[IP99] Yannis E Ioannidis and Viswanath Poosala. Histogram-based approximation of set-valued query-answers. In *VLDB*, volume 99, pages 174–185, 1999.

[JAPD08] Chris Jermaine, Subramanian Arumugam, Abhijit Pol, and Alin Dobra. Scalable approximate query processing with the dbo engine. *ACM Transactions on Database Systems (TODS)*, 33(4):23, 2008.

[JKM+98] Hosagrahar Visvesvaraya Jagadish, Nick Koudas, S Muthukrishnan, Viswanath Poosala, Kenneth C Sevcik, and Torsten Suel. Optimal histograms with quality guarantees. In *VLDB*, volume 98, pages 275–286, 1998.

[JYL11] Jeffrey Jestes, Ke Yi, and Feifei Li. Building wavelet histograms on large data in mapreduce. *Proceedings of the VLDB Endowment*, 5(2):109–120, 2011.

[KM05] Panagiotis Karras and Nikos Mamoulis. One-pass wavelet synopses for maximum-error metrics. In *Proceedings of the 31st international conference on Very large data bases*, pages 421–432. VLDB Endowment, 2005.

[KM07] Panagiotis Karras and Nikos Mamoulis. The haar+ tree: a refined synopsis data structure. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 436–445. IEEE, 2007.

[KM08] Panagiotis Karras and Nikos Mamoulis. Hierarchical synopses with optimal error guarantees. *ACM Transactions on Database Systems (TODS)*, 33(3):18, 2008.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[KSM07] Panagiotis Karras, Dimitris Sacharidis, and Nikos Mamoulis. Exploiting duality in summarization with deterministic guarantees. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 380–389. ACM, 2007.

[KSV+16] Srikanth Kandula, Anil Shanbhag, Aleksandar Vitorovic, Matthaios Olma, Robert Grandl, Surajit Chaudhuri, and Bolin Ding. Quickr: Lazily approximating complex adhoc queries in bigdata clusters. In *Proceedings of the 2016 international conference on management of data*, pages 631–646. ACM, 2016.

[lev] Leveldb. https://github.com/google/leveldb.

[LHZ+16] Xiaoyun Li, Shizhong Huang, Huanyu Zhao, Xueyan Guo, Libo Xu, Xingsen Li, and Youjia Li. Image compression based on restcted wavelet synopses with maximum error bound. In *Proceedings of the 9th International Conference on Utility and Cloud Computing*, UCC '16, pages 333–338, New York, NY, USA, 2016. ACM.

[lin] Linked sensor data. https://wiki.knoesis.org/index.php/SSW_Datasets.

[LLZO02] Tao Li, Qi Li, Shenghuo Zhu, and Mitsunori Ogihara. A survey on wavelet applications in data mining. *ACM SIGKDD Explorations Newsletter*, 4(2):49–68, 2002.

[Loh19] Sharon L Lohr. *Sampling: Design and Analysis: Design and Analysis*. Chapman and Hall/CRC, 2019.

[LTC10] Ken-Hao Liu, Wei-Guang Teng, and Ming-Syan Chen. Dynamic wavelet synopses management over sliding windows in sensor networks. *IEEE Transactions on Knowledge and Data Engineering*, 22(2):193–206, 2010.

[LWYZ16] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. Wander join: Online aggregation via random walks. In *Proceedings of the 2016 International Conference on Management of Data*, pages 615–629. ACM, 2016.

[Mal99] Stéphane Mallat. *A wavelet tour of signal processing*. Elsevier, 1999.

[mar] Histogram-based statistics in mariadb. https://mariadb.com/kb/en/library/histogram-based-statistics/.

[MF02] Samuel Madden and Michael J Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 555–566. IEEE, 2002.

[Moz15] Barzan Mozafari. Verdict: A system for stochastic query planning. In *CIDR*, 2015.

[MP03] Yossi Matias and Leon Portman. Workload-based wavelet synopses. Technical report, Technical report, Department of Computer Science, Tel Aviv University, 2003.

