Thèse n° 9017

Interactive-time Exploration, Querying, and Analysis of Large High-dimensional Datasets

Présentée le 5 septembre 2023

Faculté informatique et communications
Laboratoire de théorie et applications d’analyse de données
Programme doctoral en informatique et communications

pour l’obtention du grade de Docteur ès Sciences

par

Sachin BASIL JOHN

Acceptée sur proposition du jury

Prof. B. Falsafi, président du jury
Prof. C. Koch, directeur de thèse
Prof. D. Suciu, rapporteur
Dr M. Nikolic, rapporteur
Prof. K. Aberer, rapporteur

École
polytechnique
fédérale
de Lausanne

2023

The important thing is not to stop questioning.
Curiosity has its own reason for existence.
— Albert Einstein

To my family…

Acknowledgements

I would like to express my deepest gratitude to my advisor, Prof. Christoph Koch, for his continuous support throughout my doctoral study. His wisdom, patience, and dedication to ensuring my academic progress are incomparable. His guidance helped me navigate through the toughest times of my research journey. The lessons I've learned from him have shaped my scholarly development and will undoubtedly serve me throughout my future career.

I would also like to thank the rest of my thesis committee: Prof. Babak Falsafi, Prof. Karl Aberer, Prof. Dan Suciu, and Dr. Milos Nikolic for their insightful comments, thought-provoking questions, and encouragement.

My sincere appreciation goes to my colleagues and fellow researchers in DATA lab at EPFL, Peter and Zilu, for their helpful discussions, shared knowledge, and academic camaraderie. The stimulating environment of our lab has been a driving force behind my motivation to explore and create. I am also deeply grateful to my former colleagues Mohammad, Amir and Daniel for their guidance whenever I needed it.

I am deeply grateful to the administrative assistants of our lab, Simone and Catherine, for their valuable support, without which this journey would have been much more difficult.

I am deeply thankful to my friends for their endless cheerleading and understanding. I would like to particularly thank Aswin and Tom, who embarked on their respective PhD journeys around the same time I did, and with whom I shared several moments of joy and stress together. All of you have made this journey less solitary and far more enjoyable.

The love, encouragement, and support of my wife, Anupa, has been my beacon of hope ever since she became part of my life. Her belief in my abilities, even in times of doubt, provided the strength I needed to continue. Her sacrifices and understanding have made this achievement possible. Her unyielding love has been my fortress and my comfort.

Lastly, and most importantly, I am forever indebted to my other family members. My parents, Laju and Surekha, brother Sharath, and grandparents have showered me with unconditional love, always believed in my abilities, and constantly reminded me of what truly matters in life. Your support has been my stronghold from the moment I was born. This achievement is as much yours as it is mine.

In the grand tapestry of life, this doctoral journey is but one thread, yet it has been deeply impactful. To everyone who has touched this part of my life, thank you.

Lausanne, 29 May 2023 S.B.J.

Abstract

In the current era of big data, aggregation queries on high-dimensional datasets are frequently utilized to uncover hidden patterns, trends, and correlations critical for effective business decision-making. Data cubes facilitate such queries by employing pre-computation, but traditional data cube techniques struggle when managing hundreds of dimensions due to exponential increases in storage and time requirements for the pre-computation.

This thesis presents Sudokube, an innovative data cube system, designed to facilitate efficient querying on high-dimensional data. Sudokube introduces an approach that supports high-dimensional data cubes with interactive query speeds and moderate storage costs. It is based on judiciously partially materialized binary-domain data cubes, and quickly reconstructing missing cuboids using statistical or linear programming techniques.

Detailing Sudokube's functionality, this thesis explores the processes of data loading, cuboid selection for materialization, and efficient storage formats for optimizing space and projection time. It investigates the solvers used to reconstruct non-materialized cuboids, offering an in-depth comparison concerning speed, accuracy, and resource requirements. It also elaborates on Sudokube's supported queries and aggregation functions, underpinned by extensive experiments on real-world and synthetic datasets to demonstrate Sudokube's capabilities.

In conclusion, this thesis provides a comprehensive examination of Sudokube, positing it as an effective solution to the inherent complexities of high-dimensional data exploration. The research signifies a substantial advancement in the high-dimensional data domain, empowering users to undertake exploratory data analysis for feature engineering, eliminating the necessity for compromise while loading data into a data cube, and enhancing the performance of queries with hierarchical dimensions. The insights from this work underline Sudokube's potential to foster advancements in data science methodologies and to open up new avenues in the field of big data analysis.

Key words: data cubes, approximate query processing, online aggregation, online analytical processing, data exploration, data visualization, linear programming, moments, iterative proportional fitting, view materialization.

Résumé

Dans l'ère actuelle des mégadonnées, les requêtes d'agrégation sur des jeux de données à haute dimension sont fréquemment utilisées pour découvrir des modèles, des tendances et des corrélations cachées essentielles pour une prise de décision commerciale efficace. Les cubes de données facilitent ces requêtes en employant une pré-computation, mais les techniques traditionnelles de cubes de données rencontrent des difficultés lors de la gestion de centaines de dimensions en raison des augmentations exponentielles des exigences de stockage et de temps de pré-computation.

Cette thèse présente Sudokube, un système de cube de données innovant, conçu pour faciliter l'interrogation efficace sur des données à haute dimension. Sudokube propose une approche qui prend en charge les cubes de données à haute dimension avec des vitesses de requête interactives et des coûts de stockage modérés. Il est basé sur des cubes de données à domaine binaire judicieusement partiellement matérialisés et la reconstruction rapide des cuboïdes manquants en utilisant des techniques statistiques ou de programmation linéaire.

En détaillant la fonctionnalité de Sudokube, cette thèse explore les processus de chargement des données, la sélection des cuboïdes à matérialiser et les formats de stockage efficaces pour optimiser l'espace et le temps de projection. Elle étudie les solveurs utilisés pour reconstruire les cuboïdes non matérialisés, en offrant une comparaison approfondie en termes de vitesse, de précision et d'exigences en ressources. La thèse détaille les requêtes et les fonctions d'agrégation prises en charge par Sudokube, étayées par des expériences étendues sur des ensembles de données réels et synthétiques pour démontrer les capacités de Sudokube.

En conclusion, cette thèse fournit un examen complet de Sudokube, le présentant comme une solution efficace aux complexités inhérentes à l'exploration de données à haute dimension. La recherche représente une avancée substantielle dans le domaine des données à haute dimension, permettant aux utilisateurs d'entreprendre une analyse de données exploratoire pour l'ingénierie des caractéristiques, éliminant la nécessité de compromis lors du chargement des données dans un cube de données, et améliorant les performances des requêtes avec des dimensions hiérarchiques. Les perspectives de ce travail soulignent le potentiel de Sudokube pour favoriser les avancées dans les méthodologies de science des données et pour ouvrir de nouvelles voies dans le domaine de l'analyse des mégadonnées.

Mots-clés : cubes de données, traitement de requêtes approximatif, agrégation en ligne, traitement analytique en ligne, exploration de données, visualisation de données, programmation linéaire, moments, ajustement proportionnel itératif, matérialisation de vues.

Contents

Acknowledgements i

Abstract (English/Français) iii

List of figures ix

List of algorithms xiii

List of experiments xv

1 Introduction 1
- 1.1 Research Questions 2
- 1.2 Contributions 3
- 1.3 Thesis Outline 4

2 Background 5
- 2.1 Sets, Functions, and Vectors 5
- 2.2 Multivariate Bernoulli Distributions 7
- 2.3 Kronecker Product of Matrices 10
- 2.4 Data Cube 10

3 Motivation 15
- 3.1 Classical Implementation of Data Cubes 15
- 3.2 Advantages of High Dimensional Cubes 16
  - 3.2.1 Off-the-shelf OLAP 16
  - 3.2.2 Exploration of High-dimensional Data 17
  - 3.2.3 Avoiding Snowflake Schema 18
  - 3.2.4 More Powerful Queries 18
- 3.3 The Infeasibility of High-dimensional Data Cubes 19
- 3.4 Related Work 22
  - 3.4.1 Fully Materialized Data Cubes 22
  - 3.4.2 Partially Materialized Data Cubes 22
  - 3.4.3 Iceberg Cubes 23
  - 3.4.4 Compressed Cubes 23
  - 3.4.5 Inverted Indexes 24
  - 3.4.6 Sampling 24

3.4.7 Synopses 26

4 Sudokube System 27

4.1 Sudokube Ideas 27

4.1.1 Materialization and Querying 27

4.1.2 Binary Cuboids 29

4.2 System Architecture 31

4.2.1 Cube Specification and Querying 32

4.2.2 Frontend 33

4.2.3 Backend 34

4.2.4 Materialization Strategy 34

4.2.5 Query Approximation 35

4.3 Experimental Setup 36

4.3.1 Dataset Description 36

5 Data Loading and Querying 39

5.1 Data Loading 39

5.1.1 Static Schema 40

5.1.2 Dynamic Schema 41

5.2 Building the Data Cube 42

5.3 Querying 43

5.4 Finding Materialized Cuboids Relevant to Queries 44

5.5 Output Post-processing 46

6 Sudokube Solvers 49

6.1 Answering Queries Exactly 51

6.2 Solving Queries Using Linear Programming 55

6.3 Solving Queries Using Moments 75

6.3.1 Moments of a Cuboid 75

6.3.2 Cuboid Transformations 77

6.3.3 Cuboid Approximation from Projections 80

6.3.4 Improved Moment Extrapolation Algorithm 87

6.4 Iterative Proportional Fitting 94

6.4.1 The Base Algorithm 95

6.4.2 Convergence and Maximum Entropy 96

6.4.3 Graphical Models 96

6.4.4 Junction Tree IPF 97

6.4.5 Moment-based IPF 99

6.5 Comparing Different Solvers 106

6.6 Related Work 109

7 Materialization Strategy 111

7.1 Number of Materializable Cuboids 111

7.2 Score Functions 113
7.2.1 Score Function for Naive Solver 114
7.2.2 Score Function for Approximating from Projections 116

8 Sudokube Backend **123**
8.1 Cuboid Layout 123
8.1.1 Dense Format 123
8.1.2 Sparse Format 123
8.2 Projection 125
8.2.1 Dense and Sparse Row 127
8.2.2 Sparse Column 129
8.3 Alternative Storage Layout 133
8.3.1 Moment Store 134

9 Conclusion and Future Work **139**

Bibliography **148**

Curriculum Vitae **149**

List of Figures

2.1 An example of a 5-dimensional Bernoulli distribution 7
2.2 An example data cube for sales data 11
2.3 A snowflake schema for sales data with a central fact table and additional dimension tables for location, product, and time. 12
2.4 The lattice of cuboids for the sales data cube 13
2.5 An overview of the high-level operations that can be applied on a data cube . . 13

3.1 Simulation results for the density of a random d-dimensional projection of a n-dimensional cuboid. Each curve starts at $n = d$ for various d values between 6 and 23. 21
3.2 Relative size and density of a random k-dimensional projection of an n-dimensional cuboid with $m = 2$, $n = 64$, and $d = 20$ 21

4.1 Comparison of the classical idea of projecting smallest subsuming cuboid to answer query vs. Sudokube idea of approximating queries using all of its available projections 28
4.2 Encoding Item, Quarter and City keys using two binary dimensions each. 29
4.3 Encoding of sales data using binary dimensions in Sudokube 30
4.4 Architecture of the Sudokube system and the workflows for building (①-⑥) and querying (①-⑤) data cubes in the Sudokube system 31
4.5 The Sudokube user interface for querying. Users specify dimensions on the horizontal axis as well as for series and apply filters. They also choose the measure, the aggregation function, and the solver for answering queries. 33

5.1 An example DAG of encoders for the sales data schema. The root node is a product of its children, suggesting that queries are to be constructed by taking any combination of queries of its children. The intermediate nodes are sums of their respective children, suggesting that queries are constructed by taking the union of queries of their children. The leaf nodes represent the individual encoders for each column. 41

6.1 Lattice of cuboids containing binary dimensions 0...3 from the sales data cube 50
6.2 Values in the materialized cuboids for the example sales data cube 50
6.3 Fraction of queries answered by the naive solver by projecting cuboids of various dimensionality for different query dimensionality 52

6.4 Average time spent by the naive solver in every phase of query execution in batch mode for different query dimensionality 52
6.5 Fraction of queries answered by the naive solver by projecting cuboids of various dimensionality for different number of materialized cuboids 54
6.6 Average time spent by the naive solver in every phase of query execution in batch mode for different number of materialized cuboids 54
6.7 Fraction of queries answered by the naive solver by projecting cuboids of various dimensionality for different values of minimum dimensionality of materialized cuboids 56
6.8 Average time spent by the naive solver in every phase of query execution in batch mode for different values of minimum dimensionality of materialized cuboids 56
6.9 The 12 equations obtained from the three relevant materialized cuboids of the sales data cube for the query $Q = \{3, 1, 0\}$. Among these, at most 7 are linearly independent. 57
6.10 Average frequency count for cuboid dimensionality in Sudokube approach when the query dimensionality is varied. Histograms for both the original dimensionality of the materialized cuboid and the dimensionality after it is projected to a subset of the query dimensions are shown. 65
6.11 Average values for execution time, degrees of freedom, and error for the linear programming solver run in batch mode when the query dimensionality is varied 66
6.12 Average frequency count for cuboid dimensionality in Sudokube approach when the number of materialized cuboids is varied. Histograms for both the original dimensionality of the materialized cuboid and the dimensionality after it is projected to a subset of the query dimensions are shown. 69
6.13 Average values for execution time, degrees of freedom, and error for the linear programming solver run in batch mode when the number of materialized cuboids is varied 70
6.14 Average frequency count for cuboid dimensionality in Sudokube approach when the minimum dimensionality of materialized cuboids is varied. Histograms for both the original dimensionality of the materialized cuboid and the dimensionality after it is projected to a subset of the query dimensions are shown. 72
6.15 Average values for execution time, degrees of freedom, and error for the linear programming solver run in batch mode when the minimum dimensionality of materialized cuboids is varied 73
6.16 Improvement of error over time for linear programming solver in online mode for various query dimensionality 74
6.17 Data flow on array $\mathbf{A}$ in Algorithm 7, innermost loop, shown for $I = \{1, \dots, m\}$ and $\boldsymbol{M} = (M_{11}\ M_{12} M_{21}\ M_{22})$ 78
6.18 Computations involved in Algorithm 7 when applied to transform $\boldsymbol{\mu}_Q$ back to $\boldsymbol{C}_Q$ 79
6.19 Computing seven out of eight moments μ_Q of the query cuboid C_Q from its three projections. The last moment is not known and has to be extrapolated from the others. 81

6.20 Average error for different moment extrapolation techniques run in batch mode for various query dimensionality when no heuristics are applied to improve the error . . . 85
6.21 Average error for different moment extrapolation techniques run in batch mode for various query dimensionality after applying heuristic bounds to improve the error . . . 87
6.22 Approximating query result from projections using the improved moment solver. The improved moment solver extracts central moments directly from projections and approximates query results assuming unknown central moments to be zero. Instead of transforming extrapolated central moments directly to the query cuboid, it first transforms the central moments to raw moments and then transforms raw moments to the query cuboid while applying the heuristic bounds to improve the error. . . . 88
6.23 Average time spent by the original and the improved moment solvers in each phase of query execution in batch mode while varying the query dimensionality, the number of materialized cuboids and the minimum dimensionality of materialized cuboids . . . 90
6.24 Average error for the improved moment solver run in batch mode with and without applying the heuristic bounds for improving the error while varying the query dimensionality, the number of materialized cuboids, and the minimum dimensionality of materialized cuboids . . . 91
6.25 Improvement of error over time for the improved moment solver in online mode for various query dimensionality . . . 93
6.26 Junction tree for Example 20 . . . 96
6.27 Average time spent by each IPF solver in each phase of query execution in batch mode while varying the query dimensionality, the number of materialized cuboids and the minimum dimensionality of materialized cuboids . . . 101
6.28 Average error for each IPF solver run in batch mode while varying the query dimensionality, the number of materialized cuboids, and the minimum dimensionality of materialized cuboids . . . 102
6.29 Improvement of error over time for moment-based IPF solver in online mode for various query dimensionality . . . 105
6.30 Average execution time in batch mode for each solving technique for various query dimensionality . . . 106
6.31 Average approximation error in batch mode for each solving technique for various query dimensionality . . . 106
6.32 Average error at various times for each solver in online mode . . . 108

7.1 Expected relative size and density of a random k-dimensional projection of a base cuboid with support size 2^d . . . 112
7.2 Expected support size vs. actual size for cuboids of different dimensionality whose dimensions are chosen following the prefix and random strategy . . . 112

7.3 The number of materializable cuboids for each cuboid dimensionality for different budget factors for NYC and SSB datasets 113
7.4 Expected utility of a materialization strategy for various cuboid dimensionality for the naive solver 115
7.5 Comparing error, score, and utility for moment solver for various dimensionality of materialized cuboids 119
7.6 Expected utility for moment solver for various values of total dimensionality n 120
7.7 Expected utility for moment solver for various budget factors given by 2^b 120
7.8 Expected utility for moment solver for various support sizes of the base cuboid given by 2^d 120
7.9 Expected utility for moment solver for various query dimensionality q 121

8.1 Cuboid $C_{\{A,B,C\}}$ stored in dense, sparse row and sparse column formats 124
8.2 Projection times for each algorithm for various dimensionality of input cuboids 132
8.3 Projection times for each algorithm for various dimensionality of output cuboids 133
8.4 Projection times for each algorithm for various support sizes of input cuboids . 133
8.5 Comparison of storage between data cubes and moment store 137
8.6 Average execution time for the improved moment solver in batch mode for various query dimensionality when using each backend 138
8.7 Average error for the improved moment solver in batch mode for various query dimensionality when using each backend 138

List of Algorithms

1 Algorithm to select cuboids for materialization following the random or prefix strategy . . . 43
2 Algorithm that computes a simple plan to build cuboids by projecting the smallest subsuming cuboids materialized so far . . . 44
3 Algorithm for finding projections of queries relevant for answering it . . . 45

4 Algorithm to construct a basis from the equations yielded by projections of query cuboid C_Q . . . 61
5 Simple algorithm to improve bounds of variables in an equation using existing bounds on other variables . . . 62
6 Using linear programming to find bounds on query variables . . . 63
7 Efficient implementation of the transformation algorithm . . . 78
8 Transformation of $\boldsymbol{C}_I$ into $\boldsymbol{\mu}_I$ . . . 79
9 Algorithm for reconstructing C_Q from moments of its projections . . . 80
10 Algorithm for reconstructing C_Q by assuming unknown central moments are zero 83
11 Transformation algorithm that also applies bounds locally . . . 86
12 Reverse Transform . . . 86
13 Improved algorithm to approximate query cuboid from projections using moments 89
14 Iterative Proportional Fitting (IPF) . . . 95
15 Junction Tree IPF . . . 98
16 Moment-based IPF . . . 100
17 Minimum Spanning Tree Moment-Based IPF . . . 103

18 High-level algorithm for projecting a cuboid . . . 125
19 Aggregating duplicate keys in sparse row cuboids using sorting . . . 126
20 Algorithm for the projection loop for a sparse row or dense cuboid . . . 127
21 Simple algorithm for projecting keys . . . 128
22 Algorithm to generate the projection masks to compress bit vectors . . . 129
23 Algorithm for the projection loop for a sparse column cuboid . . . 130
24 Algorithm to transpose 64 × 64 bit matrix in place . . . 131
25 Algorithm for inserting a set to a set-trie . . . 135
26 Algorithm to retrieve values associated with every subset of a given set from a set-trie . . . 136

List of Experiments

6.1 Varying Query Dimensionality on Naive Solver 51
6.2 Varying Number of Materialized Cuboids in Naive Solver 53
6.3 Varying Minimum Dimensionality of Materialized Cuboids on Naive Solver 53
6.4 Varying Query Dimensionality on Linear Programming Solver in Batch Mode 64
6.5 Varying Number of Materialized Cuboids on Linear Programming Solver 68
6.6 Varying Minimum Dimensionality of Materialized Cuboids on Linear Programming Solver in Batch Mode 68
6.7 Studying Improvement of Error over Time for Linear Programming Solver in Online Mode While Varying Query Dimensionality 71
6.8 Comparing Moment Extrapolation Techniques in Batch Mode 85
6.9 Comparing Moment Extrapolation Techniques With Bounds in Batch Mode 86
6.10 Varying Query Dimensionality on Improved Moment Solver in Batch Mode 89
6.11 Varying Number of Materialized Cuboids on Improved Moment Solver in Batch Mode 92
6.12 Varying Minimum Dimensionality of Materialized Cuboids on Improved Moment Solver in Batch Mode 92
6.13 Studying Improvement of Error over Time for Improved Moment Solver in Online mode while Varying Query Dimensionality 93
6.14 Varying Query Dimensionality on IPF Solvers in Batch Mode 100
6.15 Varying Number of Materialized Cuboids on IPF Solvers in Batch Mode 103
6.16 Varying Minimum Dimensionality of Materialized Cuboids on IPF Solvers in Batch Mode 104
6.17 Online Experiment for IPF Solver Varying Query Dimensionality 105
6.18 Varying Query Dimensionality on all Solvers in Batch Mode 106
6.19 Online Experiments Varying Query Dimensionality 107
7.1 Finding Cuboid Dimensionality for Maximizing Utility for Naive Solver 115
7.2 Comparing Error, Score, and Utility for Moment Solver 118
7.3 Finding Cuboid Dimensionality for Maximizing Utility for Moment Solver 119
8.1 Varying Dimensionality of Input Cuboid for Projection Algorithms 131
8.2 Varying Dimensionality of Output Cuboid for Projection Algorithms 132
8.3 Varying Support Size of Input Cuboid for Projection Algorithms 132
8.4 Varying Query Dimensionality for Moment Solver in Batch Mode with Different Backends 136

1 Introduction

Data has become one of the most valuable assets for organizations in today's data-driven world. However, extracting meaningful insights from data is often challenging, particularly when dealing with large multidimensional datasets. In this regard, data cubes have emerged as powerful data analysis and visualization tools, enabling users to perform multidimensional data analysis (MDA) and online analytical processing (OLAP) tasks efficiently.

A data cube is a multidimensional representation of data that allows users to analyze and explore complex datasets along different dimensions, providing a more intuitive and flexible approach to data analysis. Data cubes organize data into a grid-like structure, with each cell representing a unique combination of dimensions and containing a measure value. The dimensions in a data cube represent the different attributes or characteristics of the data, while the measures represent the numeric values or metrics associated with the data.

Data cubes have been widely adopted in various fields, including business, finance, healthcare, and scientific research. They are useful whenever there is a need to analyze data from multiple dimensions, such as in the case of exploratory data analysis, trend analysis, and decision support systems. In addition, data cubes have been integrated into various software tools, such as database management systems and business intelligence platforms, making it easier for users to interact with and analyze data.

Instead of writing complex SQL queries, users interact with data cubes using simple, and often visual, query languages. These languages include basic operations such as aggregation, slicing, dicing, and drilling down or rolling up on dimensions, which can be combined and applied in various ways to analyze and summarize data in a data cube.

One of the main advantages of using data cubes is their ability to handle large volumes of data efficiently. By precomputing and storing the results of complex queries, data cubes can provide near-instantaneous response times to user queries, even for large datasets. Data is stored at different levels of detail, and users can analyze data from different perspectives and at varying levels of granularity.

Despite their advantages, data cubes have limitations, especially for high-dimensional data.

1. Storage Requirements: Data cubes can be huge, especially when dealing with high-dimensional data, requiring a significant amount of storage space, making them costly.
2. Data Preprocessing: Building a data cube requires preprocessing the data to aggregate it at different levels of granularity. This can be time-consuming, especially if the data is complex and requires significant transformations before it can be aggregated.
3. Data Sparsity: When working with high-dimensional data, it is common for many possible combinations of dimensions to have no data. This results in sparse data cubes, which can be challenging to work with.
4. Limited Flexibility: Data cubes are designed to support specific types of queries and are not always flexible enough to support more complex queries or ad-hoc analysis.

In this thesis, I describe Sudokube, a novel data cube system that allows fast querying of high-dimensional data. I motivate the need for high-dimensional data cubes and explore the challenges arising from high-dimensionality, such as scalability issues and their impact on query performance. I describe the architecture of Sudokube and how it overcomes these challenges and evaluate different storage models, indexing techniques, and query processing algorithms both from a theoretical and an experimental point of view. Finally, I investigate the applications of Sudokube in real-world scenarios.

1.1 Research Questions

This thesis aims to answer the following research questions.

1. What are the storage and computing requirements for materializing high-dimensional data cubes?
2. What are the existing approaches for data cubes and, in general, answering queries quickly? How do they fall short for high-dimensional data?
3. What are the motivations for high-dimensional data cubes?
4. How can we overcome the challenges for high-dimensional data cubes?
5. How do we select which cuboids to materialize in a high-dimensional data cube?
6. How do we store and project cuboids efficiently?
7. How can we reconstruct non-materialized cuboids at query time quickly and accurately from the materialized cuboids?
8. What kind of operations and aggregations are supported in this approach?

1.2 Contributions

This thesis presents my substantial contributions to the field of high-dimensional data cube processing. However, I also acknowledge the collaborative nature of this research, which involved various contributors.

In collaboration with my advisor, Prof. Christoph Koch, I explored and experimentally demonstrated the storage costs linked to high-dimensional data cubes. We observed the impracticality of maintaining fully materialized data cubes, which led to the design of Sudokube, an innovative data cube system supporting interactive querying on high-dimensional data. Sudokube is founded on two pivotal ideas: selectively materializing a subset of cuboids to manage storage costs and leveraging smaller cuboid projections for quick approximate query responses. The initial prototype of the system was built by Christoph, which I used as a foundation for my work.

A significant part of my work is centered around developing and implementing three distinct solving techniques for approximating query results from projections. These techniques employ linear programming, statistical moments, and iterative proportional fitting. I proved the correctness of these techniques and ran extensive experiments to profile their performance in terms of speed and accuracy. I optimized the initial version of the linear programming solver developed by Christoph, designed and implemented the moment solver from scratch, and refined it in collaboration with Dr. Peter Lindner. Further, Peter, Zhekai Jiang, and I co-developed the solver using the iterative proportional fitting method, and I built enhanced versions of the same.

Beyond approximation techniques, my work delved into strategies for materializing cuboids, deriving their expected utility analytically for different solving techniques, and validating these deductions experimentally. Peter and I worked with Fabian Jogl and Thomas Depian to establish the preliminary analysis for the utility of various materialization strategies, which I have refined in this thesis.

Christoph first proposed the idea of binary dimensions for data encoding to simplify both the theoretical groundwork and practical implementation of solving techniques. I introduced enhancements to the encoding process for increased speed, suggested alternative data encoding strategies, and examined various formats for storing cuboids to pinpoint the optimal design for superior performance. Tarindu Jayatilaka assisted me in designing a columnar layout for binary cuboids.

Moreover, I implemented various data structures and algorithms, enabling efficient indexing, selection, and projection of binary cuboids in the Sudokube system.

This comprehensive research resulted in three research papers, with two already published and one under review:

1. *High-dimensional Data Cubes* [13]: Sachin Basil John and Christoph Koch. *VLDB* 2022.
2. *Aggregation and Exploration of High-Dimensional Data Using the Sudokube Data Cube Engine* [14]: Sachin Basil John, Peter Lindner, Zhekai Jiang, and Christoph Koch. *SIGMOD* 2023.
3. *Fast Approximate Reconstruction of Joint Distributions from Low-Dimensional Projections* [15]: Sachin Basil John, Peter Lindner, Zhekai Jiang, and Christoph Koch. Submitted for review to *VLDB* 2023.

1.3 Thesis Outline

The organization of this thesis is as follows. Chapter 2 provides relevant background information and establishes the notations used throughout the thesis. In Chapter 3, the necessity for high-dimensional data cubes is explained, the current methodologies are discussed, and their shortcomings are brought to light. Chapter 4 introduces the Sudokube system, thoroughly elucidating its architecture and the fundamental ideas that underpin its design and functionality. The user-facing component of Sudokube, with a focus on data loading and querying processes, is described in Chapter 5. The focus then shifts in Chapter 6 to the diverse solving methods employed within Sudokube, along with proof of their correctness and results from their experimental evaluations. In Chapter 7, various strategies for selecting cuboids for materialization are examined, and their strengths and weaknesses are compared. Chapter 8 contains a detailed exposition of the backend implementation, describing the storage formats for cuboids and evaluating different projection algorithms. The thesis concludes with a summary of key findings and a discussion of potential avenues for future research.

2 Background

In this chapter, we go over some mathematical notations and essential background for various aspects of Sudokube described throughout the rest of this thesis.

2.1 Sets, Functions, and Vectors

Sets, vectors, and functions involving them are used frequently in this thesis. For some natural number n, let $[n]$ denote the set $\{1,\ldots,n\}$. For any set I, 2^I denotes the powerset of I containing all of its subsets. Moreover, $\{0,1\}^I$ denotes the set of functions $j\colon I \to \{0,1\}$ that map elements of the set I to either 0 or 1. The subset of I that is mapped by j to 1 is denoted using $\mathbb{1}_j$.

Example 1. *Let* $n = 4$ *and* $I = \{1,3,4\}$. *The function* $j\colon \{1 \mapsto 1, 3 \mapsto 1, 4 \mapsto 0\} \in \{0,1\}^I$ *maps elements of* I *to either* 0 *and* 1. *The elements of* I *mapped to* 1 *by* j *is given by* $\mathbb{1}_j = \{1,3\}$.

Throughout this thesis, sometimes, it is more convenient to concisely represent such functions using bit vectors. Formally, the elements of $\{0,1\}^I$ are treated as row vectors that are indexed by elements of I as follows. Suppose $I = \{i_1,\ldots,i_m\} \subseteq [n]$ with $i_1 < i_2 < \cdots < i_m$. Then for some function $x \in \{0,1\}^I$, we can construct the m-dimensional vector $\boldsymbol{x}$ by gathering $x(i)$ for every $i \in I$ as the i^{th} entry x_i of the vector, i.e., $\boldsymbol{x} = (x_i)_{i\in I}$. This vectorized representation of the function x is the one being referred to while writing $\boldsymbol{x} \in \{0,1\}^I$.

While explicitly specifying elements of vectors, the *reverse* order representation is used – $\boldsymbol{x} = (x_{i_m},\ldots,x_{i_1})$ or simply $\boldsymbol{x} = x_{i_m}\ldots x_{i_1}$. This format allows us to easily visualize these bit vectors as binary encodings of some natural number. The most significant bit representing the mapping x_{i_m} for the largest element i_m appears first on the left, and successive bits represent the mapping for the other elements in decreasing order.

Example 1 (continued). *The function* j *from above example can be concisely represented using the vector* $\boldsymbol{j} = (0,1,1)$ *or simply* 011. *Note that the domain* I *has to be understood from context.*

We order the set $\{0,1\}^I$ lexicographically in the explicit representation: If $\boldsymbol{x} = (x_{i_m},\ldots,x_{i_1})$ and $\boldsymbol{y} = (y_{i_m},\ldots,y_{i_1})$, then we let $\boldsymbol{x} \prec \boldsymbol{y}$ if and only if there exists $k \in I$ such that $x_i = y_i$ for all

$i \in I$ with $i > k$, and $x_k < y_k$. We use this order when we define *vectorized* version $\boldsymbol{f}$ of some function f with the set $\{0,1\}^I$ as its domain. $\boldsymbol{f}$ has $2^{|I|}$ entries, one for each element in $\{0,1\}^I$.

Example 1 (continued)**.** *Consider $I = \{1,3,4\}$ as before. Then, the eight vectors from $\{0,1\}^I$ are ordered as*

$$000 \prec 001 \prec 010 \prec 011 \prec 100 \prec 101 \prec 110 \prec 111$$

Consider some function $f\colon \{0,1\}^I \to \mathbb{R}$, defined for each of these eight vectors. Then the vectorized version $\boldsymbol{f}$ is given by

$$\boldsymbol{f} = \big(f(000), f(001), f(010), f(011), f(100), f(101), f(110), f(111)\big)^\top \in \mathbb{R}^8.$$

For any vector $\boldsymbol{x} \in \{0,1\}^I$, we define the restriction $\boldsymbol{x}_{\downarrow J}$ to some set $J \subseteq I$ as the restriction of the function to J. Conversely, for any vector $\boldsymbol{y} \in \{0,1\}^J$ and $I \supseteq J$, we define the extension $\boldsymbol{y}_{\uparrow J}$ to I by extending the domain of the function represented by $\boldsymbol{y}$. The additional elements in the domain are always mapped to zero. Note that restriction and extension are not inverse operations of each other. For vectors with disjoint domains, we also define their addition $\uplus$ as the disjoint union of their mappings. For some $J \subseteq I$, $\boldsymbol{x} \in \{0,1\}^I$, and $\boldsymbol{y} \in \{0,1\}^J$, we have the following formulas where $\mathbf{0}$ refers to the zero vector that maps all elements in $I \setminus J$ to zero.

$$\boldsymbol{x} = \boldsymbol{x}_{\downarrow J} \uplus \boldsymbol{x}_{\downarrow I \setminus J} \tag{2.1}$$

$$\boldsymbol{y}_{\uparrow I} = \boldsymbol{y} \uplus \mathbf{0} \tag{2.2}$$

When explicitly specifying vectors from multiple domains, we implicitly extend them to some common domain for better clarity, padding the missing bits using $*$.

Example 1 (continued)**.** *Consider $I = \{1,3,4\}$ and vector $\boldsymbol{x} = 011 \in \{0,1\}^I$. Then, for $J = \{1,4\}$, the restriction of $\boldsymbol{x}$ is given by*

$$\boldsymbol{y} = \boldsymbol{x}_{\downarrow J} = 01 \in \{0,1\}^J.$$

The extension of $\boldsymbol{y}$ is given by

$$\boldsymbol{y}_{\uparrow I} = 001 \in \{0,1\}^I.$$

Suppose we are additionally interested in restrictions and extensions involving another set $K = \{3\}$, then we explicitly denote the position of missing bits using $*$ as shown below.

$$\begin{aligned} \boldsymbol{y} &= \boldsymbol{x}_{\downarrow J} = 0{*}1 & \boldsymbol{y}_{\uparrow I} &= 001 \\ \boldsymbol{z} &= \boldsymbol{x}_{\downarrow K} = {*}1{*} & \boldsymbol{z}_{\uparrow I} &= 010 \end{aligned}$$

Finally, since J and K are disjoint and $I = J \cup K$, we have $\boldsymbol{x} = \boldsymbol{y} \uplus \boldsymbol{z}$.

X_5	X_4	X_3	X_2	X_1	Pr
0	0	0	1	1	0.25
0	1	1	0	1	0.3
0	1	1	1	0	0.1
1	0	0	0	1	0.35

(a) Joint distribution

X_4	X_1	Pr
0	1	0.6
1	0	0.1
1	1	0.3

(b) Marginal distribution (X_4, X_1)

X_1	Pr
0	0.1
1	0.9

(c) Marginal distribution X_1

Figure 2.1: An example of a 5-dimensional Bernoulli distribution

Given two sets $I, J \subseteq [n]$, another set $K \subseteq I \cap J$ common to both I and J, and any two vectors $\boldsymbol{x} \in \{0,1\}^I$ and $\boldsymbol{y} \in \{0,1\}^J$, we say $\boldsymbol{x}$ is consistent with $\boldsymbol{y}$ on the elements in K if and only if $\boldsymbol{x}_{\downarrow K} = \boldsymbol{y}_{\downarrow K}$.

2.2 Multivariate Bernoulli Distributions

Let $\boldsymbol{X} = (X_n, \ldots, X_1)$ be a vector of n random variables X_i, each taking values in $\{0,1\}$. Then the range of values $\boldsymbol{X}$ can take is exactly $\{0,1\}^{[n]}$. The *joint distribution* of the X_i, $i \in [n]$ is the function p mapping every $\boldsymbol{x} \in \{0,1\}^{[n]}$ to the probability

$$p(\boldsymbol{x}) = p(x_n, \ldots, x_1) = \Pr(X_n = x_n, \ldots, X_1 = x_1) = \Pr(\boldsymbol{X} = \boldsymbol{x}).$$

Such distributions are called *multivariate Bernoulli* or *n-dimensional Bernoulli distributions*.

For all $I \subseteq [n]$, we write $\boldsymbol{X}_I$ for the random vector $(X_i)_{i \in I}$ (ordered in the natural way). By p_I we denote the *marginal distribution* of $\boldsymbol{X}_I$, i.e. $p_I(\boldsymbol{x}) = \Pr(\boldsymbol{X}_I = \boldsymbol{x})$ for all $\boldsymbol{x} \in \{0,1\}^I$. Formally, for some $\boldsymbol{x} \in \{0,1\}^I$,

$$p_I(\boldsymbol{x}) = \sum_{\substack{\boldsymbol{y} \in \{0,1\}^{[n]} \\ \boldsymbol{y}_{\downarrow I} = \boldsymbol{x}}} p(\boldsymbol{y}). \tag{2.3}$$

The vectorizations $\boldsymbol{p}$ and $\boldsymbol{p}_I$ are obtained as described in the previous subsection.

Example 2. *Figure 2.1a shows an example of the joint distribution of five Bernoulli random variables $(X_5, \ldots, X_1)$. All missing entries have a probability of zero. For this example, $n = 5$, $p(00011) = 0.25$, $p(10001) = 0.35$, and $p(10101) = 0.0$. Then, the marginal distribution of (X_4, X_1) is obtained as shown in Figure 2.1b by summing up the probabilities of entries having the same values for X_4 and X_1 in the joint distribution. Then, we write $p_{\{4,1\}}(01) = 0.6$ and $p_{\{4,1\}}(10) = 0.3$. Similarly, we write $p_{\{1\}}(0) = 0.1$ and $p_{\{1\}}(1) = 0.9$ from the marginal distribution of X_1 from Figure 2.1c.*

For every $i \in [n]$, we define θ_i as the expected value of the random variable X_i and $\overline{\theta}_i$ as its complement.

$$\theta_i = \mathbb{E}[X_i] = \Pr(X_i = 1) \qquad \overline{\theta}_i = 1 - \theta_i = \Pr(X_i = 0) \tag{2.4}$$

For any $\boldsymbol{x} \in \{0,1\}^{[n]}$, we define the raw moment $\mu(\boldsymbol{x})$ and the central moment $\sigma(\boldsymbol{x})$ as follows.

$$\mu(\boldsymbol{x}) = \mathbb{E}\left[\prod_{i \in \mathbb{1}_{\boldsymbol{x}}} X_i\right] \tag{2.5}$$

$$\sigma(\boldsymbol{x}) = \mathbb{E}\left[\prod_{i \in \mathbb{1}_{\boldsymbol{x}}} (X_i - \theta_i)\right] \tag{2.6}$$

These moments capture inter-dependencies between random variables and constitute alternate representations of the probability distribution. It is clear from the definitions that both $\mu(\mathbf{0})$ and $\sigma(\mathbf{0})$ are equal to 1, where $\mathbf{0} \in \{0,1\}^{[n]}$ is the zero vector with all elements mapped to 0. Furthermore, for some $\boldsymbol{x} \in \{0,1\}^{[n]}$ that maps exactly a single element $i \in [n]$ to 1 and the rest to 0, $\mu(\boldsymbol{x}) = \theta_i$ and $\sigma(\boldsymbol{x}) = 0$.

In addition to the joint distribution, moments can also be defined on marginal distributions. For some $I \subseteq [n]$ and $\boldsymbol{y} \in \{0,1\}^I$, we define the moments of the marginal distribution for I as shown below. These expressions are closely related to those in Equations (2.5) and (2.6) as stated in Proposition 1.

$$\mu_I(\boldsymbol{y}) = \mathbb{E}\left[\prod_{i \in \mathbb{1}_{\boldsymbol{y}}} X_i\right] \qquad \sigma_I(\boldsymbol{y}) = \mathbb{E}\left[\prod_{i \in \mathbb{1}_{\boldsymbol{y}}} (X_i - \theta_i)\right]$$

Proposition 1. *For any sets $J \subseteq I \subseteq [n]$ and $\boldsymbol{x} \in \{0,1\}^J$,*

(i) $\mu_J(\boldsymbol{x}) = \mu_I(\boldsymbol{x}_{\uparrow I}) = \mu(\boldsymbol{x}_{\uparrow [n]})$, *and*

(ii) $\sigma_J(\boldsymbol{x}) = \sigma_I(\boldsymbol{x}_{\uparrow I}) = \sigma(\boldsymbol{x}_{\uparrow [n]})$.

Proof. The proof follows immediately from $\mathbb{1}_{\boldsymbol{x}} = \mathbb{1}_{\boldsymbol{x}_{\uparrow I}} = \mathbb{1}_{\boldsymbol{x}_{\uparrow [n]}}$ since the new dimensions are mapped to 0. □

Example 2 (continued)**.** *For the probability distribution shown in Figure 2.1, $\theta_1 = 0.9$, $\overline{\theta}_1 = 0.1$, $\theta_4 = 0.4$, and $\overline{\theta}_4 = 0.6$. The raw moment $\mu(01001)$ is computed as*

$$\mu(01001) = \mathbb{E}[X_4 \cdot X_1] = \Pr(X_4 = 1, X_1 = 1) = 0.3$$

Similarly, the central moment $\sigma(01001)$, which is the covariance of X_4 and X_1, is computed as

$$\begin{aligned}\sigma(01001) &= \mathbb{E}[(X_4 - \theta_4) \cdot (X_1 - \theta_1)] \\ &= \theta_4 \cdot \theta_1 \cdot \Pr(X_4 = 0, X_1 = 0) - \theta_4 \cdot \overline{\theta}_1 \cdot \Pr(X_4 = 0, X_1 = 1) \\ &\quad - \overline{\theta}_4 \cdot \theta_1 \cdot \Pr(X_4 = 1, X_1 = 0) + \overline{\theta}_4 \cdot \overline{\theta}_1 \cdot \Pr(X_4 = 1, X_1 = 1) \\ &= 0.4 \cdot 0.9 \cdot 0 - 0.4 \cdot 0.1 \cdot 0.6 - 0.6 \cdot 0.9 \cdot 0.1 + 0.6 \cdot 0.1 \cdot 0.3 \\ &= 0 - 0.024 - 0.054 + 0.018 = -0.06\end{aligned}$$

Note that from Proposition 1, we have $\mu_{\{4,1\}}(11) = \mu(01001)$ and $\sigma_{\{4,1\}}(11) = \sigma(01001)$.

We now establish a relationship between μ_I and σ_I for some set $I \subseteq [n]$ in the following lemma.

Lemma 2. *For any set $I \subseteq [n]$ and vector $\boldsymbol{x}, \boldsymbol{y} \in \{0,1\}^I$, we have*

$$\mu_I(\boldsymbol{x}) = \sum_{\mathbb{1}_{\boldsymbol{y}} \subseteq \mathbb{1}_{\boldsymbol{x}}} \sigma_I(\boldsymbol{y}) \prod_{i \in \mathbb{1}_{\boldsymbol{x}} \setminus \mathbb{1}_{\boldsymbol{y}}} \theta_i$$

Proof. To establish this relationship, we start with the definition of $\mu_I(\boldsymbol{x})$, and we rewrite the term X_i as $(X_i + \theta_i) - \theta_i$. Then, multi-binomial expansion is used to express the product of the difference as a sum of products. Each term in the sum corresponds to a partition of the set $\mathbb{1}_{\boldsymbol{x}}$ into K and $\mathbb{1}_{\boldsymbol{x}} \setminus K$. Using linearity of expectation, we turn the expectation of the sum into a sum of expectation scaled by a factor. By interpreting the subset K as $\mathbb{1}_{\boldsymbol{y}}$ for some $\boldsymbol{y} \in \{0,1\}^I$, the expectation is equal to $\sigma_I(\boldsymbol{y})$, thus establishing the relationship between $\mu_I(\boldsymbol{x})$ and $\sigma_I(\boldsymbol{y})$.

$$\begin{aligned}
\mu_I(\boldsymbol{x}) &= \mathbb{E}\left[\prod_{i \in \mathbb{1}_{\boldsymbol{x}}} \big((X_i - \theta_i) + \theta_i\big)\right] \\
&= \mathbb{E}\left[\sum_{K \subseteq \mathbb{1}_{\boldsymbol{x}}} \prod_{k \in K} \big(X_k - \theta_k\big) \cdot \prod_{\ell \in \mathbb{1}_{\boldsymbol{x}} \setminus K} \theta_\ell\right] && \text{(using multi-binomial expansion)} \\
&= \sum_{K \subseteq \mathbb{1}_{\boldsymbol{x}}} \mathbb{E}\left[\prod_{k \in K} \big(X_k - \theta_k\big)\right] \cdot \prod_{\ell \in \mathbb{1}_{\boldsymbol{x}} \setminus K} \theta_\ell \\
&= \sum_{\mathbb{1}_{\boldsymbol{y}} \subseteq \mathbb{1}_{\boldsymbol{x}}} \sigma_I(\boldsymbol{y}) \cdot \prod_{\ell \in \mathbb{1}_{\boldsymbol{x}} \setminus \mathbb{1}_{\boldsymbol{y}}} \theta_\ell && \text{(iterate over } \boldsymbol{y} \text{ such that } K = \mathbb{1}_{\boldsymbol{y}}) \ \square
\end{aligned}$$

One can easily vectorize θ, μ, μ_I, σ and σ_I following the approach in the previous subsection to obtain $\boldsymbol{\theta}$, $\boldsymbol{\mu}$, $\boldsymbol{\mu}_I$, $\boldsymbol{\sigma}$ and $\boldsymbol{\sigma}_I$. Furthermore, Proposition 1 guarantees that, for any $I \subseteq [n]$, $\boldsymbol{\mu}_I$ is a subvector of $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}_I$ a subvector of $\boldsymbol{\sigma}$.