[MP04] Yossi Matias and Leon Portman. $\tau$-synopses: a system for run-time management of remote synopses. In *International Conference on Extending Database Technology*, pages 865–867. Springer, 2004.

[mss] Sql server statistics. https://docs.microsoft.com/en-us/sql/relational-databases/statistics/statistics?view=sql-server-ver15.

[Mut05] S Muthukrishnan. Subquadratic algorithms for workload-aware haar wavelet synopses. In *FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science*, pages 285–296. Springer, 2005.

[MVW98] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. Wavelet-based histograms for selectivity estimation. In *ACM SIGMOD Record*, volume 27, pages 448–459. ACM, 1998.

[noa] National oceanic and atmospheric administration. https://www1.ncdc.noaa.gov/pub/data/noaa/.

[nyc] Nyct. http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml.

[ope] Opencl standard. https://www.khronos.org/opencl/.

[ora] Oracle 12c histograms. https://docs.oracle.com/database/121/TGSQL/tgsql_histo.htm.

[par] Parquet. https://parquet.apache.org/.

[PGD12] Odysseas Papapetrou, Minos Garofalakis, and Antonios Deligiannakis. Sketch-based querying of distributed sliding-window data streams. *Proceedings of the VLDB Endowment*, 5(10):992–1003, 2012.

[PMSW18] Yongjoo Park, Barzan Mozafari, Joseph Sorenson, and Junhao Wang. Verdictdb: universalizing approximate query processing. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1461–1476. ACM, 2018.

[pre] Presto: Distributed sql query engine for big data. https://prestodb.github.io/.

[PSC84] Gregory Piatetsky-Shapiro and Charles Connell. Accurate estimation of the number of tuples satisfying a condition. *ACM Sigmod Record*, 14(2):256–276, 1984.

[PZHM09] Chaoyi Pang, Qing Zhang, David Hansen, and Anthony Maeder. Unrestricted wavelet synopses under maximum error bound. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 732–743. ACM, 2009.

[QAEA03] Lin Qiao, Divyakant Agrawal, and Amr El Abbadi. Supporting sliding window queries for continuous data streams. In *Scientific and Statistical Database Management, 2003. 15th International Conference on*, pages 85–94. IEEE, 2003.

[RBM15] Nicoló Rivetti, Yann Busnel, and Achour Mostefaoui. *Efficiently Summarizing Distributed Data Streams over Sliding Windows*. PhD thesis, LINA-University of Nantes; Centre de Recherche en Économie et Statistique; Inria Rennes Bretagne Atlantique, 2015.

[RMW+16] Jags Ramnarayan, Barzan Mozafari, Sumedh Wale, Sudhir Menon, Neeraj Kumar, Hemant Bhanawat, Soubhik Chakraborty, Yogesh Mahajan, Rishitesh Mishra, and Kishor Bachhav. Snappydata: A hybrid transactional analytical store built on spark. In *Proceedings of the 2016 International Conference on Management of Data*, pages 2153–2156. ACM, 2016.

[SÇZ05] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *ACM Sigmod Record*, 34(4):42–47, 2005.

[SDS96] Eric J Stollnitz, Tony D DeRose, and David H Salesin. *Wavelets for computer graphics: theory and applications*. Morgan Kaufmann, 1996.

[Shn84] Ben Shneiderman. Response time and display rate in human performance with computers. *ACM Computing Surveys (CSUR)*, 16(3):265–285, 1984.

[SJBK08] Cyrus Shahabi, Mehrdad Jahangiri, and Farnoush Banaei-Kashani. Proda: An end-to-end wavelet-based olap system for massive datasets. *Computer*, 41(4):69–77, 2008.

[SMTZ17] Zubair Shah, Abdun Naser Mahmood, Zahir Tari, and Albert Y Zomaya. A technique for efficient query estimation over distributed data streams. *IEEE Transactions on Parallel & Distributed Systems*, (10):2770–2783, 2017.

[spa] Spark SQL. https://spark.apache.org/sql/.

[SZB+16] Hong Su, Mohamed Zait, Vladimir Barrière, Joseph Torres, and Andre Menck. Approximate aggregates in oracle 12c. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 1603–1612. ACM, 2016.

[TK15] Immanuel Trummer and Christoph Koch. An incremental anytime algorithm for multi-objective query optimization. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1941–1953. ACM, 2015.

[VW99] Jeffrey Scott Vitter and Min Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *ACM SIGMOD Record*, volume 28, 1999.

[XTB08] Bojian Xu, Srikanta Tirthapura, and Costas Busch. Sketching asynchronous data streams over sliding windows. *Distributed Computing*, 20(5):359–374, 2008.

[yah] Fast, approximate analysis of big data (yahoo's druid). https://yahooeng.tumblr.com/post/135390948446/data-sketches.

[YG+03] Yong Yao, Johannes Gehrke, et al. Query processing in sensor networks. In *Cidr*, pages 233–244, 2003.

[ZGG+14] Kai Zeng, Shi Gao, Jiaqi Gu, Barzan Mozafari, and Carlo Zaniolo. Abs: a system for scalable approximate queries with accuracy guarantees. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1067–1070. ACM, 2014.

[ZGMZ14] Kai Zeng, Shi Gao, Barzan Mozafari, and Carlo Zaniolo. The analytical bootstrap: a new method for fast error estimation in approximate query processing. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 277–288. ACM, 2014.

[ZPH09] Qing Zhang, Chaoyi Pang, and David Hansen. On multidimensional wavelet synopses for maximum error bounds. In *International Conference on Database Systems for Advanced Applications*, pages 646–661. Springer, 2009.

[ZS02] Yunyue Zhu and Dennis Shasha. Statstream: Statistical monitoring of thousands of data streams in real time** work supported in part by us nsf grants iis-9988345 and n2010: 0115586. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pages 358–369. Elsevier, 2002.

[ZWY16] Xuhong Zhang, Jun Wang, and Jiangling Yin. Sapprox: enabling efficient and accurate approximations on sub-datasets with distribution-aware online sampling. *Proceedings of the VLDB Endowment*, 10(3):109–120, 2016.

APPENDIX A

# The MinHaarSpace Algorithm

MinHaarSpace [KSM07] is a DP algorithm that targets the error-bound Problem 2 that was described in Section 2.4. The coefficients that MinHaarSpace selects for the synopsis are not strictly in the set of the ones produced by the HWT but can be any value $z \in \mathbb{R}$. However, as we discussed in Chapter 2 for unrestricted wavelets, we do not have to explore all real numbers but we can restrict coefficient values in a bounded range.

For bounding the search space for candidate coefficient values, MinHaarSpace uses the notion of the *incoming value*. An incoming value at node $c_i$ of the Haar tree is a value reconstructed in the path of ancestors from the root node up to $c_i$ in the sparse representation $\hat{Z}$ of $A$. In a wavelet decomposition $W(A)$, this is the average value in the interval $I$ under the scope of $c_i$, henceforward called *real incoming value* at $c_i$. With that in mind, the algorithm first bounds the candidate incoming values for a node (Lemma 16) and then uses this bound to define the search space for coefficient values (Lemma 17). For making the exploration of both incoming values at $c_i$ and assigned values in the coefficients feasible, the real-valued domains of $v$ and $z_i^v$ are quantized into multiples of a small resolution step $\delta$.

**Lemma 16.** *Let $v_i$ be the real incoming value at node $c_i$. Let $v$ be an incoming value to $c_i$ for which the error bound $\epsilon$ under the $L_\infty$ can be satisfied, and $\bar{\epsilon} = \frac{\epsilon}{min_{j \in I}}\{|w_j|\}$, where $I$ is the interval under the scope of node $c_i$; then $|v_i - v| \leq \bar{\epsilon}$.*

**Lemma 17.** *Let $v_i$ be the real incoming value at node $c_i$, $z_i$ the real assigned value at $c_i$, $v \in S_i$ be a possible incoming value at $c_i$ for which the maximum error bound $\epsilon$ can be satisfied, and $z_i^v$ be a value that can be assigned at $c_i$ for incoming value $v$, satisfying $\epsilon$; then $|z_i - z_i^v| \leq \bar{\epsilon} - |v_i - v|$.*