We close this section by introducing a metric for probability distributions. The *(joint) entropy* of $\boldsymbol{X} = (X_n, \ldots, X_1)$ with joint distribution p is given by

$$\mathrm{H}(p) = -\sum_{\boldsymbol{x}:\ p(\boldsymbol{x}) \neq 0} p(\boldsymbol{x}) \cdot \log\big(p(\boldsymbol{x})\big) \tag{2.7}$$

where we use the base 2 logarithm. For n-dimensional Bernoulli distributions, $\mathrm{H}(p)$ takes values between 0 (when $\boldsymbol{X}$ has one outcome with probability 1) and n (when $\boldsymbol{X}$ is distributed uniformly). Intuitively, entropy is a measure of non-uniformity.

Example 2 (continued)**.** *The entropy of the joint distribution in Figure 2.1 is given by*

$$H(p) = -\big(0.25\log(0.25) + 0.3\log(0.3) + 0.1\log(0.1) + 0.35\log(0.35)\big) \approx 1.883$$

Details and additional background on the above notions can be found in the appendix of [63].

2.3 Kronecker Product of Matrices

The Kronecker product of two matrices $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ and $\boldsymbol{B} \in \mathbb{R}^{k \times \ell}$ is the $mk \times n\ell$ block matrix

$$\boldsymbol{A} \otimes \boldsymbol{B} = \begin{pmatrix} a_{11}\boldsymbol{B} & \dots & a_{1n}\boldsymbol{B} \\ \vdots & \ddots & \vdots \\ a_{m1}\boldsymbol{B} & \cdots & a_{mn}\boldsymbol{B} \end{pmatrix}.$$

We write $\boldsymbol{A}^{\otimes m}$ for $\boldsymbol{A} \otimes \cdots \otimes \boldsymbol{A}$ (m times).

Proposition 3. *If $\boldsymbol{M}_1, \dots, \boldsymbol{M}_m$ are 2×2 matrices and $\boldsymbol{I}_2$ is the 2×2 identity matrix, then*

$$\boldsymbol{M}_1 \otimes \ldots \otimes \boldsymbol{M}_m = \prod_{i=1}^{m} \left(\boldsymbol{I}_2^{\otimes(i-1)} \otimes \boldsymbol{M}_i \otimes \boldsymbol{I}_2^{\otimes(m-i)}\right), \tag{2.8}$$

where the right-hand involves matrix products. These products fully commute with each other.

This follows from the well-known properties of the Kronecker product. The proof is included for completeness.

Proof. We use the following particular identities [109], where all $\boldsymbol{A}$ and $\boldsymbol{A}_i$ are $r \times r$ matrices, and all $\boldsymbol{B}$ and $\boldsymbol{B}_i$ are $s \times s$ matrices:

$$\left(\prod_{i=1}^{m} \boldsymbol{A}_i\right) \otimes \left(\prod_{i=1}^{m} \boldsymbol{B}_i\right) = \prod_{i=1}^{m} \left(\boldsymbol{A}_i \otimes \boldsymbol{B}_i\right) \tag{2.9}$$

$$(\boldsymbol{A}_1 \otimes \boldsymbol{B}_1) \cdot (\boldsymbol{A}_2 \otimes \boldsymbol{B}_2) = (\boldsymbol{A}_1 \boldsymbol{A}_2) \otimes (\boldsymbol{B}_1 \boldsymbol{B}_2) \tag{2.10}$$

$$(\boldsymbol{A} \otimes \boldsymbol{I}_s) \cdot (\boldsymbol{I}_r \otimes \boldsymbol{B}) = (\boldsymbol{I}_r \otimes \boldsymbol{B}) \cdot (\boldsymbol{A} \otimes \boldsymbol{I}_s). \tag{2.11}$$

Furthermore, we note that $\boldsymbol{I}_2^{\otimes k} = \boldsymbol{I}_{2^k}$ is the $2^k \times 2^k$ identity matrix. For $m = 1$, there is nothing to show. We prove the general case by induction on $m > 1$. Letting $\boldsymbol{N}_i = \boldsymbol{I}_2^{\otimes(i-1)} \otimes \boldsymbol{M}_i \otimes \boldsymbol{I}_2^{\otimes(m-i-1)}$, and starting from the right-hand side of (2.8), we have

$$\begin{aligned}
&\left(\prod_{i=1}^{m-1} (\boldsymbol{N}_i \otimes \boldsymbol{I}_2)\right) \cdot \left(\boldsymbol{I}_2^{\otimes(m-1)} \otimes \boldsymbol{M}_m\right) \\
&= \left(\left(\prod_{i=1}^{m-1} \boldsymbol{N}_i\right) \otimes \boldsymbol{I}_2^{m-1}\right) \cdot \left(\boldsymbol{I}_2^{\otimes(m-1)} \otimes \boldsymbol{M}_m\right) && \text{(using (2.9))} \\
&= \left((\boldsymbol{M}_1 \otimes \cdots \otimes \boldsymbol{M}_{m-1}) \otimes \boldsymbol{I}_2\right) \cdot \left(\boldsymbol{I}_2^{\otimes(m-1)} \otimes \boldsymbol{M}_m\right) && \text{(using Induction Hypothesis)} \\
&= \boldsymbol{M}_1 \otimes \ldots \otimes \boldsymbol{M}_m,
\end{aligned}$$

using (2.10). Commutativity on the right-hand side of (2.8) follows directly by inspecting the factors i and $i + 1$ and applying (2.11). □

2.4 Data Cube

The basic idea behind a data cube [39] is to represent data in a multi-dimensional space, where each axis represents a different dimension or attribute of the data. The data is then aggregated over each combination of attribute values, resulting in a multi-dimensional array, or cube. We

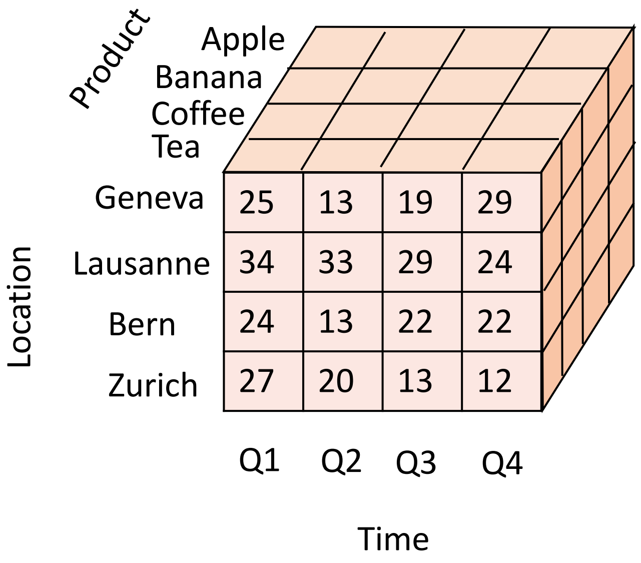


Figure 2.2: An example data cube for sales data

refer to the values in dimension as *keys*, the combination of keys from different dimensions as *cells*, and the aggregate value associated with each cell as *measures*. The primary aggregation operator is typically the summation operator, but other aggregations such as count, average, minimum, maximum, variance, and standard deviation are also commonly used.

Example 3. *Consider a data cube for storing the sales data for a retail corporation. It might have dimensions such as product, time, and region, and the cells in the cube would represent aggregated measures such as total sales, average price, or number of units sold. Figure 2.2 shows an instance of such a data cube. The product dimension has keys representing different items — tea, coffee, banana, and apple. Similarly, each key of the location dimension represents a city, and that of the time dimension represents a fiscal quarter. Note that only the measures for tea are shown in the figure for brevity.*

It is very common for dimensions in a data cube to be hierarchical. The dimensions are arranged in levels, with the highest level representing the most general category and the lower levels representing increasingly specific subcategories. By enabling querying of data at different levels of granularity, data cubes allow users to gain insights into trends and patterns quickly.

Due to limitations that restrict the number of supported dimensions that we will go into later in this thesis, data cubes typically organize data using a star or snowflake schema. In a star schema, a central fact table stores the measure values and the combination of keys at the lowest level of the dimensional hierarchy. The data cube would be built on this central *fact table*, storing data at the finest granularity. Then, separate *dimension tables* store additional attributes of each dimension and are linked to the central fact table through foreign-key constraints. The term "star schema" comes from the visual representation of the database schema. The fact table is in the center of the schema, surrounded by several dimension tables, which resemble the points of a star. A snowflake schema further normalizes data in the dimension tables into a set of related tables, each representing a specific level of hierarchy within a dimension. All these tables are related through foreign keys, forming a hierarchical structure that can be navigated while querying.

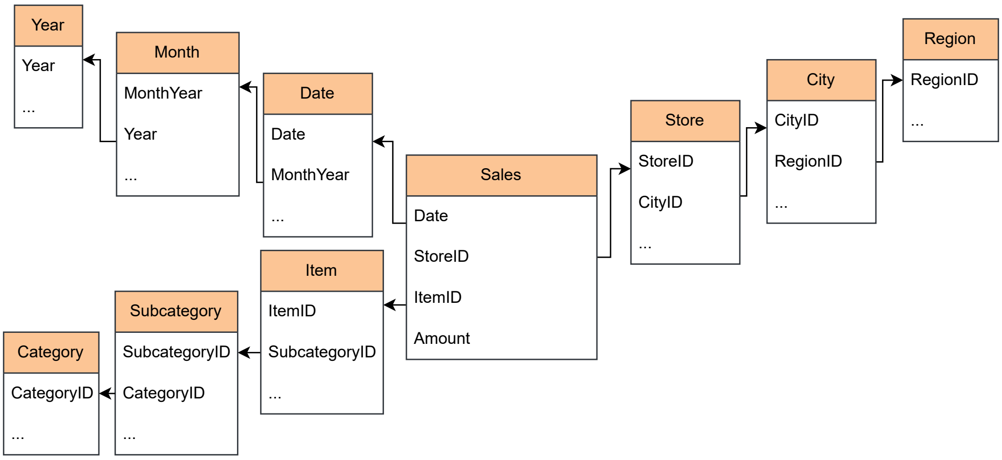


Figure 2.3: A snowflake schema for sales data with a central fact table and additional dimension tables for location, product, and time.

Example 3 (continued)**.** *In the example sales data cube, the time dimension could have levels for year, quarter, month, and day, as shown in Figure 2.3. Similarly, the product dimension could have levels for the category, subcategory, and individual product, and the location dimension could have levels for region, city, and individual stores.*

Data cubes are built by defining views that aggregate measures grouped by the subsets of dimensions and materializing these views. We refer to these views as *cuboids*. The *base cuboid* contains all the dimensions, and all other cuboids are its projections. All these cuboids form a lattice based on their projection hierarchy. When we speak of data cubes, we refer to the lattices of all their cuboids. Given a n-dimensional data cube, for each $0 \le k \le n$, there are $\binom{n}{k}$ many k-dimensional projections.

Example 3 (continued)**.** *The sales data cube has three dimensions – product, location, and time. So, the base cuboid is three-dimensional, with three two-dimensional cuboids, three one-dimensional cuboids, and one zero-dimensional cuboid as its projections as shown in Figure 2.4.*

Users query the data cube using simple operations shown in Figure 2.5. The basic operations that can be performed on a data cube include pivoting, slicing, dicing, and drilling down or rolling up on dimensions. These operations can be performed on one or more dimensions of the data cube and can be combined to provide more complex views of the data. Slicing selects

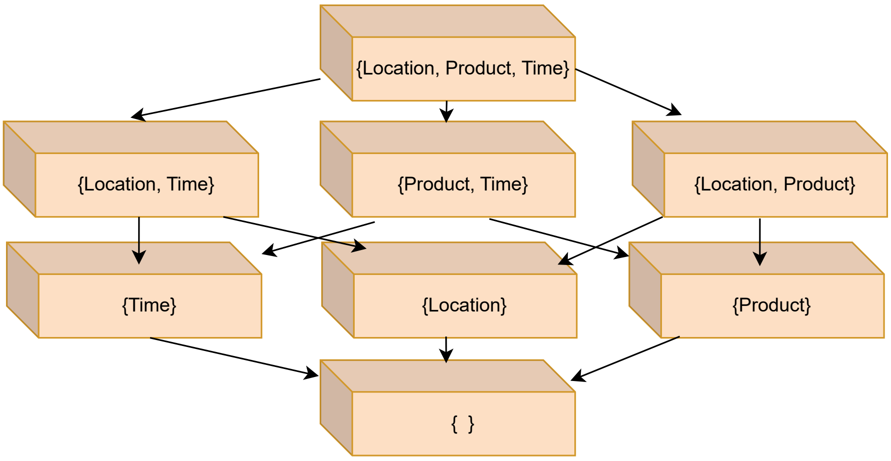


Figure 2.4: The lattice of cuboids for the sales data cube

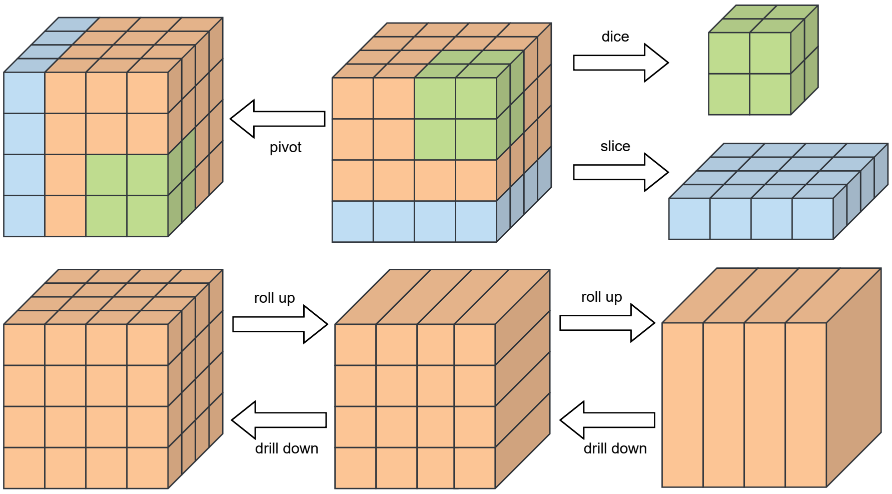


Figure 2.5: An overview of the high-level operations that can be applied on a data cube

a subset of the data cube by fixing the keys of one or more dimensions. Dicing is similar to slicing, but instead of fixing the keys of one or more dimensions, we select a subset of the keys for those dimensions. Drilling down or rolling up involves navigating the hierarchy of dimensions to either add or remove levels of granularity. Finally, pivoting rotates the data cube to change the dimensions displayed along the horizontal and vertical axis in the output table.

Example 3 (continued)**.** *For the sales data cube, users may be interested in the total sales for a particular product in a specific region over a certain period. They could use a slice operation to filter the data for particular products and regions and a dice operation to narrow down the time period to the range they are interested in. By rolling up on the location dimension, they can get the total sales for that product across all regions in the specified time period. Alternatively, they can drill down on the location dimension to get the total sales for that product for individual cities in that region. Pivoting the data cube allows the user to change the orientation of the output to have each row denoting different quarters and each column different cities.*

3 Motivation

Data cubes are important tools for data analysis, and in the previous chapter, we covered basic terminology and operations. In this chapter, we shall discuss the techniques that allow data cubes to answer queries quickly. We will then explore several scenarios that motivated my research where having a data cube built with a high number of dimensions could be beneficial. We will also cover why the existing approaches for data cubes struggle in these scenarios.

3.1 Classical Implementation of Data Cubes

A data cube traditionally achieves fast response times for queries by precomputing all cuboids in its lattice by projecting other cuboids, starting from the base cuboid. Then, when users interact with the data cube, the sequence of query operations applied by the user so far can be interpreted as a single aggregation query with the following format.

```sql
select dimension1, dimension2, ..., sum(measure) as aggregated_measure
from facttable
where condition1 and condition2 and ...
group by dimension1, dimension2, ...
```

The `where` clause contains conditions formed from slice and dice operations. Dimensions are added to the `group by` clause when they are drilled down on and removed when they are rolled up on. Any such query can be answered from the precomputed cuboid that contains all the dimensions in the `group by` and `where` clauses without performing any aggregation. Rolling up on a hierarchical dimension to aggregate on a coarser level of that dimension does involve joins and additional aggregations. These operations are slower in general, as we shall see later in this chapter.

In general, the number of dimensions n in a data cube is usually kept low to avoid the astronomical storage and compute costs for precomputing all 2^n cuboids when n is large. However, there are several scenarios where having a high-dimensional data cube would be useful.

3.2 Advantages of High Dimensional Cubes

While classical data cubes typically encompass a limited number of dimensions, there exist numerous scenarios in which a high-dimensional data cube – one that facilitates querying across a much larger set of dimensions – could deliver substantial advantages to users.

3.2.1 Off-the-shelf OLAP

Data cubes are widely used in Online Analytical Processing (OLAP), where large volumes of multidimensional data are analyzed from different perspectives to gain insights and make informed decisions. Data cubes offer a user-friendly interface that allows users to interact with data, explore it quickly and easily, and identify patterns and trends.

Data is often stored in different formats and different locations. Furthermore, the data quality may be poor, and the data may contain duplicates or inconsistencies. Before the data can be queried, it goes through an Extract-Transform-Load (ETL) pipeline [62] with three steps. The first step is extracting data from various sources such as databases, flat files, and web services. The extraction phase involves identifying the data that needs to be extracted and pulling it from the source systems into a staging area. This step is followed by the transformation phase, where the extracted data is transformed into a suitable format for analysis. This may involve cleaning the data, removing duplicates, and resolving inconsistencies. The transformation phase may also include aggregating or summarizing the data and creating derived datasets that are useful for analysis. Finally, in the loading phase, the transformed data is loaded into a data cube by mapping the data to the appropriate dimensions.

Designing an ETL pipeline is a complex and time-consuming task. As the number of dimensions supported by the data cube is limited, data needs to be reduced to a minimal number of dimensions during the transformation phase before it can be loaded into a data cube. This means that ETL designers need to anticipate what queries will be asked and carefully select the dimensions to be loaded into the data cube. The whole data cube must be rebuilt if the ETL designers change their minds and want to modify some dimensions. Building a data cube can take anywhere from a few hours to several days, depending on the data size [23]. Faulty ETL scripts can also necessitate rebuilding data cubes. A bug in the script that incorrectly transforms the data and maps it to the wrong dimensions can go unnoticed during the cube construction time. It may only be discovered after a series of queries to analyze the data.

Schema change is another issue in ETL pipeline design. Columns could be renamed without notice while loading data from a relational update stream into a data cube. This is common in large-scale event logging systems, such as those used in data center operation management, where many software packages produce events and frequently get updated. Suppose some column gets renamed in the log stream, for example, due to some software update, and the ETL pipeline is not updated. In that case, the renamed column will be ignored, and the corresponding data not loaded into the data cube.

However, an OLAP engine that supports high-dimensional data cubes can come ready to use right after installation, with a universal loader that reads all enterprise data into the data cube, whether clean or not, without discarding anything. Design regret can largely be eliminated as the ETL designers no longer have to distill the data to a minimal number of dimensions. Schema change is no longer a problem, as old and new columns can be loaded as independent dimensions in a high-dimensional data cube. Once the data is loaded into the data cube, users can use its interactive data exploration features to explore transformations on the data that make it easier to analyze. Data cubes excel in aggregation queries that routinely form part of such exploration. Once these transformations have been identified, users can define a view that applies these transformations on the existing data cube or build a new one with the refined ETL scripts. This saves a lot of time and effort for the designers.

3.2.2 Exploration of High-dimensional Data

Several applications with natively high-dimensional data can profit from analytical querying in a data-cube-style system. One class of such scenarios is exploring feature-rich data to be fed into machine learning pipelines as part of feature engineering.

Feature engineering is a crucial step in the machine learning pipeline that significantly impacts the performance of the final model. It improves the accuracy and efficiency of machine learning algorithms by extracting the most relevant features from data. These are often identified by exploring the data to find patterns, trends, and insights that are most predictive of the target variable. Some standard techniques used during data exploration include data visualization (such as scatter plots, heat maps, and histograms), summary statistics (such as mean, median, and standard deviation), and dimensionality reduction (such as principal component analysis or t-SNE).

Exploratory data analysis is a complex and iterative process; the person doing that is usually the bottleneck of the loop. Humans have a significant cognitive load while analyzing high-dimensional data, especially without specialized tools. Data cube systems are known for their easy-to-use visual interfaces that have successfully reduced the perceived complexity of massive datasets. They excel at computing summary statistics over different subsets of data and features and would be an excellent tool for data scientists if they could support high-dimensional data. Further motivation for using data cubes for feature engineering can be found in prior work [22], [59], [60] that proposes a similar interface for evaluating the accuracy of different models on subsets of data.

Exploratory data analysis can be applied to find trends and insights even in datasets not used for machine learning. For example, New York City Parking Violations [31] is a dataset that contains information regarding parking violations issued in New York between 2014 and 2021. This is an example of a dataset that humans in law enforcement would need to explore and analyze. It inherently contains more than three or four dimensions typically present in classical data cubes. Such datasets are most likely explored with rudimentary tools currently due to the lack of a better alternative. However, high-dimensional data cubes, capable of loading all the data dimensions, enable analysts to quickly discover trends and patterns in these datasets.

3.2.3 Avoiding Snowflake Schema

Due to a limited number of supported dimensions, data cubes typically organize data as a star or snowflake schema for representing hierarchical dimensions. A central fact table stores data for every dimension combination in such a schema, but only at the finest granularity. The dimensional tables store the relationship between keys at different levels of the hierarchy and are linked to each other through foreign-key constraints. To coarsen the aggregation for some dimension, the aggregation is first computed at the finest granularity from the data cube built on the fact table. Then the dimension is coarsened level by level by successively joining the current result with a dimension table that coarsen keys to the next higher level and then aggregating the result grouped by the coarsened key. Performing the joins and aggregations at query time can slow down queries significantly.