Lemma 17 implies that the finite set $S_i^v \subset \mathbb{R}$ of possible assigned values we have to examine at node $c_i$, for a given incoming value $v \in S_i$, consists of the multiples of $\delta$ in the interval $[z_i - (\bar{\epsilon} - |v_i - v|), z_i + (\bar{\epsilon} - |v_i - v|)]$; hence, $|S_i^v| \leq \lfloor \frac{2(\bar{\epsilon} - |v_i - v|)}{\delta} \rfloor + 1 = O\left(\frac{\epsilon}{\delta}\right)$.

Based on that Lemma, MinHaarSpace comprises a bottom-up, left-to-right procedure over the error-tree. At each visited node $c_i$ it calculates an array $D$ of size $|S_i|$ from the pre-calculated arrays $L$ and $R$ of its children nodes $c_{i_L}$, $c_{i_R}$ (a single array $C$ for the child $i_C$ of the root node). $D$ holds an entry $D[v]$ for each possible incoming value $v$ at $c_i$. Such an entry contains: (i) the minimum number $D[v].s = S(i, v)$ of non-zero coefficients that need to be retained in the sub-tree rooted at $c_i$ with incoming value $v$, so that the resulting synopsis satisfies the error bound $\epsilon$; (ii) the $\delta$-optimal value $D[v].z$ to assign at $c_i$, for incoming value $v$; and (iii) the actual minimized maximum error $D[v].e$ that is obtained in the scope of $c_i$. Then, a DP procedure is formulated and recursively expressed as:

$$S(i, v) = \min_{z \in S_i^v} \{S(i_L, v + z) + S(i_R, v - z) + (z \neq 0)\}$$

$$S(0, 0) = \min_{z \in S_0^0} \{S(i_C, z) + (z \neq 0)\}$$

These equations tabulate all possible space allocations. They compute: (i) the minimum required space if a non-zero coefficient value $z$ is assigned at node $c_i$ and (ii) the required space if the coefficient is discarded. The latter case only applies if $0 \in S_i^v$. Let $\bar{S_i^v} \subset \mathbb{R}$ denote the set of those assigned values at node $c_i$ for incoming value $v$ that require the minimum space in order to achieve the error bound $\epsilon$:

$$\bar{S_i^v} = argmin_{z \in S_i^v} \{S(i_L, v + z) + S(i_R, v - z) + (z \neq 0)\}$$

$$\bar{S_0^0} = argmin_{z \in S_0^0} \{S(i_C, z) + (z \neq 0)\}$$

The $\delta$-optimal value to select is the one among these candidates that also minimizes, in a secondary priority, the obtained $L_\infty$-error in the scope of $c_i$. Let $E(i, v)$ be the minimum $L_\infty$ error obtained in the scope of $c_i$ with incoming value $v$ and an assigned value $z$, with $S(i, v)$ coefficients retained in the sub-tree rooted at $c_i$:

$$E(i, v) = \min_{z \in \bar{S_i^v}} \{max\{E(i_L, v + z), E(i_R, v - z)\}\}$$

$$E(0,0) = \min_{z \in S_0^0} \{E(i_C, z)\}$$

Algorithm 15 presents the pseudocode of MinHaarSpace as a recursive procedure.