Without practical limitations on the dimensionality of data cubes, it would be possible to construct high-dimensional cubes that can perform aggregations across hierarchical dimensions without the need for joins. Instead of limiting the users to what is traditionally represented by a single dimension, we can generously grant dimension status to attributes that otherwise would not be considered dimensions. Every dimension attribute can be turned into individual dimensions, and all the dimension tables would be collapsed into the fact table. For instance, instead of a single *time* dimension, one can have individual dimensions for week, month, quarter, and year attributes. High-dimensional data cubes would have no difficulty loading the data from the denormalized table despite it having all the additional dimensions. Coarse-grained aggregations can be performed on such data cubes without joins, as each level of the hierarchy is a dimension of its own in the data cube.

3.2.4 More Powerful Queries

High-dimensional cubes enable users to run more powerful queries than those supported by classical data cubes. If there is no restriction on the number of supported dimensions in the data cube, users can break up what is traditionally a single dimension into multiple dimensions. For instance, a Name dimension can be split into FirstName and LastName dimensions, or even one dimension for every character position in the string. This allows pattern-matching queries on the original dimension to be expressed using roll-up and slice operations on the new dimensions. For example, in the case name was split into two dimensions for first and last names, rolling up on the dimension for first names aggregates the measure values grouped by last names. Alternatively, if every character position in the name is a queryable dimension, one can filter the names to only those starting with some character by slicing on that dimension.

The idea of splitting dimensions into smaller components can be taken further to enable even more powerful pattern-matching queries. Consider a dimension with a domain containing m keys. Any key in this domain can be represented using $\lceil \log_2 m \rceil$ bits without any loss of information. In a high-dimensional data cube, each bit of the binary representation of keys can be turned into a queryable dimension. This allows aggregations to be performed according to specific groupings of the keys. For example, suppose there exists a Year dimension with 10 keys. This dimension can be replaced by 4 binary dimensions, say y_3, y_2, y_1, and y_0, in a high-dimensional data cube. Querying all 4 binary dimensions is equivalent to querying the Year dimension. However, one can query a subset of these dimensions to group years according to some pattern. For example, querying y_3 and y_2 groups 4 years together, and querying y_0 splits years into those at odd and even positions.

Splitting dimensions into binary dimensions also helps encode the structure and hierarchy among dimensions more efficiently. Binary dimensions encoding different levels of hierarchical dimensions can overlap, reducing the overall number of dimensions required. Consider a dimension for time with hierarchy Year > Quarter > Month. Suppose the domain consists of 10 years, each with 4 quarters, which in turn, comprises 3 months for a total of 120 entries at the granularity of months, 40 entries at the granularity of quarters, and 10 entries at the granularity of years. Instead of splitting Year, Quarter, and Month into 4, 6, and 7 disjoint binary dimensions totaling 17 if the binary dimensions are reused, the total would only be 8 binary dimensions to represent the entire hierarchy. The four binary dimensions identifying the year can be shared between all levels, the quarter can be identified with two additional dimensions, and finally, the month can be identified with two more additional dimensions. Encoding hierarchical dimensions this way yields a continuous extension of the hierarchy in the form of prefixes of these binary dimensions. The first four binary dimensions represent years, and the first six represent quarters, but additionally, the first five dimensions now represent half-years.

3.3 The Infeasibility of High-dimensional Data Cubes

Despite all the advantages of having high-dimensional data cubes, they are not available in practice due to several challenges.

The first challenge is the storage costs associated with high-dimensional data cubes. For simplicity of analysis, let us assume that there are n dimensions, and every dimension has m distinct keys in its domain. The overall storage cost of a data cube is the cost of materializing all of its cuboids. While analyzing the storage costs of the cuboids, it is important to consider two storage formats separately.

First, let us examine the *dense* format where cuboids are stored as multidimensional arrays. Storing a k-dimensional projection of the n-dimensional data in the dense format requires storing m^k cells and there are $\binom{n}{k}$ as many such projections. When the value of n is in the range of hundreds, even for a moderate value of m, say 10, storing even a single k-dimensional cuboid is infeasible for $k \geq 20$. All cells take up space even though entries in most of the cells are likely to be zero in such high-dimensional setting due to the curse of dimensionality.

Next, we examine the *sparse* format that is more suitable for high-dimensional cuboids. This format stores only the cells with non-zero entries and is therefore more efficient for storing sparse cuboids where most entries are zero. The number of cells with non-zero entries and therefore, the storage cost of sparse cuboids depend significantly on data distributions.

Without loss of generality, let the number of non-zero entries in the base cuboid be m^d. Consider a scenario where these m^d entries are spread uniformly and randomly among the m^n cells of the base cuboid. What is the expected size of a random k-dimensional projection in this case? This cuboid has m^k cells whose entries are obtained by aggregating the entries of m^{n-k} cells from the base cuboid. Pick one of these m^k cells. This cell has a zero entry if and only if all the corresponding m^{n-k} cells contain zero entries. Let X be a random variable that denotes the number of cells that contain non-zero entries among those m^{n-k} cells. Then

$$\Pr(\text{cell is non-zero in cuboid}) = 1 - \Pr(\text{cell is zero in cuboid}) = 1 - \Pr(X = 0)$$

The random variable X follows a hyper-geometric distribution describing the number of successes when sampling m^{n-k} values without replacement from a population of m^n among which only m^d values result in a success. Therefore,

$$\Pr(X = 0) = \frac{\binom{m^d}{0}\binom{m^n - m^d}{m^{n-k}}}{\binom{m^n}{m^{n-k}}}$$

When k is sufficiently large that the sample size m^{n-k} is much smaller than the population size of m^n, replacement does not really have an impact and the hyper-geometric distribution can be approximated using a binomial distribution with the probability of success calculated from the number of success values in the population.

$$\Pr(X = 0) \approx \binom{m^{n-k}}{0}\left(\frac{m^d}{m^n}\right)^0\left(1 - \frac{m^d}{m^n}\right)^{m^{n-k}} = \left(1 - \frac{m^{d-k}}{m^{n-k}}\right)^{m^{n-k}}$$

For very large values of n, applying the limit $n \to \infty$, we get

$$\lim_{n\to\infty} \Pr(X = 0) = e^{-m^{d-k}}.$$

From this, the expected size of the k-dimensional projection relative to the base cuboid and its limit are given by

$$\begin{aligned}\lim_{n\to\infty} \mathbb{E}(\text{relative size of cuboid}) &= \frac{m^k}{m^d} \lim_{n\to\infty} \Pr(\text{cell is non-zero in cuboid}) \\ &= m^{k-d}(1 - e^{-m^{d-k}})\end{aligned}$$

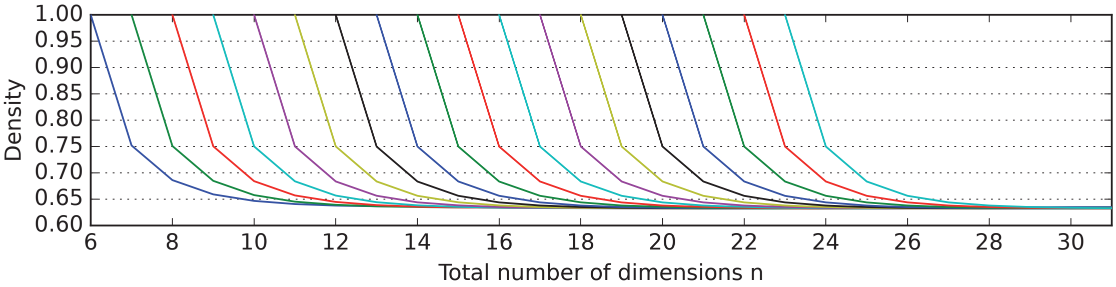


Figure 3.1: Simulation results for the density of a random d-dimensional projection of a n-dimensional cuboid. Each curve starts at $n = d$ for various d values between 6 and 23.

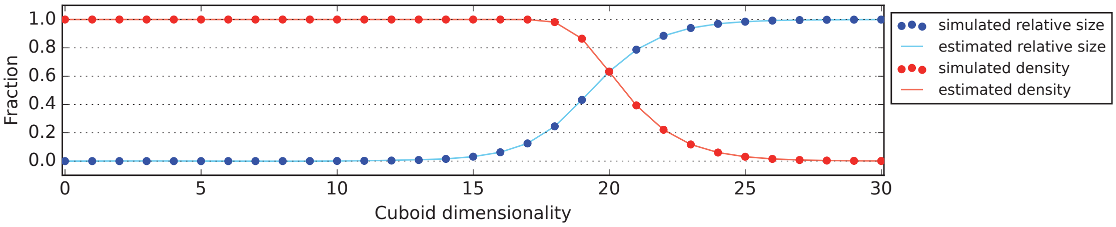


Figure 3.2: Relative size and density of a random k-dimensional projection of an n-dimensional cuboid with $m = 2$, $n = 64$, and $d = 20$

Plugging in $k = d$ in the equation above, we get that a single random d-dimensional projection of an n-dimensional base cuboid with m^d non-zero entries is expected to have a size that is $1 - e^{-1} \approx 0.63$ times the size of the base cuboid. For $k > d$, the ratio is very close to 1.

We also experimentally verified this claim. Figure 3.1 shows the result after running extensive simulations for $m = 2$, projecting a randomly generated n-dimensional base cuboid with 2^d non-zero entries to a random set of d dimensions. The graph plots the relative size of the projection compared to that of the base cuboid for different values of n and d. For a fixed value of d, we observe that the density is 1.0 when n equals d and approaches 0.63 when n becomes much larger.

The expected size of a random k-dimensional projection derived above also matches the results we obtain in experiments as shown in Figure 3.2. In the figure, the density is close to 1 for $k \ll d$, it is 0.63 for $k = d$ then quickly drops to 0 as $k \gg d$. The figure also shows that the relative size of the projection compared to the base cuboid is also close to 1 for $k > d$.

In other words, big projections of a sparse high-dimensional base cuboid are expected to have the same size as the base cuboid, with no significant reduction in size due to the projection. But, there are $\binom{n}{k}$ such k-dimensional cuboids, and storing all of them is not possible when n is very large. Of course, such a random data cube does not model all practical scenarios well, but one has to assume an extraordinary scenario (such as the pervasive presence of functional dependencies between dimensions) for it to fare much better than the random case. So fully materialized data cubes, whether sparse or dense, with large n really cannot be built.

3.4 Related Work

The problem of using materialized views to answer queries efficiently has been extensively studied [43], [70], [97]. However, in this thesis, we focus on a narrow class of queries that do not have joins or multiple relations as in the general setting that prior work looks into. Without joins or multiple relations, figuring which views can answer which queries is straightforward – for any given query, any view that aggregates values grouped by a superset of its group-by dimensions and filters based on a subset of its filter conditions can be used to answer it. We will, therefore, not go into a detailed discussion of the related work in the general setting, and focus only on techniques for data cubes and similar approaches. Much research has gone into optimizing data cubes due to their importance in analytical processing and business intelligence. An overview of the research can be found in [20] and [79].

3.4.1 Fully Materialized Data Cubes

There has been prior work on efficient algorithms to speed up the construction of the entire cuboid lattice. The general idea is to generate a spanning tree rooted at the base cuboid from the cuboid lattice and construct the data cube top-down starting from the root. All cuboids along some path in the spanning tree share computation to reduce the overall costs of building the data cube. [6] introduces two algorithms, *PipeSort* and *PipeHash*, that smartly share sorting and partitioning costs, respectively, across multiple cuboids. *Overlap* [6] is very similar to PipeSort and tries to overlap as much sorting as possible while building multiple cuboids. *PartitionCube* [87] is a divide-and-conquer algorithm that recursively partitions the data cube on one of the dimensions until it fits entirely in memory and uses a variant to the *PipeSort* algorithm that minimizes the number of sorting passes required. *MultiWay Array Cube* [110] partitions the base cuboid into multidimensional array *chunks* and aggregates dimensions away in an order that ensures the minimum number of chunks to be kept in memory. While all these approaches do reduce the cost of building the data cube, they don't scale to data cubes with hundreds of dimensions, and the storage costs alone would be prohibitive as described in Section 3.3.

3.4.2 Partially Materialized Data Cubes

The expensive storage and computing costs associated with fully materialized data cubes have led researchers to work on partially materialized data cubes. In these data cubes, only a subset of the cuboids is precomputed and materialized following some materialization strategy. Since all cuboids are not materialized, if a query is mapped to a cuboid that is not materialized, it has to be computed at run time from the smallest subsuming cuboid that contains all the dimensions of the query. The Greedy Algorithm [45] aims to minimize the average time taken to evaluate a lattice node on the fly while maintaining a limited number of materialized nodes. The algorithm iteratively selects nodes to be materialized based on their benefit relative to the current set of materialized nodes. The benefit is calculated as the improvement materializing a node offers in the cost of computing itself and its descendants. There exist variants that consider the absolute benefit as well as the benefit per unit space. PickBySize [91] proposes a much simpler heuristic that starts with the base cuboid and iteratively adds the smallest remaining cuboid until some space constraint is reached. The Greedy-Interchange [41] starts with the solution of the Greedy Algorithm and iteratively exchanges selected and non-selected cuboids to improve the total benefit. MDred-lattice [10] focuses on optimizing the average response time for a specific query workload rather than the overall average query, which significantly reduces the solution space. Some commercial data warehouse systems materialize all small dimensionality cuboids (cube shell) [72]. Most of these algorithms cannot be scaled to deal with exponentially large numbers of cuboids present in a high-dimensional data cube. Given a storage budget, these algorithms are also likely to prefer low-dimensional cuboids due to their lower storage costs. Due to the curse of dimensionality, except for extremely low-dimensional queries, the smallest subsuming materialized cuboid is most likely the base cuboid for most queries, making the idea of precomputing the projections moot.

3.4.3 Iceberg Cubes

Instead of selecting which cuboids are materialized, other approaches save storage space by pruning computations that they deem insignificant. Iceberg cubes focus on storing only those cells that meet certain criteria post-aggregation as in `HAVING` clauses, such as having a minimum support or its measure value meeting a specified threshold. BUC [16] proposes a bottom-up approach for building Iceberg cubes, where the cells of low-dimensional cuboids are ancestors of the cells of high-dimensional cuboids. Under this strategy, if a cell does not meet the iceberg condition, all of its descendants are pruned, and their values are not computed. It relies on the *anti-monotonic* property of aggregates such as sum and count that guarantees that the value of an aggregate function for a larger subset of data will never be smaller than the value for a smaller subset of the same data. [44] extends the idea to cover average, which does not satisfy the anti-monotonic property, by using a weaker condition called top-k average that does satisfy it while also proposing H-tree, a hypertree data structure for sharing computation. Star-Cubing [107] extends the data structure further to combine iceberg pruning with simultaneous aggregations of MultiWay Array Cube [110]. Despite all these algorithms, due to the exponential growth of the number of cells with an increase in the number of dimensions, it is unrealistic to compute even an iceberg cube for high-dimensional data. Iceberg cubes also suffer some drawbacks, such as having to determine a threshold appropriate for the use case and being sensitive to data skew. For these reasons, we don't explore them further in this thesis.

3.4.4 Compressed Cubes

Researchers have also focused on identifying redundancies in data cubes and using special data structures to store data cubes compactly by removing these redundancies without any loss of information. Condensed cubes [103] identify special tuples called Base Single Tuples

(BST) that are the only tuples in some partition of the base cuboid over any subset of the dimensions. This tuple is the only one that contributes to some cells in several cuboids. All these cells have the same value, and can therefore be condensed together to save space. Quotient Cubes [66] extends this idea by taking cells from multiple cuboids whose values are aggregations of the same set of base cuboid cell values and assigning them into an equivalence class of cells with identical aggregate values. This is similar to the suffix coalescing used in Dwarf Cubes [95]. Cure [78] extends the idea to also cover hierarchical dimensions and applies further compression to save space. While these frameworks successfully compute and store the full cube lattice at a fraction of its total unoptimized size, their size grows polynomially with respect to dimensionality [96] and may require a storage space several orders of magnitude larger than the base cuboid [29]. This makes them impractical for large datasets.

3.4.5 Inverted Indexes

There has also been prior work on high-dimensional data cubes. Frag-Cubing [72] proposes partitioning the dimensions into small sets called fragments and fully materializing all cuboids of every fragment. A query with dimensions from multiple fragments is evaluated using joins on either inverted indices built on the cuboids of each fragment. These inverted indexes register a list of tuple ids associated with each attribute value in every dimension. Compressed Bitmap Index Based Method [69] improves the storage cost by using compressed bitmap indexes instead of inverted indexes that store tuple ids using integers. qCube [94] discusses how to use inverted indexes to answer range queries on data cubes. bCubing [93] proposes a hybrid memory solution employing two-level indexes for big data cubes. The first level index stores in RAM, for every attribute value, block ids and the number of tuples where that attribute value occurs. The second level index stores information about individual tuple ids for each block that contain some attribute value as well as the measure values associated with each tuple in external storage. 3iCubing [30] proposes using interval inverted indexes that represent back-to-back tuple ids in an inverted index using intervals to reduce memory space. All of these approaches have to join and aggregate tuples on-the-fly for dimensions spanning multiple fragments and cannot support interactive-time query results for large datasets. Commercial data warehouse solutions such as Vertica [67] and Amazon Redshift [40] that rely on indexes also have the same drawback.

3.4.6 Sampling

Sampling techniques have been well studied in the database community [4], [21], [50], [83], [102]. There are two main classes of techniques that use sampling to answer queries – on-the-fly samples and precomputed samples.

Online aggregation uses on-the-fly samples to approximate answers to database queries quickly and efficiently. This technique was pioneered by [48] and [42], where samples are continuously drawn from the database, refining the results as more samples are retrieved.

This enables users to view preliminary results swiftly and decide whether to proceed with the query, depending on the precision of these initial outcomes.

However, the implementation of online aggregation requires special operator designs to facilitate progressive execution [42], [54], [55], [71], [74]. A notable example is the Ripple Join operator proposed by [42], designed specifically for the progressive computation of joins. However, this algorithm demands inputs to the join operation to reside in memory for peak performance, which may pose limitations when dealing with substantial datasets. Addressing this issue, [56] introduced the Sort-Merge-Shrink (SMS) join. This approach divides the overall join operation into a union of numerous Ripple Joins, each of which handles a portion of the input guaranteed to fit within memory. This strategy makes join operations more scalable and manageable. Building upon this, the Database Online (DBO) system proposed by [54] further enhances join operator efficiency, demonstrating how to effectively utilize indices on input data to achieve this improvement.

There have also been substantial advancements in extending online aggregation to distributed and parallel environments [84]–[86], [105], [108]. [106] suggests a strategy to stream samples that can be concurrently used by multiple queries, potentially reducing the amount of data processed for each query and increasing overall system efficiency.

The uniform sampling used in online aggregation comes with several challenges when there is data skew or there are too few entries in certain groups. The second class of systems that use precomputed samples for Approximate Query Processing (AQP) employs stratified sampling to mitigate these issues.

Some systems assign strata purely based on the schema and statistics of the table. Approximate Query Answering (AQUA) [2], [3] is one such system that considers all possible combinations of grouping columns and adopts a different sampling fraction for each combination. However, as the number of combinations increases exponentially with the number of columns, this approach does not scale well to high-dimensional data with many potential grouping columns. In a different vein, [7] propose small group sampling, a stratified sampling technique that constructs both a global uniform sample and separate tables for single grouping columns containing only a few rows. Coupled with outlier indexes [18], this method is capable of handling skewed data distribution. However, it does not include samples to cover cases where small groups may arise when more than one grouping column is used in the query.

An alternative line of research depends on historical workloads to determine sampling weights, premised on the assumption that future workloads will resemble past ones. Ganti et al. [36] apply this concept by sampling tuples with weights proportional to the number of queries in the workload that include it in their results. Similarly, STRAT [19] selects tuples in a manner that minimizes the relative error of the expected query workload. SciBORQ [92] also employs a similar strategy, using special structures called impressions where tuples are selected based on past query results. BlinkDB [5] expands on the idea proposed by [7] to address small groups from multiple columns that frequently appear in the workload. However, this strategy is limited by storage constraints, which may restrict the number of such groups of columns that can be efficiently supported, hampering the system in a high-dimensional setting.

While stratified sampling techniques can significantly improve the quality of approximate query answers in the face of skewed data and outliers, further research is needed to develop techniques that can handle high-dimensional data and adapt to changes in the workload.

3.4.7 Synopses

Several AQP systems use synopses to have interactive response times while querying big data. These synopses capture statistical properties of the data, usually with loss of information, while occupying much less space. By running queries on these much smaller synopses instead of the actual data, AQP systems trade accuracy for speed. We covered sampling-based synopses in the previous section and focus on other types of synopses in this section.

[11] converts values of 2-D cuboids into probability matrices and computes linear regression models that compute any entry of these matrices. This idea is further refined in [12], where they model dense regions of the base cuboid using log-linear models. A similar approach is suggested in [90], where a Gaussian kernel that explains the data distribution is obtained. Alternatively, [100] proposes approximating cuboids by applying a wavelet transformation on the logarithm of partial sums of values. These approaches require a large number of summary information to approximate high-dimensional data accurately. Due to the sparsity of high-dimensional data, the space required for storing the summary information may far exceed the space needed for storing the base cuboid.

The advantages of high-dimensional data cubes and the challenges faced by current approaches in supporting interactive-time querying on high-dimensional data led me to explore new approaches for selecting the cuboids to be materialized, how these cuboids are stored as part of the data cube and how they are used in answering queries.

4 Sudokube System

Addressing the challenges of high-dimensional data cubes requires a solution that strikes a balance between pre-computation storage requirements and swift query response times. With this in mind, this chapter presents Sudokube, a novel data cube system that permits fast querying of high-dimensional data and provides a broad spectrum of functionalities for Online Analytical Processing (OLAP) tasks.