**Algorithm 15:** MinHaarSpace($i$,$\epsilon$)

```
input: Index i, error-bound ε, data vector A
if i = 0 then
    C = MinHaarSpace(1, ε);
    compute s, z ∈ S_0^0, e of D from C;
else if i < N/2 then
    L = MinHaarSpace(i_L, ε);
    R = MinHaarSpace(i_R, ε);
    for each v ∈ S_i do
        compute s, z ∈ S_i^v, e of D[v] from L, R;
else if i ≥ N/2 then
    for each v ∈ S_i do
        compute s ∈ {0, 1}, z ∈ {0, c_i}, e of D[v] from A;
return D;
```

**Complexity Analysis**. The result array $D$ on each node $c_i$ holds $|S_i|$ entries, one for each possible incoming value, hence its size is $O\left(\frac{\epsilon}{\delta}\right)$. Furthermore, at each node $c_i$ and for each $v \in S_i$, the loop through all $|S_i^v|$ possible assigned values takes $O\left(\frac{\epsilon}{\delta}\right)$ time. Hence, the runtime of MinHaarSpace(0, $\epsilon$) is $O\left(\left(\frac{\epsilon}{\delta}\right)^2 N\right)$. Besides, since at most $logN + 1$ arrays need to be concurrently stored, the space complexity is $O\left(\frac{\epsilon}{\delta} logN + N\right)$

APPENDIX B

# MapReduce for $L_2$-error Synopses

In this Appendix, there can be found a description of the Mapreduce algorithms presented in [JYL11] for the construction of the $L_2$-optimal wavelet synopsis. The described algorithms are: Send-Coef, Send-V and H-WTopk. All the algorithms in [JYL11] compute wavelet synopses over histograms. Thus, in order to compare them against the proposed algorithms of this thesis, the algorithms are first modified not to compute histograms but perform the wavelet transform directly on the input data. For all the descriptions that follow, we denote $N$ as the dataset size, $m$ the number of map tasks, $S$ the input size of a map task and $R$ the size of the root sub-tree in datapoints.

## B.1 Send-V

The simplest algorithm presented in [JYL11] for the computation of a conventional synopsis is Send-V. The Send-V algorithm computes a histogram in the map phase of the job. The reducer centrally computes the wavelet coefficients and retains the $B$ largest ones. As the histogram computation is not required in our case, Send-V is, in effect, a sequential algorithm, where the reducer reads and centrally computes the wavelet transform for all the input data.

## B.2 Send-Coef

For computing the wavelet coefficients, Send-Coef is based on the basis vectors method, as described in Section 2.3. The distributed computation of Send-Coef is based on the following observation:

$$w_i = \langle A, \phi_i \rangle = \sum_{j=1}^{m} \langle A_j, \phi_i \rangle,$$

where $A_j$ is the $j$-th partition of the initial input data. Thus, every wavelet coefficient is a linear combination of the data values that belong to its sub-tree in the error-tree.

Send-Coef partitions the data in a different way than the one proposed in Section 3.2. Each mapper takes up as many datapoints as they fit in a typical HDFS block size. The block size does not need to be aligned to a power of two. For every datapoint $d_i$, the mapper computes its contribution to the final value of every wavelet coefficient in $path_{d_i}$. Thus, a mapper partially computes all the coefficients along the path from its datapoints to the root of the error-tree and thus sub-tree locality is not preserved. The reducer computes the final coefficients by aggregating the partially computed values and then retains the B largest ones in absolute normalized value. Algorithm 16 gives the pseudocode for the mappers of the Send-Coef algorithm.

As data locality is not preserved and for every data value we need to compute its contribution to $logN+1$ nodes (the path to the root), the computational complexity of a mapper is $O\left(SlogN\right)$. Furthermore, every mapper emits $O\left(S\left(logN - logS\right)\right)$ key-values to the reducer. By having $m$ mappers, the whole communication cost is $O\left(mS\left(logN - logS\right)\right) = O\left(N\left(logN - logS\right)\right)$. Compared to Send-Coef, CON achieves better computational complexity by a factor of $logN$ and communication cost by $logN - logS$.