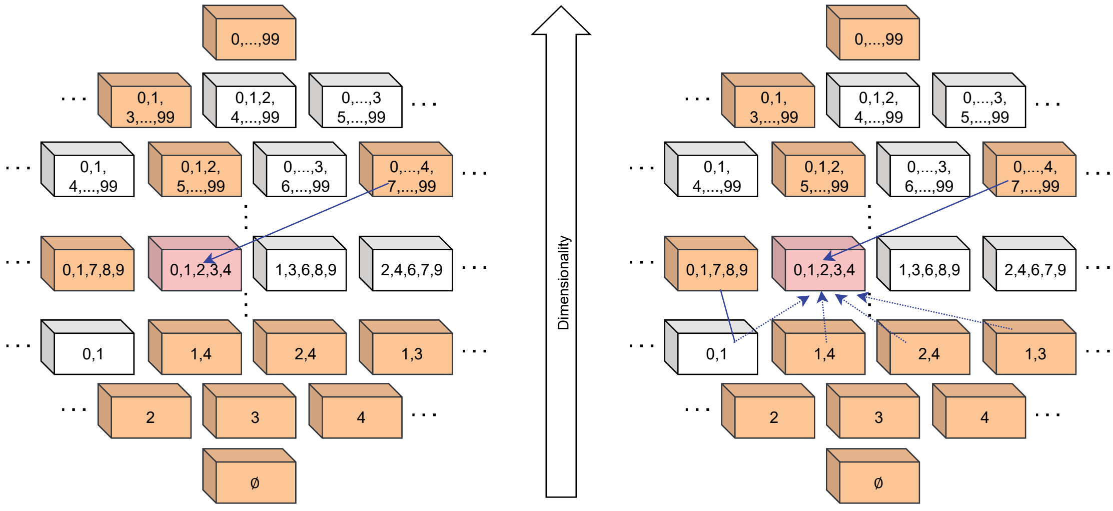
Sudokube judiciously selects a subset of cuboids for materialization, keeping storage costs low. Utilizing these materialized cuboids, it swiftly approximates queries under certain statistical assumptions. Detailed techniques for efficient approximation and their performance assessment in terms of speed and accuracy will be discussed in Chapter 6.

In this chapter, we will discuss the key concepts and the overall architecture of Sudokube. This discussion provides a foundation for understanding Sudokube's innovative approach to high-dimensional data cube processing.

4.1 Sudokube Ideas

4.1.1 Materialization and Querying

The number of possible cuboids in a high-dimensional data cube is astronomical, and we have already discussed the infeasibility of materializing the entire cuboid lattice for these data cubes. Sudokube takes an approach following well-established methods [10], [41], [45], [91] that only materialize a subset of the cuboids that form the data cube. However, the fraction of the cuboids that can be materialized while keeping the storage costs feasible is extremely small, given the high dimensionality. The classical approach [45] for computing some query cuboid by projecting the smallest materialized cuboid that subsumes it cannot guarantee interactive time for answering the query. In most cases, this approach ends up projecting the base cuboid or some high-dimensional cuboid containing nearly all the dimensions, which can take a long time.



Project Smallest Subsuming Cuboid

Approximate Using Projections of Query

Figure 4.1: Comparison of the classical idea of projecting smallest subsuming cuboid to answer query vs. Sudokube idea of approximating queries using all of its available projections

Sudokube takes an alternative approach by which, the materialized cuboids are used to efficiently *approximate or reconstruct* missing query cuboids that were not selected for materialization. While a query cuboid may not be computed precisely from its projections, the projections still hold information that can be harnessed to approximately reconstruct it. The system operates under the assumption that queries are relatively low-dimensional as their results need to be displayed and comprehended by humans. Consequently, their projections are also few and low-dimensional. The approximation of a query from its projections can be accomplished interactively through an online approach [48], processing all available query projections in ascending order of dimensionality. Sudokube quickly generates approximate results after processing low-dimensional projections, and refines the results as more projections are processed. Eventually, the exact result is obtained after processing a cuboid containing all query dimensions.

Example 4. *Imagine a data cube with* 100 *dimensions labeled* 0 *to* 99*. As explained in Section 3.3, fully materializing all the cuboids in the lattice is not feasible, so only a subset of them can be materialized. Let us assume that during cube construction, only the orange-colored cuboids shown in Figure 4.1 were chosen for precomputation based on a particular materialization strategy. Now, consider a query that corresponds to the cuboid containing dimensions* $\{0,\ldots,4\}$ *which is not materialized. Classical data cube approaches would answer the query precisely by projecting the smallest subsuming cuboid containing all query dimensions, in this case, the* 98*-D cuboid with all dimensions except* 5 *and* 6*. Projecting this cuboid to answer the query* $0,\ldots,4$ *takes nearly as much time as projecting the base cuboid with all* 100 *dimensions.*

Sudokube can use any of the 32 *projections of the cuboid* $\{0,\ldots,4\}$ *to approximate it to some degree. However, not all of them are used. The materialized* 2*-D cuboid* $\{2,4\}$ *contains more*

Item	b_5	b_4
Apple	0	0
Banana	0	1
Coffee	1	0
Tea	1	1

Quarter	b_3	b_2
Q1	0	0
Q2	0	1
Q3	1	0
Q4	1	1

City	b_1	b_0
Geneva	0	0
Lausanne	0	1
Bern	1	0
Zurich	1	1

Figure 4.2: Encoding Item, Quarter and City keys using two binary dimensions each.

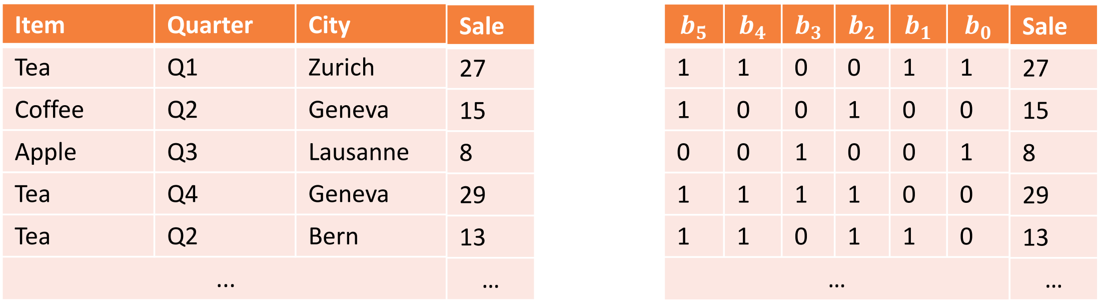
information than both the 1*-D cuboids* $\{2\}$ *and* $\{4\}$ *combined, while also being low-dimensional enough for quick processing. Therefore, Sudokube processes the* $\{2,4\}$ *cuboid but not* $\{2\}$ *or* $\{4\}$ *individually. Similarly, it uses the information from the* 2*-D cuboids* $\{1,4\}$ *and* $\{1,3\}$ *to approximate the cuboid* $\{0,\ldots,4\}$*. All query projections need not have been precomputed either. The cuboid* $\{0,1\}$*, though not materialized, can be obtained by projecting the* $\{0,1,7,8,9\}$ *cuboid and used to approximate the query* $\{0,\ldots,4\}$*. Finally, after processing the cuboid* $\{0,\ldots,4,7,\ldots,99\}$*, the query projection itself is processed, and the exact answer is obtained.*

This approach is best suited for scenarios where the number of dimensions requested in a single query is small, but not too small. The number of dimensions has to be low enough so that a human user is capable of interpreting the displayed result [72]. The low-dimensionality of the query has an added benefit – it guarantees that the projections are even more low-dimensional, and therefore, cheaper to process and have a greater likelihood of being chosen for materialization.

4.1.2 Binary Cuboids

Sudokube exclusively operates with *binary data cubes* consisting of binary dimensions with domains containing only 0 and 1. While real-world data does not solely comprise such binary dimensions, this model does not limit applicability. Any classical dimension with a domain of m values can be encoded using $\lceil \log_2 m \rceil$ bits, and Sudokube converts each of those bits into binary dimensions. To maintain data semantics and create the appearance of non-binary domains, Sudokube groups these $\lceil \log_2 m \rceil$ binary dimensions into a *cosmetic dimension.* Users interact with cosmetic dimensions, and the binary encoding remains transparent to them. Throughout this thesis, we will use b_i to denote the value of the binary dimension labeled i.

Example 5. *Consider the sales data from Example 3. Each of the Time, Product, and Location dimensions have* 4 *keys in their domain, which can be encoded using* 2 *bits each. Figure 4.2 shows the relationship between* 6 *binary dimensions* $0\ldots5$ *and the cosmetic dimensions obtained by grouping them. Figure 4.3b shows a part of the binary base cuboid containing these binary dimensions constructed from the sales fact table, which is shown in Figure 4.3a.*



Item	Quarter	City	Sale
Tea	Q1	Zurich	27
Coffee	Q2	Geneva	15
Apple	Q3	Lausanne	8
Tea	Q4	Geneva	29
Tea	Q2	Bern	13
…			…

(a) Fact table

b_5	b_4	b_3	b_2	b_1	b_0	Sale
1	1	0	0	1	1	27
1	0	0	1	0	0	15
0	0	1	0	0	1	8
1	1	1	1	0	0	29
1	1	0	1	1	0	13
…						…

(b) Binary base cuboid

Figure 4.3: Encoding of sales data using binary dimensions in Sudokube

Utilizing binary data cubes in high-dimensional settings provides several advantages that Sudokube capitalizes on. First, the mathematics and algorithms for several operations such as selection, projection, storage, and approximation of cuboids become much cleaner and simpler when every dimension is binary. Second, the finer granularity of dimensions in a binary data cube enables encoding structure and hierarchy within the relationships among dimensions themselves, eliminating the need for star or snowflake schemas and joins.

Example 5 (continued)**.** *In the binary data cube for the sales data, the binary dimensions* 5 *and* 4 *together encode the item. However, the binary dimension* 5 *by itself encodes the product category.* $b_5 = 0$ *indicates fruits and* $b_5 = 1$ *indicates beverages. Similarly, binary dimensions* 3 *and* 1 *encode half-years and regions, respectively. If a query asks to break down sales by product category instead of individual items, there is no need to do joins with dimension tables to coarsen the dimension granularity. This query is mapped to some binary cuboid that contains only dimension* 5 *instead of both* 5 *and* 4 *as would have been the case if the query asked to group by individual items. This cuboid may have been materialized and the result of the query directly available; otherwise, Sudokube uses its reconstruction techniques to answer it.*

Let n be the number of such binary dimensions in the data cube. Then, there are 2^n cells in the base cuboid corresponding to every combination of keys from these binary dimensions. Each cell is identified by some mapping $x \in \{0,1\}^{[n]}$, concisely represented using a vector $\boldsymbol{x}$ as described in Section 2.1. We denote the measure value associated with the cell $\boldsymbol{x}$ in the base cuboid by $C(\boldsymbol{x})$. In general, for some $I \subseteq [n]$ and $\boldsymbol{y} \in \{0,1\}^I$, we denote the measure value associated with the cell $\boldsymbol{y}$ in the projection of the base cuboid to I as $C_I(\boldsymbol{y})$. The cuboid C_I can be obtained by projecting the base cuboid or any other cuboid C_J with $I \subset J \subseteq [n]$. The computation of the projection for the sum aggregation is given by

$$C_I(\boldsymbol{y}) = \sum_{\substack{\boldsymbol{z} \in \{0,1\}^J \\ \boldsymbol{z}_{\downarrow I} = \boldsymbol{y}}} C_J(\boldsymbol{z}). \tag{4.1}$$

This formula is similar to that for the marginalization of probability distributions described in Equation (2.3). If all the measure values in a data cube are non-negative, then the binary

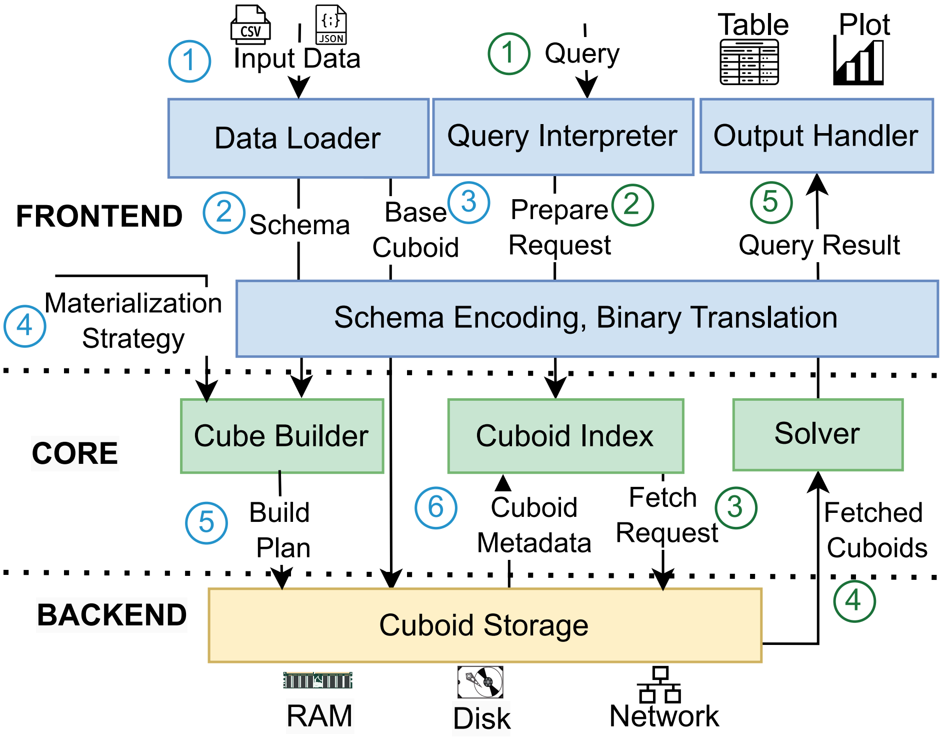


Figure 4.4: Architecture of the Sudokube system and the workflows for building (①-⑥) and querying (①-⑤) data cubes in the Sudokube system

base cuboid can be considered an unnormalized joint probability distribution, and its cuboids could be the marginal distributions. Let *total* denote the total sum of the measure values for every cell in the base cuboid. Then, for any $I \subseteq [n]$ and $\boldsymbol{x} \in \{0,1\}^I$, the marginal probability distribution and the cuboid corresponding to dimensions in I are related to each other by

$$p_I(\boldsymbol{x}) = \frac{C_I(\boldsymbol{x})}{total} \tag{4.2}$$

We will assume throughout this thesis that the measure values are non-negative. If that is not the case, the data cube can be split into two data cubes, one containing all the positive measures and the other containing (the absolute value of) all the negative measures. Any query on the original data cube can be calculated as the difference between the results of the same query on these data cubes.

4.2 System Architecture

Sudokube comprises three components – *frontend, core query execution engine*, and *backend.* The frontend offers a basic user interface for data loading, querying, and displaying the results. It also provides schema support and handles the binary encoding of keys. The rest of the

Sudokube system sees only the binary dimensions representing the individual bits of these keys. The core engine decides what cuboids to materialize during cube construction time and what cuboids to fetch and process during query time. It allows users to choose from several solvers to extrapolate query results from the fetched cuboids. Finally, the backend is responsible for storing, projecting, and retrieving materialized cuboids.

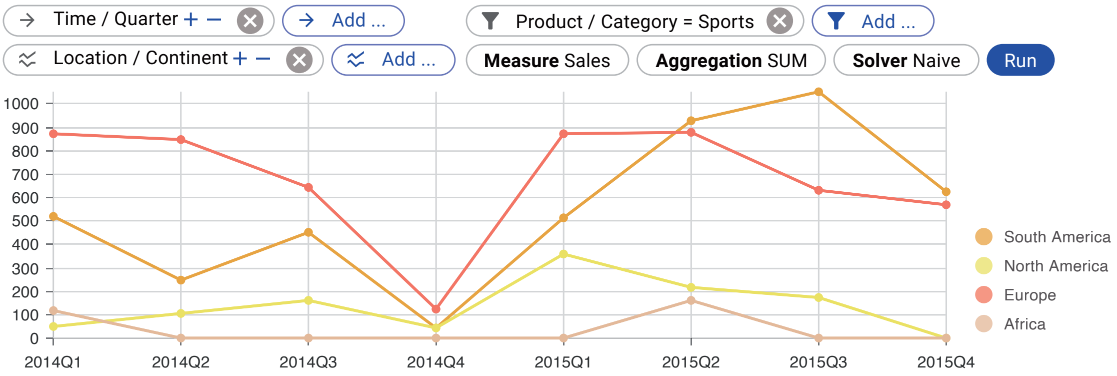
The workflow for using Sudokube is shown in Figure 4.4. First, a preconfigured loader reads data ① from a source and produces a schema ② that encodes the data to form the (binary) base cuboid ③, which is then stored in the backend. Next, the cube builder selects which cuboids will be materialized based on a given materialization strategy ④. The backend is then provided with a build plan ⑤ that describes which cuboids need to be materialized by projecting other cuboids. After constructing the data cube, the core engine indexes references to the cuboids ⑥ returned by the backend.

When the user submits a query ①, the frontend converts it into a query on the binary dimensions and forwards it to the core engine. The core then queries the cuboid index ② to find the materialized cuboids relevant to the query. After this, the core instructs the backend to fetch (possibly projections of) those cuboids ③. The fetched cuboids are fed into the solvers ④, which use them to extrapolate the query results ⑤. Finally, the output handler displays the result in the requested format.

4.2.1 Cube Specification and Querying

Sudokube supports all the fundamental data cube operations. Before a cube is constructed, Sudokube requires all *measures* of interest to be specified so they can be precomputed. Users can designate individual columns as measures or specify functions that produce measure values from multiple columns. After the cube has been built, users can query one or more of these measures. Sudokube allows users to specify a variety of aggregation operations such as sum, count, average, variance, correlation, and linear regression coefficient. Users can *pivot* dimensions across both the horizontal and vertical axes. In addition to traditional dimensional hierarchies, Sudokube allows fine-grained virtual hierarchies for each dimension where the consecutive values are grouped together in sizes of powers of two. For example, users can specify a hierarchy for the time dimensions such as Year - Month - Day, and Sudokube additionally offers virtual dimensions such as Year/4 or Day/2 where four consecutive years or two consecutive days are grouped together, respectively. Users can then *drill down* on the result by either going down one level on the hierarchy for some dimension or adding a new one to some axis. Conversely, the user can *roll up* going up the hierarchy for some dimension or removing one. Finally, users can *slice* and *dice* on multiple dimensions to filter the keys in the result. We provide users with a visual interface (inspired by existing visualization tools such as [77]) as shown in Figure 4.5 for easily specifying such queries.

Sudokube contains a library that implements other operations through post-processing. These operations include window-based aggregations, user-defined grouping of values, and defining



Time / Quarter

Location / Continent

Product / Category = Sports

Measure Sales

Aggregation SUM

Solver Naive

Figure 4.5: The Sudokube user interface for querying. Users specify dimensions on the horizontal axis as well as for series and apply filters. They also choose the measure, the aggregation function, and the solver for answering queries.

views that transform data cubes to add, remove, or modify dimensions in some way. High-level operations for data exploration that wrap several basic operations are also included. For example, users can load arbitrary semi-structured data into a data cube and analyze possible schema changes or functional dependencies. We will go into more detail concerning all the types of queries and aggregation functions later in Chapter 5.

4.2.2 Frontend

The Sudokube frontend interacts with users through a graphical interface. It handles data loading, query interpretation, and output, and converts data between human-readable and binary values. Sudokube supports two types of schema — static and dynamic. In a static schema, the columns in the input data are known beforehand, and the functions mapping them to dimensions in the data cube and all hierarchical structures are programmed into the data loader. However, in a dynamic schema, the schema need not be known (or even fixed) before data loading. New bits are assigned automatically whenever Sudokube discovers a new column or when an existing column requires a larger domain. The downside of dynamic schema is that the bits for a dimension need not be next to each other, which slows down the binary encoding process. Sudokube can load data from CSV or other fixed format files using a static schema and data from JSON files using a dynamic schema.

Sudokube uses multiple encoders to encode values to binary. The dictionary encoder encodes a value using its position in a dictionary. For a static schema, the values are sorted and added to the dictionary before data loading, whereas, in a dynamic schema, Sudokube adds them when discovered during data loading. Sudokube encodes integer and fixed-point numbers using their offset from the minimum value in the domain for static schemas, but encodes them using their absolute values and sign bits for dynamic schemas. More details concerning data loading and binary encoding are described in Chapter 5.

4.2.3 Backend

The Sudokube backend uses multiple formats for storing cuboids – *dense*, *sparse row*, and *sparse column*. A cuboid stored in *dense format* is a collection of multi-dimensional arrays, one for each measure. In each array, the position encodes the values of the binary keys, and the entry at that position is the associated measure. This format does not require additional space to store keys and is mainly used to store low-dimensional cuboids where most keys have associated measure values. However, it is infeasible for high-dimensional cuboids where the support (the number of keys with a non-zero measure value) is small compared to the domain size. In such cases, Sudokube opts to use the *sparse format* where both the key and the measure values are stored, but only for the keys where at least one associated measure value is non-zero. Sudokube always stores base cuboids in the sparse format. There are two variants of the sparse format as well. In the *sparse row format*, a cuboid is stored as a collection of records containing a binary key and the associated measures. This format is used as an intermediate representation for the base cuboid during data loading. Once data loading is completed, Sudokube converts the base cuboid to the *sparse column format*. In this format, a cuboid is a collection of arrays, one array for each key bit and each measure. The sparse column is better suited for materialization as the projection operation is faster and more efficient for this format compared to the sparse row format. While projecting a materialized cuboid during querying, only the relevant dimensions need to be processed, improving cache efficiency. Sudokube deploys different techniques for duplicate elimination during projection depending on the projection size. If the projection is sufficiently low-dimensional and fits in memory, hashing is used for duplicate elimination; otherwise, sorting is used. We will cover all of this in more detail in Chapter 8.

4.2.4 Materialization Strategy

Given a base cuboid containing hundreds of binary dimensions, precomputing and materializing all its projections is infeasible in terms of time and space. Sudokube, therefore, employs multiple *random partial materialization* strategies that randomly select which projections are materialized. Each of these strategies selects cuboids of different dimensionality following some probability distribution. For a given dimensionality distribution, Sudokube supports two ways to specify how the binary dimensions are selected for a cuboid of some particular dimensionality. The binary dimensions could be chosen either uniformly at random or by picking prefixes of binary dimensions that encode some cosmetic dimension. The latter approach yields better results as the selected cuboids closely match queries involving hierarchical dimensions. Moreover, Sudokube has heuristics to predict the storage cost of these strategies and can suggest a preferred strategy for a given storage budget. We will analyze different materialization strategies and their utility for different solving techniques in Chapter 7.

4.2.5 Query Approximation

While executing a query, Sudokube goes through three phases — *prepare*, *fetch*, and *solve*. During the prepare phase, Sudokube processes the cube meta-data to produce a fetch plan. The cuboids relevant to a query are found by grouping the cuboids by their intersection with the query and choosing the cheapest cuboid in each group. We use a cuboid's original dimensionality (before the intersection with the query) as a heuristic for its cost. Finally, any cuboid that contains only a subset of dimensions from another relevant cuboid is eliminated. At the end of the prepare phase, the core engine produces a fetch plan that lists the remaining cuboids and what projection needs to be obtained from each.