**Algorithm 16:** *Send-CoefMapper*

**Require:** $S$: mapper input data
1: **for all** datapoints $d_i \in S$ **do**
2:  **for all** error-tree nodes $j \in path_{d_i}$ **do**
3:  compute contribution $c_{i,j}$ of $d_i$ to coefficient $c_j$
4:  **if** $c_j$ is fully computed **then** emit $(j, c_j)$
5: **for all** datapoints $d_i \in S$ **do**
6:  **for all** error-tree nodes $j \in path_{d_i}$ **do**
7:  **if** $c_j$ is partially computed **then** emit $(j, c_{i,j})$

### B.2.1 H-WTopk

In order to reduce the communication cost between the map and the reduce phase, the H-WTopk algorithm is proposed in [JYL11]. H-WTopk is based on the TPUT [CW04] algorithm for the distributed top-k problem. In contrast to TPUT, H-WTopk can handle both positive and negative values, as both are possible for a wavelet coefficient. The intuition behind the algorithm is to use a partial sum to prune items that cannot be in the top-k. Thus, a mapper does not need to send all of its data to the reducer but only a set of candidate nodes, according to the local partial sums. The algorithm requires three communication rounds between the mappers and the reducer. For a coefficient $x$, $c(x)$ denotes its value and $c_j(x)$ its partially computed value at mapper j.

**Round 1**: Each mapper first emits the coefficients with the k highest and k lowest (i.e., most negative) values. For each coefficient $x$ seen at the reducer, a lower bound $\tau(x)$ is computed on its total value's magnitude $|c(x)|$ (i.e., $|c(x)| \geq \tau(x)$), as follows. First, an upper bound $\tau^+(x)$ and a lower bound $\tau^-(x)$ are computed on its total value $c(x)$ (i.e., $\tau^-(x) \leq c(x) \leq \tau^+(x)$): If a mapper sends out the value of $x$, its exact value is added. Otherwise, for $\tau^+(x)$, the k-th highest value this mapper sends out is added and for $\tau^-(x)$ the k-th lowest value is added. Then we set $\tau(x) = 0$ if $\tau^+(x)$ and $\tau^-(x)$ have different signs and
$\tau(x) = min\{|\tau^+(x)|, |\tau^-(x)|\}$ otherwise. Doing so ensures $\tau^-(x) \leq c(x) \leq \tau^+(x)$ and $|c(x)| \geq \tau(x)$. Now, the k-th largest $\tau(x)$, denoted as $T_1$, is used as a threshold for the magnitude of the top-k coefficients.

**Round 2**: A mapper $j$ next emits all local coefficients $x$ having $|c_j(x)| > T_1/m$. This ensures a coefficient in the true top-k in magnitude must be sent by at least one mapper after this round, because if a coefficient is not sent, its aggregated value's magnitude can be no higher than $T_1$.

Now, with more values available from each mapper, upper and lower bounds $\tau^+(x), \tau^-(x)$ are refined for each coefficient $x \in L$, where $L$ is the set of coefficients ever received. If a mapper did not send the value for some $x$, $T_1/m$ (– $-T_1/m$) is now used for computing $\tau^+(x)$ ($\tau^-(x)$). This produces a new better threshold, $T_2$ (calculated in the same way as computing $T_1$ with improved $\tau(x)$'s), on the top-k coefficients' magnitude.

Next, coefficients are further pruned from $L$. For any $x \in L$ a new threshold $\tau'(x) = max\{|\tau^+(x)|, |\tau^-(x)|\}$ is computed based on refined upper and lower bounds $\tau^+(x), \tau^-(x)$. If $\tau'(x) < T_2$, coefficient $x$ is deleted from $L$. The final top-k coefficients must be in the set $L$.

**Round 3**: Finally, the values of all coefficients in $L$ are requested from each mapper. Then the aggregated values of exactly these coefficients are computed, and the k of largest magnitude among them are selected as the synopsis.

# Index of Algorithms

# Acronyms

**AQP** Approximate Query Processing. xi, xxi, 11, 13–15, 93, 105–107, 109, 111, 113–115

**DP** Dynamic Programming. ix, x, xiii, 15, 17, 18, 26, 29, 31–39, 51, 52, 56, 68, 87, 107, 110, 111, 115, 129, 130

**GPU** Graphics Processing Unit. 16, 18, 35–37, 55–57

**HWT** Haar Wavelet Transform. 26–28, 30, 129

**IoT** Internet of Things. xxi, xxii, 17, 71, 88, 115