During the fetch phase, the specified cuboids are projected by the backend and fetched as described by the plan. Finally, the fetched cuboids are fed into the *solver* during the solve phase. Depending on their needs, users can choose from several solvers offered by Sudokube. First, the *naive solver* gives the exact result for any query by projecting the smallest materialized cuboid that subsumes it. However, in practice, this subsuming cuboid is almost always the base cuboid for which projection may take a long time. Next, we have the *linear programming solver* that constructs linear equations [49] on query result variables from the fetched cuboids and outputs lower and upper bounds for each variable. While these bounds are guaranteed to be correct, they can be quite lax, and their computation does not scale well to higher-dimensional queries. Finally, we have two *approximate* query solvers. The *moment solver* [13] extracts (stochastic) moments [99] that capture inter-dependencies between the query dimensions from the fetched cuboids using a process similar to the Fourier transform. It then extrapolates them by assuming uncorrelatedness for query dimensions with unknown interaction. Additional heuristics are employed to counteract cases where the assumption is infeasible. This solver yields query results very quickly but is less accurate, particularly for high-dimensional queries. The *graphical model solver* uses iterative proportional fitting [28], [98] to find the maximum entropy query result subject to the constraints imposed by the fetched cuboids. It starts with a uniform distribution of data and iteratively scales the data to fit the fetched cuboids until convergence. This yields more accurate results but takes more time than the moment solver.

Example 6. *Consider a query that sums the measure values grouped by three binary dimensions. Each of the three 2-D projections of this query yields four linear constraints with sums of two measure values for fixed keys of the dimension pair. The linear programming solver finds the upper and lower bounds for each entry in the query result by maximizing and minimizing an objective function comprising that entry subject to the constraints. Alternatively, the three 2-D projections capture dependencies between any two dimensions, treated as random variables, in the form of moments [99] – three covariances, three means, and the total sum. The moment solver makes use of the fact that the query result can be exactly reconstructed from eight moments, out of which seven are known from the available projections. It fills in known seven moments and assumes that unknown moments are zero, such as the "generalized covariance" of all three dimensions in this example. The graphical model solver initially assigns a uniform value to all* *eight entries in the query result. These values are scaled up so that projecting the query result matches the given projections, successively, until convergence.*

Furthermore, Sudokube supports both *online* and *batch* modes for querying. In batch mode, the base cuboid is never fetched, and the solve phase starts after the fetch phase ends. The final (approximate) query result is then returned to the frontend for decoding and displaying to the user. In online mode, however, the fetch and the solve phases are concurrent, and the query result is updated and displayed periodically using a callback function as more and more cuboids are fetched. Chapter 6 discusses each of these solvers in more detail and evaluates their performance on multiple metrics such as speed and accuracy.

4.3 Experimental Setup

In the following chapters, we will experimentally demonstrate the performance of the Sudokube system on various aspects. These experiments are conducted on a prototype of the Sudokube system that we have implemented. This prototype operates on a single node with its backend implemented in C++. The other system components are developed in Scala. The C++ backend is accessed via the Java Native Interface, which allows data storage and processing to be handled outside of the Java Virtual Machine (JVM). We chose this hybrid design to exploit the computational efficiency of C++ while utilizing the ease of prototyping in Scala.

In the backend, multithreading is utilized to concurrently fetch and project multiple cuboids. However, cuboids are not sharded; a single thread always processes a given cuboid. To maintain a fair comparison in the experimental tests, all solvers refrain from fetching cuboids in parallel, even though many of them have the capacity to do so. This restriction was implemented because the naive solver operates on a single cuboid using a single thread.

Parallelism is, however, employed in data cube construction. The frontend and the core engine are executed on separate threads to facilitate the online query mode. Although the query engine is designed to execute different queries in parallel, this is not done in the experiments.

All data cubes utilized in the experiments are constructed offline and saved to disk. Before starting any experiment, any data cube used in that experiment is loaded into RAM. Accordingly, all experiments in this thesis measure the time to fetch cuboids from RAM, most of which is spent projecting the cuboid to the specified dimensions. All experiments are conducted on a server with 2 × 12-core Intel® Xeon® E5-2680 v3 (Haswell) CPUs, 30 MB cache, 256 GB DDR4-2133 RAM, and 200 GB SATA3 SSD.

4.3.1 Dataset Description

The experimental setup uses two datasets for evaluation — one real-world and one synthetic. The first dataset, referred to as NYC, contains actual data concerning parking violations issued in New York City between the years 2014 and 2021 [31]. This dataset comprises 43 columns that detail specifics about the vehicles involved, the violation, and so on. The dataset contains nearly 93 million rows, with data distributed fairly evenly across the eight-year span.

The second dataset used is the Star Schema Benchmark (SSB) dataset [82][88]. This dataset provides synthetic business-related data modeled using a star schema. The fact table, lineorder, includes various details about order items, such as quantity and price, while supplementary information is stored in the customer, part, supplier, and date dimension tables. Scale factor 100 is used to populate the tables, yielding 600 million rows of lineorder data.

Prior to loading into Sudokube, both datasets underwent minor preprocessing. The SSB dataset was flattened by joining the lineorder table with all dimension tables using the respective keys. Dimensions not contributing to meaningful aggregations were discarded, including those such as name, address, or customer_id. The majority of dimensions in both datasets are categorical and were encoded to binary dimensions using a dictionary encoder. Numerical dimensions like tax or revenue were directly encoded as fixed-width integers. Date and time columns were encoded by breaking them down into components, such as year or hour, each of which was then individually encoded as integers. This encoding strategy resulted in a total of 188 binary dimensions for the SSB dataset and 429 binary dimensions for the NYC dataset. Tables 4.1 and 4.2 show the original columns as well as the number of binary dimensions assigned to each of them by Sudokube for SSB and NYC datasets.

As the measure value to be aggregated, the contribution of each line item toward the total order price is used for the SSB dataset and for the NYC dataset, the number of rows for each combination of keys is used. Sudokube assigns one 8-byte word for storing each of these measure values. We use the sparse column format to store the base cuboids for these datasets. The rows are arranged in groups of 64 and each group is assigned one word per dimension for storing the keys plus 64 words for storing the measure values. Consequently, the base cuboid size is approximately 18.9 GB for the SSB dataset, calculated as $\left\lceil \frac{600\cdot 10^6}{64} \right\rceil \cdot (188+64)\cdot 8$ bytes and approximately 5.73 GB for the NYC dataset, calculated as $\left\lceil \frac{93\cdot 10^6}{64} \right\rceil \cdot (429+64)\cdot 8$ bytes.

For our experiments, we build several data cubes using Prefix and Random strategies on both datasets with a variety of values for the parameters N specifying the total number of materialized cuboids and $d_{\min}$ specifying the minimum dimensionality of materialized cuboids. The storage costs for various data cubes are summarized in Table 4.3. Note that the figure shows the additional storage costs for each data cube after excluding the cost for the base cuboid in gigabytes and as a fraction of the base cuboid size.

In most of our experiments, we select 100 queries of some specified dimensionality following the same strategy as the data cube they would run on.

Table 4.1: The schema for the SSB dataset. The original columns in the dataset are listed along with the number of binary dimensions that are assigned to encode it as a cosmetic dimension.

Column	#bits	Column	#bits	Column	#bits	Column	#bits
order_date	14	sup_cost	17	cust_nation	5	mfgr	2
ord_priority	2	container	6	cust_region	2	category	5
ship_priority	0	tax	4	cust_mkt_segment	2	brand	10
quantity	6	commit_date	14	supp_city	8	color	7
extended_price	24	ship_mode	3	supp_nation	5	type	8
discount	4	cust_city	8	supp_region	2	size	6
revenue	24						

Table 4.2: The schema for the NYC dataset. The original columns in the dataset are listed along with the number of binary dimensions that are assigned to encode it as a cosmetic dimension.

Column	#bits	Column	#bits	Column	#bits	Column	#bits
Plate ID	24	Issuer Precinct	10	Violation Time	12	House Number	17
Registration State	7	Issuer Code	17	Violation Code	7	Street Name	19
Plate Type	7	Issuer Command	14	Violation County	6	Intersecting Street	20
Vehicle Make	15	Issuer Squad	6	Violation Front/Opposite	4	Street Code1	13
Vehicle Expiration Date	24	Issuing Agency	5	Violation Legal Code	3	Street Code2	13
Vehicle Color	13	Issue Date	18	Violation Location	10	Street Code3	13
Unregistered Vehicle?	3	Date First Observed	17	Violation Precinct	10	Meter Number	17
Vehicle Year	12	Time First Observed	12	Law Section	4	Feet From Curb	6
Vehicle Body Type	13	From Hours In Effect	11	Sub Division	8	To Hours In Effect	11
Days Parking In Effect	8						

Table 4.3: Additional storage costs for various data cubes excluding the base cuboid cost.

N	$d_{\min}$	Storage (GB)	Fraction
2^{15}	6	0.091	0.016
2^{15}	10	1.01	0.176
2^{15}	14	8.833	1.542
2^{15}	18	57.198	9.982
2^{12}	18	7.222	1.26
2^{9}	18	0.857	0.15
2^{6}	18	0.08	0.014

(a) NYC Random

N	$d_{\min}$	Storage (GB)	Fraction
2^{15}	6	0.053	0.009
2^{15}	10	0.447	0.078
2^{15}	14	3.267	0.57
2^{15}	18	17.757	3.099
2^{12}	18	2.317	0.404
2^{9}	18	0.278	0.049
2^{6}	18	0.025	0.004

(b) NYC Prefix

N	$d_{\min}$	Storage (GB)	Fraction
2^{15}	6	0.126	0.007
2^{15}	10	2.013	0.107
2^{15}	14	31.345	1.658
2^{12}	14	3.221	0.17
2^{9}	14	0.302	0.016
2^{6}	14	0.025	0.001

(c) SSB Random

N	$d_{\min}$	Storage (GB)	Fraction
2^{15}	6	0.096	0.005
2^{15}	10	1.364	0.072
2^{15}	14	17.259	0.913
2^{12}	14	2.012	0.106
2^{9}	14	0.212	0.011
2^{6}	14	0.015	0.001

(d) SSB Prefix

5 Data Loading and Querying

The user-facing component of any system plays a pivotal role in its utility and acceptance. As the bridge between the user and the intricate computational processes within the system, the frontend has a critical mandate. In the context of Sudokube, the frontend affords an intuitive, powerful interface that empowers users to load data, specify queries, and interpret results.

This chapter explores the frontend of Sudokube in depth, elucidating its capabilities, architectural choices, and functionalities. We initiate the discussion with how users first interact with the system, specifically, the data loading process. The complexities of handling both static and dynamic schemas are considered, including the trade-offs associated with each. Data transformation and binary encoding mechanisms also form an integral part of this discussion. We illuminate the various encoders utilized within Sudokube and how they encode different types of data to binary dimensions, allowing the remainder of the system to operate with uniform, standardized data cubes comprising only binary dimensions.

Subsequently, we delve into the diverse types of queries Sudokube supports. This chapter explains how the frontend converts basic OLAP operations into queries on cuboids and how more complex operations are implemented via post-processing.

This exploration offers a comprehensive understanding of the frontend's integral role in Sudokube's functionality. It paves the way for further discussions on solution strategies and backend operations in subsequent chapters.

5.1 Data Loading

Before querying, data is first loaded into Sudokube. The Sudokube data loader accepts CSV or JSON files, dividing them into chunks to process each one in parallel for increased efficiency. This process results in the creation of the base cuboid and the corresponding schema.

The schema plays a crucial role in converting input tuples into keys for binary dimensions that form the data cube in Sudokube. It consists of a collection of encoders, each assigned to a specific column in the input data. Each encoder, in turn, is assigned a unique set of binary dimensions. These encoders map arbitrary values to integers and then permute the bits of the integer to form keys in their assigned binary dimensions. When combined, the binary keys from all encoders form a binary key for the tuple. In reverse operation, these encoders can decode a binary key back into a combination of keys in each column.

The collection of binary dimensions assigned to a column encoder forms a *cosmetic* dimension in Sudokube that can be queried with the semantics of the original column in the data. Furthermore, for every cosmetic dimension, we define prefixes, assuming an ordering of binary dimensions from least significant to most significant. Each prefix encodes a range of keys in the domain, and together all these prefixes constitute a virtual hierarchy of dimensions for querying. Consider a cosmetic dimension Year comprising three binary dimensions y_2, y_1, and y_0 in decreasing order of significance. This cosmetic dimension can encode up to 8 unique years. We examine two prefixes of this cosmetic dimension: $y_2 y_1$ and y_2. The prefix y_2 encodes two periods of four consecutive years each. Extending the prefix to $y_3 y_2$ gives us four periods of two consecutive years each. In essence, prefixes allow us to seamlessly transition between grouping all years together (using a prefix of zero binary dimensions) and grouping years individually (using all three binary dimensions). Users can adjust their queries to focus on broader or narrower time periods, depending on the granularity they need for data analysis. There are two types of schemas employed in Sudokube: static and dynamic.

5.1.1 Static Schema

A static schema presupposes that the schema is known and fixed ahead of time. The data loader is programmed with a template for the static schema for each data source, which is used to create new instances when needed. Encoders are organized hierarchically, represented as a Directed Acyclic Graph (DAG). In this graph, leaf nodes are encoders, and non-leaf nodes represent a combination of multiple encoders. Each non-leaf node can be tagged with either a $'+'$ or $'\times'$ attribute, indicating the semantics of how its children should be combined.

Nodes with $'+'$ attribute: The $'+'$ attribute represents a union of its children. It implies that, at most, one of its children can be selected in a query. In terms of the semantic hierarchy, this relationship is often used to represent different levels of the same dimension. For example, in Figure 5.1, there are two leaf nodes with encoders for City and State that are children of a node tagged with $'+'$ for $\mathsf{Location}$. This structure signifies that City and State are different granularities of the $\mathsf{Location}$ dimension and would not be selected simultaneously in a query.

Nodes with $'\times'$ attribute: The $'\times'$ attribute represents a product of its children. It signifies that any combination of its children can be selected in a query. These nodes are typically used to represent dimensions that are independent of each other. In Figure 5.1, $\mathsf{Product}$, $\mathsf{Location}$, and Time are children of a root node tagged with $'\times'$, suggesting that users may want to query any combination of these three dimensions.

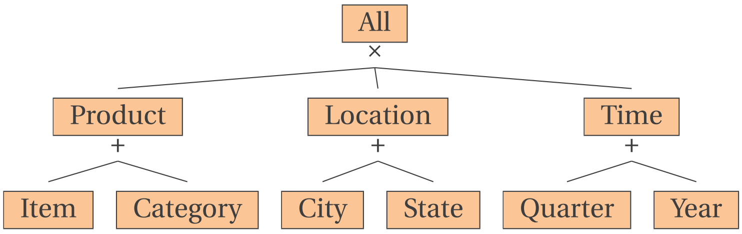


Figure 5.1: An example DAG of encoders for the sales data schema. The root node is a product of its children, suggesting that queries are to be constructed by taking any combination of queries of its children. The intermediate nodes are sums of their respective children, suggesting that queries are constructed by taking the union of queries of their children. The leaf nodes represent the individual encoders for each column.

This DAG-based hierarchy is not just a logical representation but is also used during querying and materialization. It allows the system to understand the relationships between various dimensions, aiding in the efficient choice of cuboids to materialize. It's important to note that the order of nodes in this hierarchy can impact the efficiency of Sudokube's query operations. Therefore careful consideration should be given to the design of this schema hierarchy.

Several types of encoders are implemented in Sudokube for a static schema. For instance, `DictionaryEncoder` uses the index of a value in a dictionary to represent it as an integer. The `IntegerOffsetEncoder` encodes integer values as offsets from a minimum value. The `DateEncoder` encodes dates as a combination of year, month, and day and timestamps as a combination of hours, minutes, and seconds. Custom transformations can be defined for each value, such as multiplying a number by 10^k to get k decimal digits before truncating it to an integer.

Once these encoders represent values using integers, the bits in the binary encoding of these integers are permuted to form keys of the binary dimensions. The permutation of integers to binary dimensions is significantly faster when the binary dimensions form a range. Hence, encoders for static schemas pre-allocate binary dimensions for each column. To achieve this, we pre-process the data to find unique values in each column and store these values in a file. These values are also sorted for columns encoded using `DictionaryEncoder`. The `IntegerOffsetEncoder` uses the unique value stored in the file to determine the minimum value for defining offsets. The loading process can be fully programmed, allowing for dropping columns, rearranging them, and defining custom columns. The users can designate any of the columns as the measure value or specify a function that computes it from the values of one or more columns.

5.1.2 Dynamic Schema

In contrast to a static schema, a dynamic schema is not programmed into the loader. The schema doesn't even need to be fixed throughout the data-loading process. In fact, every tuple could potentially have its own schema. Whenever a new column is encountered, a new encoder is assigned to it with a single binary dimension that encodes whether the key for this column is `NULL` or not. As more tuples are processed, encoders may request additional binary dimensions from a central coordinator to expand their domain. A DAG of nodes still represents this schema, but it is simply a tree with all the encoders as children of the root node.

Unlike static schemas, binary dimensions assigned to an encoder in a dynamic schema need not be adjacent to each other. While this results in slower encoding times, it eliminates the need for pre-processing. Without pre-processing, integers are encoded directly to binary using the `IntegerDirectEncoder`. An additional sign bit is requested when negative values are encountered. `DictionaryEncoder`, on the other hand, can be used with dynamic schema, but it adds entries to the dictionary in the order they appear first and not according to specified sort order.

Similar to the case of a static schema, users can specify user-defined functions to produce measure values from a tuple. In the absence of any such function, the constant 1 is used as the default measure value for all tuples in the data.

During data loading, the created schema instance encodes tuples to form pairs of binary keys and measure values. This collection of binary keys and measure values is sent to the backend for storage as the binary base cuboid. Dynamic schemas are serialized to disk after data loading so that they can be loaded back in the future for decoding the binary keys. Static schemas need not be serialized; a new one can be instantiated from the programmed template when required.

5.2 Building the Data Cube

After the binary base cuboid is stored in the backend, the next step is to select cuboids for materialization and build the data cube. Sudokube implements two strategies that randomly pick cuboids for materialization based on two parameters. The first parameter is the total number of cuboids to be materialized N; the second parameter is the minimum dimensionality for the materialized cuboids d_{min}. Algorithm 1 describes the procedure followed by Sudokube for selecting cuboids to materialize. It picks a total of N cuboids for materialization randomly, starting with $N/2$ cuboids of dimensionality d_{min}, $N/4$ cuboids of dimensionality $d_{\text{min}}+1$, and so on until the required number of cuboids are selected. The two strategies differ in how they select cuboids of a given dimensionality. The Random strategy picks binary dimensions uniformly at random without replacement, whereas the Prefix strategy picks only prefixes of cosmetic dimensions while selecting binary dimensions for the cuboids.

Once the list of cuboids to be materialized is finalized by Algorithm 1, a build plan must be created describing how these cuboids will be computed by projecting which other cuboids. Algorithm 2 describes the procedure to construct a simple but efficient plan to build cuboids from the smallest subsuming cuboid among the already materialized cuboids. In practice, our

Algorithm 1: Algorithm to select cuboids for materialization following the random or prefix strategy

```
def ApplyStrategy(strategy, schema, $N$, $d_{\min}$):
    $n \leftarrow$ number of binary dimensions in schema
    $N \leftarrow N/2$ ; $k \leftarrow d_{\min}$ ; $M \leftarrow \emptyset$
    while $N \geq 1$ do
        $N_0 \leftarrow \min\left(N, \binom{n}{k}\right)$
        while $N_0 > 0$ do
            if strategy is Prefix then
                $I \leftarrow$PrefixCuboid(root node of DAG in schema,$k$)
            else
                $I \leftarrow$ random subset of $[n]$ of size $k$
            if $I \notin M$ then
                $M \leftarrow M \cup \{I\}$; $N_0 \leftarrow N_0 - 1$
        $k \leftarrow k + 1$
        $N \leftarrow N/2$
    return $M$

def PrefixCuboid(node, $k$):
    if node is a leaf node then
        return prefix of length $k$ of cosmetic dimension in node
    else if node is intermediate node with $'+'$ attribute then
        child $\leftarrow$ random child of node with at least $k$ binary dimensions in subtree
        return PrefixCuboid(child, $k$)
    else
        Randomly partition $k$ into $k_1, \ldots, k_c$ for each child $child_1 \ldots child_c$ such that $k_i$ is not more than the number of binary dimensions in the subtree of $child_i$
        return $\bigcup_{i=1}^{c}$ PrefixCuboid($child_i$, $k_i$)
```

experiments show that the smallest subsuming cuboid always tends to be the base cuboid in high-dimensional scenarios, and we are better off using a plan that computes all cuboids to be materialized by projecting the base cuboid.

5.3 Querying

At its core, Sudokube is an analytical processing system that provides answers to complex queries across vast datasets. Users can leverage its querying capabilities to extract valuable insights once data is loaded, encoded, and stored in Sudokube. The system supports a wide variety of queries, making it flexible and robust for diverse analytical needs.

In the Sudokube query interface, users select a data cube to query, specifying dimensions to be shown on the horizontal and vertical axes. In a static schema, any level within the dimension

Algorithm 2: Algorithm that computes a simple plan to build cuboids by projecting the smallest subsuming cuboids materialized so far

```
input  : set $M$ containing dimensions $I$ of cuboids $C_I$ to be materialized, total number
         of dimensions $n$, base cuboid $C_{[n]}$
output : a build plan comprising sets of pairs of dimensions of input and output
         cuboids
def BuildPlan($M$, $n$, $C_{[n]}$):
    Sort $M$ in descending order to form list $L = \{I_1, \dots, I_N\}$
    Add dimensions $[n]$ as $I_0$ in the list $L$
    $P \leftarrow \emptyset$
    foreach $i \in 1 \dots N$ do
        $j \leftarrow$ largest index such that $I_j \supseteq I_i$
        $P \leftarrow P \cup (I_j, I_i)$
    return $P$
```

hierarchy can be selected for querying. Both static and dynamic schemas support querying on prefixes of cosmetic dimensions, which bridge various levels of a hierarchy and enable aggregations at intermediate granularities. Users can add multiple dimensions to either axis, and the order of dimensions specified directly influences the order of the displayed results. Filters can be applied to one or more dimensions at any level of the hierarchy. Applying filters to prefixes of cosmetic dimensions results in filtering ranges of keys. Users can select multiple values for any dimension level, with these combined using OR logic, while separate filters are combined using AND logic.

The query is completed by selecting a measure from the data cube to aggregate and by determining the function to aggregate measure values. The primary aggregation function is `SUM`, but other sum-based aggregations like `COUNT` and `AVG` are also supported. The user specifies the solver used to answer the query and decides whether it is run in batch or online mode.

The Sudokube query interpreter uses the data cube's schema to translate queries on cosmetic dimensions and their prefixes into queries on binary dimensions. The translated query includes binary dimensions for the horizontal and vertical axes and filters and binary encoding of user-selected filter keys. The query is then mapped to a cuboid in the data cube's lattice and relayed to the core engine. The core engine reconstructs the cuboid using the specified solver and returns the aggregate value in each cell, either as exact, approximate, or bounds, either once for batch mode or through a callback function for online mode.

5.4 Finding Materialized Cuboids Relevant to Queries

Given a query and a set of materialized cuboids, the first step in answering the query is to determine the set of relevant cuboids. We refer to this phase in processing the query as the *prepare* phase. Different techniques for answering queries differ in which cuboids they consider relevant for answering a query. We characterize the requirements of these solvers during the prepare phase using two parameters $d_{\max}$ and d_{cheap}. $d_{\max}$ sets an upper bound on the dimensionality of the cuboids Sudokube processes during the prepare phase. Any materialized cuboid with a dimensionality greater than the specified value of $d_{\max}$ is ignored by Sudokube while preparing for the given query. The other parameter d_{cheap} specifies the dimensionality below which Sudokube considers the cost of projecting cuboids insignificant.

There are two steps in the prepare phase as described in Algorithm 3. In the first step, the intersection of the query dimensions with the dimensions of each materialized cuboid is carried out. Multiple materialized cuboids may yield the same cuboid after projecting down to dimensions shared with the query. We keep only the cheapest such cuboid and discard the others. The dimensionality of the cuboid before projection is used as an estimate for the cost of projecting it. Thus, at the end of the first step, we have a set of subsets of the query along with an associated cost for computing that projection of the query cuboid.

In the second step of the prepare phase, any redundant intersection is removed. We call an intersection K redundant if there exists another intersection J such that $J \supseteq K$ and either the cost of obtaining J is at least as cheap as the cost for obtaining K or the cost of obtaining J is within the specified threshold d_{cheap}.

Algorithm 3: Algorithm for finding projections of queries relevant for answering it

```
input  : Query $Q$, set of dimensions of materialized cuboids $M$, number of binary
         dimensions $n$, parameters $d_{\max}$ and $d_{\text{cheap}}$ indicating maximum dimensionality
         to consider and the maximum dimensionality for which the projection is
         considered cheap
output : Fetch plan $P$ specifying which materialized cuboids are to be projected
def Prepare($Q$, $M$, $n$, $d_{\max}$, $d_{cheap}$):
    $P \leftarrow$ empty map with default value $[n]$
    foreach $I \in M$ with $|I| \leq d_{\max}$ do
        $J \leftarrow P(I \cap Q)$ // Cheapest cuboid with same intersection
        if $|I| < |J|$ then
            $P(I \cap Q) \leftarrow I$
    foreach $K \in keys(P)$ do
        Remove entry with key $K$ from $P$ if there exists another key $J$ such that $J \supset K$
         and $(|P(J)| < |P(K)|$ or $|P(J)| < d_{\text{cheap}})$
    return $P$
```

To find the smallest subsuming cuboid of Q that is materialized in M, we set the parameters $d_{\max} = d_{\text{cheap}} = n$. Setting $d_{\max} = n$ allows the base cuboid to be included, and setting $d_{\text{cheap}} = n$ ensures that any projection containing only a subset of the query dimensions is eliminated in the second round. The other solvers in Sudokube that aim to approximate query results from their projections set the parameters $d_{\max} = d_{\text{cheap}} = n - 1$ while preparing for batch mode and $d_{\max} = n$, $d_{\text{cheap}} = 2$ while preparing for online mode. These parameters ensure that, in the batch mode, only maximal projections are processed, and the base cuboid

Table 5.1: Implementation of other aggregations

Operation	Implementation
$\texttt{COUNT}(X)$	$\texttt{SUM}(1)$
$\texttt{AVG}(X)$	$\frac{\texttt{SUM}(X)}{\texttt{SUM}(1)}$
$\texttt{VAR}(X)$	$\frac{\texttt{SUM}(1)\texttt{SUM}(X^2) - (\texttt{SUM}(X))^2}{(\texttt{SUM}(1))^2}$
$\texttt{COR}(X,Y)$	$\frac{\texttt{SUM}(1)\texttt{SUM}(X \cdot Y) - \texttt{SUM}(X)\texttt{SUM}(Y)}{\sqrt{\texttt{SUM}(1)\texttt{SUM}(X^2) - (\texttt{SUM}(X))^2}\sqrt{\texttt{SUM}(1)\texttt{SUM}(Y^2) - (\texttt{SUM}(Y))^2}}$
$\texttt{REG}(X,Y)$	$\frac{\texttt{SUM}(1)\texttt{SUM}(X \cdot Y) - \texttt{SUM}(X)\texttt{SUM}(Y)}{\texttt{SUM}(1)\texttt{SUM}(X^2) - (\texttt{SUM}(X))^2}$

is never projected. But, in the online mode, the base cuboid can be fetched, and all query projections, no matter how small, are processed. Throughout the rest of this thesis, we shall refer to the set of cuboids prepared for answering query Q as $\mathscr{I}(Q)$.

Algorithm 3 returns the map containing both the relevant projections $\mathscr{I}(Q)$ and the materialized cuboids that must be projected to obtain them. This constitutes the fetch plan that is given to the backend to execute. We shall cover algorithms for projecting cuboids in detail later in Chapter 8. The projected cuboids are sent to the solvers, which use them to answer queries. We will go into more detail about the solving techniques in Chapter 6.

5.5 Output Post-processing

Once the solver produces some query result, the output handler applies post-processing operations: it decodes the binary keys to ranges of original column keys, permutes the results to match the user-specified order, and displays the results in table or chart form. If a chart format is selected, each row of the tabular query result is mapped to a different data series in the chart.

The current version of the Sudokube core engine reconstructs cuboids in their entirety and cannot construct only slices specified by the filters. Therefore, the cuboid mapped from the query contains all the binary dimensions specified in the query, including those of the filters. The output handler applies the filters by slicing the results to include only the specified keys for the filter dimensions. The result is projected to contain only binary dimensions for the horizontal and vertical axes, aggregating multiple keys selected for filters together.

While the current solving techniques support `SUM` as the primary aggregation, related aggregations are achieved through post-processing. Table 5.1 illustrates how other aggregations can be computed on some measures X and Y using sums of measures that are functions of X and Y. To use these aggregations, the specified functions on X and Y must be computed as measures during cube construction.

Other operations that could be performed by post-processing query results include window-based aggregations and aggregations based on any user-defined grouping. In these cases, the interpreter maps the query to a cuboid that includes all the binary dimensions required to specify the custom grouping. The output handler performs the aggregation with the custom grouping during post-processing. Queries on data cube views that transform the dimensions in some way are also handled in a similar way.

6 Sudokube Solvers

Sudokube chooses only a subset of the cuboid lattice to materialize during data cube construction and relies on one of several *solvers* that use these materialized cuboids to answer queries quickly. In this chapter, we perform a comprehensive exploration and comparative analysis of these solvers, shedding light on their distinctive features, advantages, and trade-offs.

We begin our journey by introducing the solvers available in Sudokube. We explore how these solvers leverage various algorithms, techniques, and approximation methods to derive query results from the subset of materialized cuboids. Furthermore, we explore the implementation aspects of integrating these solvers into Sudokube, discussing the technical considerations, optimizations, and design choices that enhance their efficiency and effectiveness.

To validate and benchmark the solvers' performance, we conduct extensive experiments using diverse datasets and query workloads. The experimental evaluations provide valuable insights into the solvers' behavior in real-world scenarios, enabling us to make informed comparisons and draw meaningful conclusions.

Throughout this chapter, we shall use a running example that describes how the same query is evaluated by different solvers in Sudokube. This example contains a low-dimensional query on a low-dimensional dataset for the sake of showing all the different computations, but they generalize to higher-dimensional data and queries as well.

Example 7. *Consider the binary sales data cube described in Example 5. For simplicity, let us ignore the binary dimensions* 5 *and* 4 *encoding items and focus on the remaining dimensions. Figure 6.1 shows the lattice of all cuboids. Suppose only the orange-colored cuboids are picked for materialization. Figure 6.2 shows the contents of these cuboids.*

Suppose we want to find the total sales grouped by city and half-year. In this data cube, the city is encoded by a cosmetic dimension formed by grouping binary dimensions 1 *and* 0*, and the half-year is encoded by binary dimension* 3*, a prefix of the binary dimensions forming the cosmetic dimension for quarters. The query is mapped to the binary cuboid* $C_{\{3,1,0\}}$ *containing the total sales for every combination of these binary dimensions, but it is not materialized.*

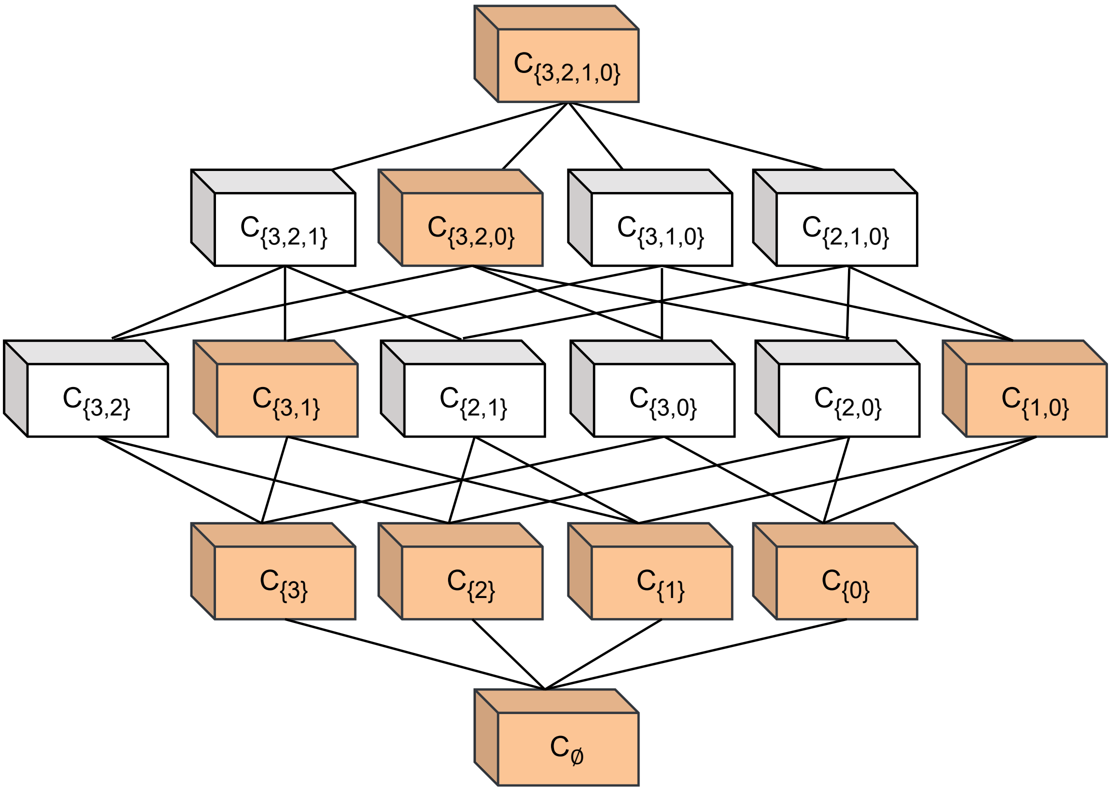


Figure 6.1: Lattice of cuboids containing binary dimensions 0…3 from the sales data cube

3	2	1	0	Sales
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	1	0	2
1	0	0	0	3
1	0	1	0	2
1	1	0	0	4
1	1	0	1	2
1	1	1	0	1

3	2	0	Sales
0	0	0	1
0	0	1	2
0	1	0	2
1	0	0	5
1	1	0	5
1	1	1	2

Sales
17

3	1	Sales
0	0	1
0	1	4
1	0	9
1	1	3

1	0	Sales
0	0	7
0	1	3
1	0	6
1	1	1

3	Sales
0	5
1	12

2	Sales
0	8
1	9

1	Sales
0	10
1	7

0	Sales
0	13
1	4

Figure 6.2: Values in the materialized cuboids for the example sales data cube

6.1 Answering Queries Exactly

The first solver we examine in Sudokube is the *naive* solver. This solver follows the approach described in [45] and aims to provide exact query answers using the subset of cuboids that are materialized in the data cube. The query can be answered exactly only by projecting a subsuming cuboid that encompasses all the dimensions of the query. The naive solver selects the smallest subsuming cuboid for the projection, as this cuboid typically has the lowest projection cost. It then processes only this single cuboid to obtain the query result, without requesting or processing any other cuboid.

Example 8. *The smallest subsuming cuboid for the query cuboid* $C_{\{3,1,0\}}$ *is the cuboid* $C_{\{3,2,1,0\}}$*. The naive solver projects this* 4*-D cuboid to obtain the query result.*

As long as the base cuboid is materialized in a data cube, the naive solver can always answer any query exactly by projecting this base cuboid in the worst case. However, as discussed in Section 3.4.2, the expected size of the smallest subsuming cuboid for a query with more than 3 or 4 dimensions is comparable to that of the base cuboid in a high-dimensional data cube. Consequently, projecting these large cuboids can be time-consuming for big datasets, resulting in slower query processing. We can observe this behavior in the experiment results that we examine next.

Experiment 6.1 *Varying Query Dimensionality on Naive Solver*

We evaluate 100 random queries each for various levels of dimensionality, using the naive solver on two data cubes each, on either dataset, one following the Random and the other following the Prefix materialization strategies. For the NYC dataset, we pick both data cubes with the total number of cuboids $N = 2^{15}$ and minimum dimensionality $d_{\min} = 18$, whereas for the SSB dataset, we pick data cubes materialized with parameters $N = 2^{15}$ and $d_{\min} = 14$. Figure 6.3 shows the histogram describing the fraction of queries answered by projecting a cuboid of a particular dimensionality.

Our observations reveal a consistent pattern regarding the dimensionality of cuboids utilized in responding to queries by the naive solver. Specifically, data cubes built on the NYC dataset have more binary dimensions compared to those built on the SSB dataset. This implies that for any non-zero cuboid dimensionality, the space encompassing all cuboids is smaller for SSB compared to NYC. In parallel, due to imposed constraints, the Prefix strategy chooses from a considerably smaller space of cuboids in contrast to the Random strategy. In both scenarios, a reduced space of cuboids to choose from amplifies the likelihood that a particular cuboid is materialized given the same number of materialized cuboids. Consequently, for the same query dimensionality, the naive solver is more likely to answer queries from smaller cuboids successfully. This success rate is observed to be higher for data cubes based on SSB compared to those based on NYC, and similarly for cubes utilizing the Prefix strategy in contrast to the Random strategy.

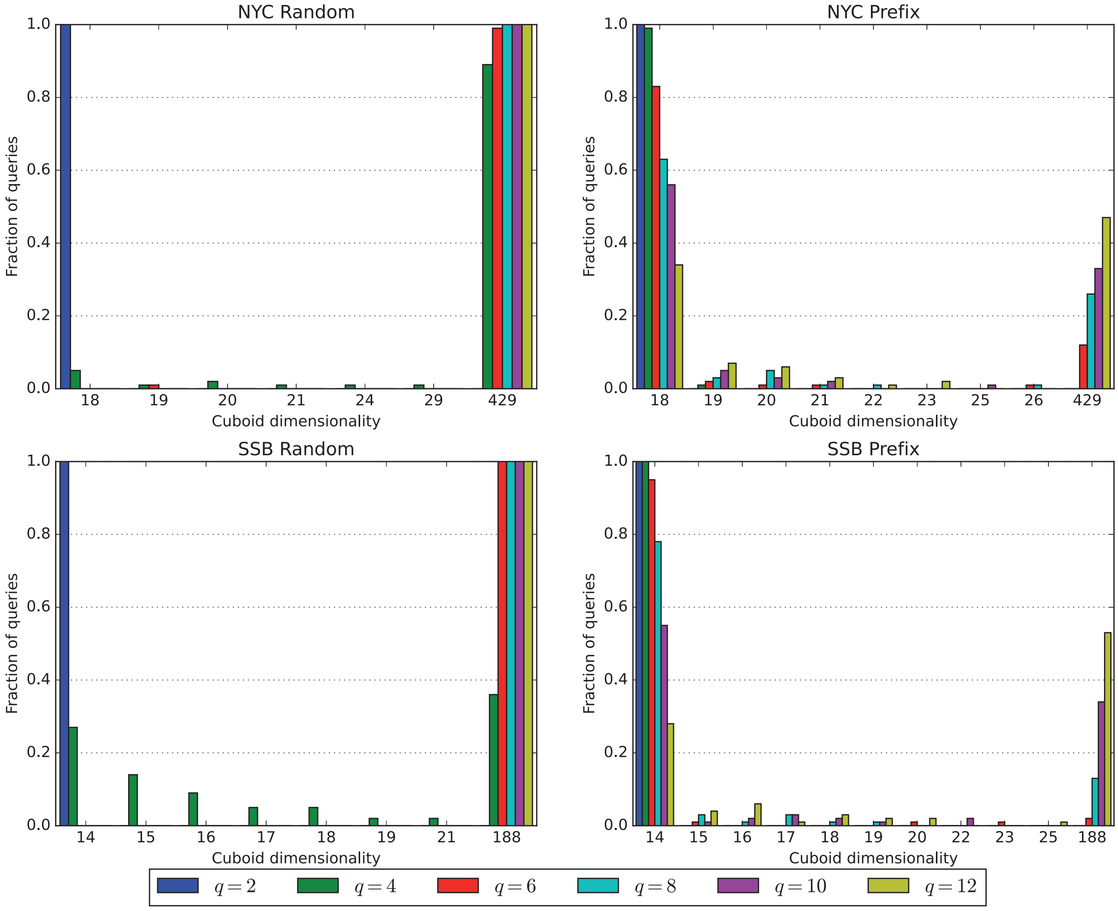


Figure 6.3: Fraction of queries answered by the naive solver by projecting cuboids of various dimensionality for different query dimensionality

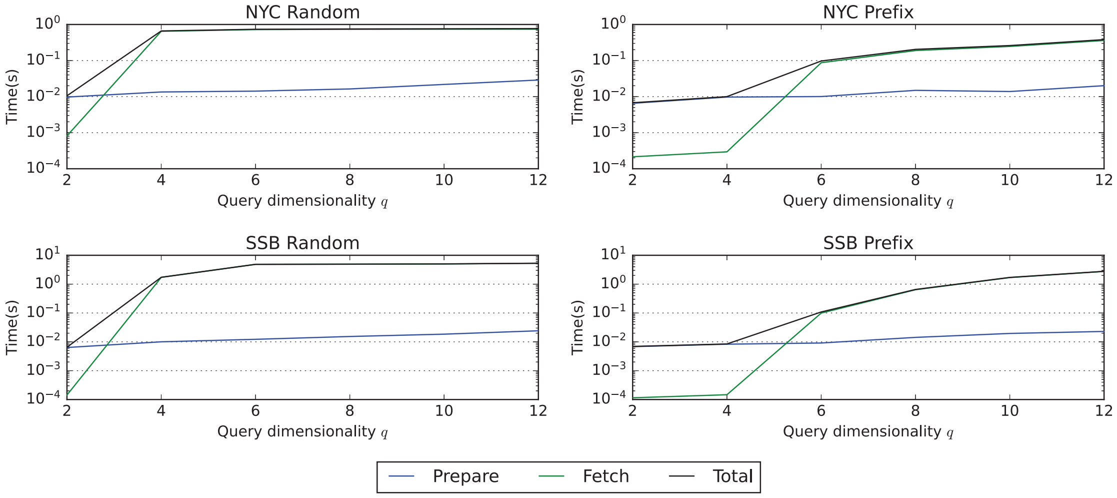


Figure 6.4: Average time spent by the naive solver in every phase of query execution in batch mode for different query dimensionality

Across all four data cubes, all 2-dimensional queries are answered from the lowest dimensionality materialized cuboids. As the query dimensionality increases, fewer queries are answered from lower-dimensional cuboids, and more queries are answered by projecting the base cuboid. Nearly all queries are answered by projecting the base cuboid of the Random strategy data cubes when the query dimensionality is 6 or higher. The fraction of 6-dimensional queries answered by projecting the base cuboid is 12% for NYC Prefix and 2% for SSB Prefix data cubes, which increases to 47% and 53% when the query dimensionality is increased to 12.

Figure 6.4 presents the average time spent on prepare and fetch for the same set of queries and data cubes. We observe that the average fetch time begins quite low, then rises as higher-dimensional cuboids are projected more frequently, and finally, plateaus when all queries are answered from the base cuboid. The maximum fetch time is greater for SSB because it has more rows, which significantly influence the fetch time in sparse storage layouts. Conversely, the prepare time only experiences a marginal increase with the query dimensionality. △

Experiment 6.2 *Varying Number of Materialized Cuboids in Naive Solver*

We fix query dimensionality and the minimum dimensionality d_{min} and run 100 queries on data cubes that differ in the number of materialized cuboids. We repeat the experiments for NYC and SSB datasets following both prefix and random strategies. We select random queries of dimensionality 4 and prefix queries of dimensionality 6 for this experiment. The minimum dimensionality of materialized cuboids was selected to be 18 for NYC and 14 for SSB.

Figure 6.5 shows, for each cuboid dimensionality, the fraction of queries answered by projecting some cuboid of that dimensionality, when run on data cubes with different numbers of cuboids materialized. By increasing the total number of materialized cuboids, we increase the probability of finding a materialized subsuming cuboid among the smaller cuboids. For instance, in the NYC Prefix data cube, the fraction of queries answered by projecting one of the 18-dimensional cuboids increases from 13% when a total of 2^6 cuboids are materialized to 83% when 2^{15} cuboids are materialized.

Figure 6.6 shows the average prepare and fetch time for the same set of queries. The prepare time increases linearly with the number of materialized cuboids as more of them need to be processed to find the cuboid used for answering the query. However, the average fetch time decreases in tandem with the increased rate of projecting low-dimensional cuboids to answer queries. We observe that increasing the number of materialized cuboids generally leads to an overall decrease in the execution time for the query for the range of parameters we examine in this experiment. But, this trend will continue only as long as fetch time dominates the prepare time, after which the total time increases due to a more significant prepare time. △

Experiment 6.3 *Varying Minimum Dimensionality of Materialized Cuboids on Naive Solver*

In another experiment, we maintain the total number of materialized cuboids at 2^{15} and run 100 queries on data cubes with varying minimum dimensionality of the materialized cuboids.

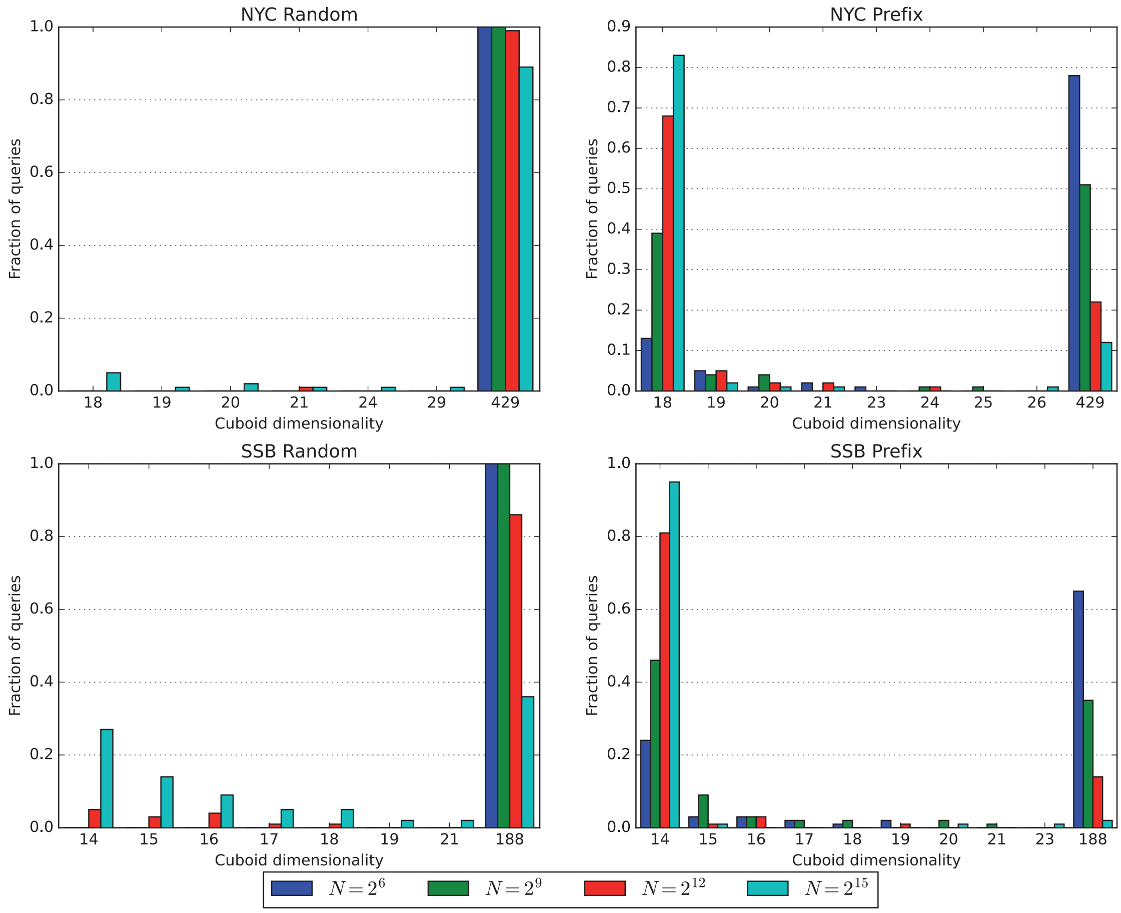


Figure 6.5: Fraction of queries answered by the naive solver by projecting cuboids of various dimensionality for different number of materialized cuboids

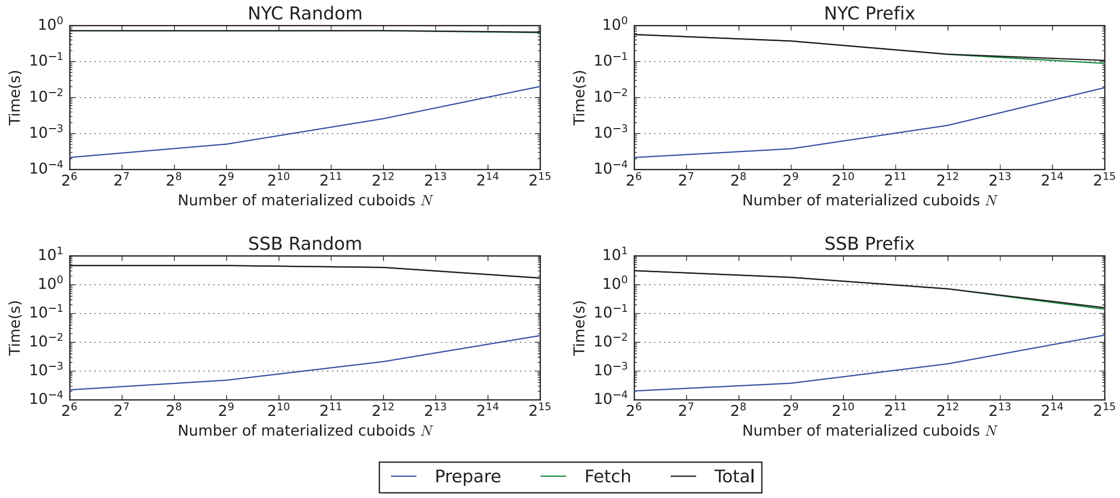


Figure 6.6: Average time spent by the naive solver in every phase of query execution in batch mode for different number of materialized cuboids

The dimensionality of the queries is set to 4 for the data cubes built using the Random strategy and 6 for those built using the Prefix strategy.

Figure 6.7 presents the average fraction of queries answered by projecting cuboids of various dimensionality for different minimum dimensionality of materialization. As we increase d_{min}, which also signifies the dimensionality with the most number of materialized cuboids, more queries are answered by projecting cuboids of this dimensionality. Consequently, fewer queries are answered by projecting the base cuboid.

Figure 6.8 depicts the average prepare and fetch times for the same experiment. As d_{min} rises, the prepare phase takes slightly more time. While an increase in d_{min} does lengthen the time required to project those cuboids, the reduction in the more time-consuming base cuboid projections results in an overall decrease in the average fetch time and, thus, the total time. △

The results of these experiments show that nearly 100% of queries are answered by the naive solver by projecting the base cuboid beyond query dimensionality 6 in the case of the Random cubes. In the case of the Prefix cubes, about 50% of the 12-D queries are answered by projecting the base cuboid. Projecting the base cuboid takes about 1 second in the case of NYC cubes and around 7 seconds for the SSB dataset, and would take longer for bigger datasets. Thus, the naive solver would not be ideal for answering queries on large datasets.

We explore more interesting solvers in the following sections of this chapter. These alternative solvers employ a different approach, reconstructing the query from its materialized projections. This allows them to provide approximate results quickly after processing a few low-dimensional projections. As more projections are processed, the results are continuously updated until the smallest subsuming cuboid of the query cuboid is projected, and the exact query answer is obtained. We use the naive solver as the baseline to compare the performance and accuracy of these other solvers.

6.2 Solving Queries Using Linear Programming

The *linear programming* solver in Sudokube offers an alternative approach to answer queries by constructing a system of linear equations from its projections. These linear equations describe how the unknown values in the query cuboid are aggregated to form values in its projections and capture the constraints imposed by these projections. By solving this system, the solver is able to provide lower and upper bounds for each variable, representing the values in the query result.

For a given query Q, the query result, represented by the cuboid C_Q, consists of $2^{|Q|}$ cells identified by the vector version $\boldsymbol{q}$ of a function $q \in \{0,1\}^Q$. To simplify the notation, we use the variable $v_{\boldsymbol{q}}$ to represent the unknown value $C_Q(\boldsymbol{q})$. For a cuboid C_J where $J \subseteq Q$, Equation (4.1) yields a system of $2^{|J|}$ linear equations constraining variables $v_{\boldsymbol{q}}$, one for each cell in the cuboid

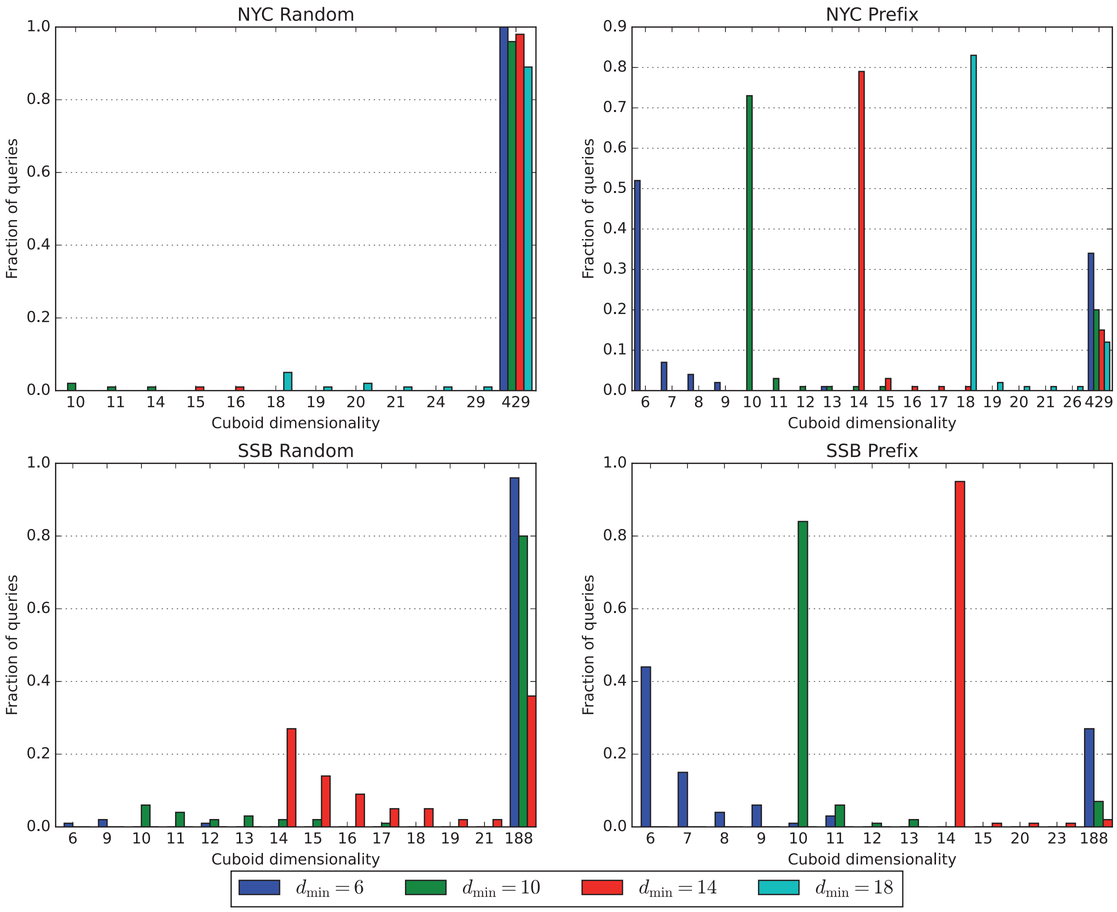


Figure 6.7: Fraction of queries answered by the naive solver by projecting cuboids of various dimensionality for different values of minimum dimensionality of materialized cuboids

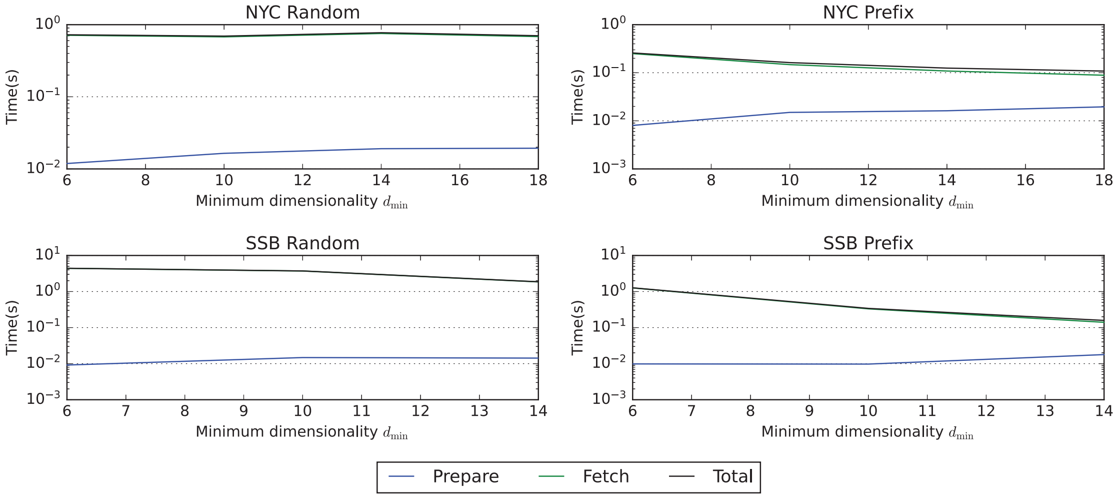


Figure 6.8: Average time spent by the naive solver in every phase of query execution in batch mode for different values of minimum dimensionality of materialized cuboids

C_J. The equation $e_{\boldsymbol{r}}$ for a cell $\boldsymbol{r} \in \{0,1\}^J$ sums up all variables $v_{\boldsymbol{q}}$ such that $\boldsymbol{q}$ is consistent with $\boldsymbol{r}$ on J and is given by:

$$e_{\boldsymbol{r}}: \qquad \sum_{\boldsymbol{q} \in \{0,1\}^Q} \left\{ v_{\boldsymbol{q}} \mid \boldsymbol{q}_{\downarrow J} = \boldsymbol{r} \right\} \quad = \quad C_J(\boldsymbol{r})$$

We use the notation of cuboid C_J to also refer to the set of equations given by the cuboid. In scenarios where multiple projections of the cuboid C_Q are available to constrain its values, we can combine the linear equations from each cuboid. However, this system of equations is generally not linearly independent. To optimize the solving time, we aim to obtain a maximal set of linearly independent equations that capture all the constraints from the fetched cuboids while using as few equations as possible.

		v_{000}	v_{001}	v_{010}	v_{011}	v_{100}	v_{101}	v_{110}	v_{111}	
$C_{\{1,0\}}$	e_{*00}	①				1				7
	e_{*01}		①				1			3
	e_{*10}			①				1		6
	e_{*11}				①				1	1
$C_{\{3,1\}}$	e_{00*}	1	1							1
	e_{01*}			1	1					4
	e_{10*}					①	1			9
	e_{11*}							①	1	3
$C_{\{3,0\}}$	e_{0*0}	1		1						3
	e_{0*1}		1		1					2
	e_{1*0}					1		1		10
	e_{1*1}						①		1	2

Figure 6.9: The 12 equations obtained from the three relevant materialized cuboids of the sales data cube for the query $Q = \{3,1,0\}$. Among these, at most 7 are linearly independent.

Example 9. *For the query $Q = \{3,1,0\}$ in the sales data cube, the materialized projections of the cuboid C_Q include $C_{\{1,0\}}$ and $C_{\{3,1\}}$. Additionally, the projection $C_{\{3,0\}}$ can be obtained by projecting $C_{\{3,2,0\}}$ at query time. All other proper projections are subsumed by these three projections and, therefore, need not be processed. The system of linear equations defined by these three projections is shown in Figure 6.9. Each row in the table denotes one equation, and each column contains the coefficients for some variable $v_{\boldsymbol{q}}$ or the total sum for that equation.*

Each equation represents a constraint on the values of cells in the cuboid C_Q. For example, the equation e_{*00} represents the constraint $v_{000} + v_{100} = 7$. It is worth noting that the mappings are explicitly shown for all dimensions of Q, with missing dimensions indicated by $*$. Out of the 12 equations, at most 7 are linearly independent. The other 5 equations do not add any additional constraint and are colored gray.

Given the non-homogeneous system of linear equations $\boldsymbol{Av} = \boldsymbol{b}$, the row vectors of $\boldsymbol{A}$ define a vector space [49], a subspace of the $2^{|Q|}$-dimensional space spanned by all the v_q. We can find a maximal set of linearly independent equations by constructing a basis [49] for this vector space. We are also interested in the kernel of $\boldsymbol{A}$, which represents the independent solutions for the homogeneous system of equations $\boldsymbol{A}\boldsymbol{v} = \boldsymbol{0}$. The solutions to the original non-homogeneous system of equations can be expressed as the sum of a fixed solution $\boldsymbol{u}$ and an arbitrary element of the kernel [68]. A zero-dimensional kernel means that there is only one solution and we can reconstruct C_Q precisely. The existence of at least one solution is guaranteed by construction.

A basis for the row space of $\boldsymbol{A}$ can be constructed by grouping equations into equivalence classes and picking one from each. One way to group the equations is to use their first variable when ordered lexicographically according to the order $\prec$ for $\boldsymbol{q}$, as described in Section 2.1. Formally, we define $\hat{v}_e$ to be the minimal variable that occurs in the equation e and write $e_1 \equiv e_2$ for two equations e_1 and e_2 when $\hat{v}_{e_1} = \hat{v}_{e_2}$. Of course, $\equiv$ forms an equivalence relation.

Example 9 (continued)**.** *The equivalence classes of equations w.r.t.* $\equiv$ *are* $S_{000} = \{e_{*00}, e_{00*}, e_{0*0}\}$, $S_{001} = \{e_{*01}, e_{0*1}\}$, $S_{010} = \{e_{*10}, e_{01*}\}$, $S_{011} = \{e_{*11}\}$, $S_{100} = \{e_{10*}, e_{1*0}\}$, $S_{101} = \{e_{1*1}\}$, $S_{110} = \{e_{11*}\}$ *and* $S_{111} = \emptyset$. *We pick the equations with circled ones to form a basis.*

Before we prove the correctness of our claim regarding the construction of a basis, we need to introduce some additional notations and lemmas. Given two sets J and L such that $L \subseteq J \subseteq Q$, and a mapping $\boldsymbol{\ell} \in \{0,1\}^L$, we define the equations for the slice ℓ of the cuboid C_J that are consistent with $\boldsymbol{\ell}$ as

$$C_{J|\boldsymbol{\ell}} := \left\{ e_{\boldsymbol{j}} \in C_J \mid \boldsymbol{j} = \{0,1\}^J \text{ and } \boldsymbol{j}_{\downarrow L} = \boldsymbol{\ell} \right\}.$$

Lemma 4. *Given any two sets* $L_1, L_2 \subseteq Q$, *and a slice* $\boldsymbol{\ell} \in \{0,1\}^{L_1 \cap L_2}$ *present in both* C_{L_1} *and* C_{L_2},

$$\sum C_{L_1|\boldsymbol{\ell}} = C_{L_1 \cap L_2}(\boldsymbol{\ell}) = \sum C_{L_2|\boldsymbol{\ell}}.$$

Proof. From its definition, we know that the set $C_{L_1|\boldsymbol{\ell}}$ contains equations $e_{\boldsymbol{r}}$ from cuboid C_{L_1} that are consistent with $\boldsymbol{\ell}$ on $L_1 \cap L_2$. The equations $e_{\boldsymbol{r}}$ themselves are sums of variables $v_{\boldsymbol{q}}$ for $\boldsymbol{q} \in \{0,1\}^Q$ that are consistent with $\boldsymbol{r}$ on L_1. Combining them, we have

$$\begin{aligned}
\sum C_{L_1|\boldsymbol{\ell}} &= \sum_{\boldsymbol{r} \in \{0,1\}^{L_1}} \left\{ e_{\boldsymbol{r}} \mid e_{\boldsymbol{r}} \in C_{L_1} \text{ and } \boldsymbol{r}_{\downarrow L_1 \cap L_2} = \boldsymbol{\ell} \right\} \\
&= \sum_{\boldsymbol{r} \in \{0,1\}^{L_1}} \sum_{\boldsymbol{q} \in \{0,1\}^{Q}} \left\{ v_{\boldsymbol{q}} \mid \boldsymbol{q}_{\downarrow L_1} = \boldsymbol{r} \text{ and } \boldsymbol{r}_{\downarrow L_1 \cap L_2} = \boldsymbol{\ell} \right\} \\
&= \sum_{\boldsymbol{q} \in \{0,1\}^{Q}} \left\{ v_{\boldsymbol{q}} \mid \boldsymbol{q}_{\downarrow L_1 \cap L_2} = \boldsymbol{\ell} \right\} \\
&= C_{L_1 \cap L_2}(\boldsymbol{\ell})
\end{aligned}$$

A similar derivation exists for $C_{L_2|\boldsymbol{\ell}}$ as well, proving the lemma. □

Example 10. *Let $\boldsymbol{u} \in \{0,1\}^{\{0\}}$, $\boldsymbol{v} \in \{0,1\}^{\{1\}}$ and $\boldsymbol{w} \in \{0,1\}^{\{3\}}$ be zero vectors on dimensions* 0, 1 *and* 3 *respectively. Continuing our running example of the sales data cube, we have the following set of equations from different slices of the projections $C_{\{1,0\}}$, $C_{\{3,1\}}$ and $C_{\{3,0\}}$:*

$$\begin{aligned} C_{\{1,0\}|\boldsymbol{u}} &= \{e_{*00}, e_{*10}\} & C_{\{1,0\}|\boldsymbol{v}} &= \{e_{*00}, e_{*01}\} \\ C_{\{3,1\}|\boldsymbol{v}} &= \{e_{00*}, e_{10*}\} & C_{\{3,1\}|\boldsymbol{w}} &= \{e_{00*}, e_{01*}\} \\ C_{\{3,0\}|\boldsymbol{u}} &= \{e_{0*0}, e_{1*0}\} & C_{\{3,0\}|\boldsymbol{w}} &= \{e_{0*0}, e_{0*1}\} \end{aligned}$$

Then, from Lemma 4, we have the following linear dependencies.

$$\begin{aligned} e_{0*0} &= e_{*00} + e_{*10} - e_{1*0} && \textit{from } C_{\{1,0\}|\boldsymbol{u}} \textit{ and } C_{\{3,0\}|\boldsymbol{u}} \\ e_{00*} &= e_{*00} + e_{*01} - e_{10*} && \textit{from } C_{\{1,0\}|\boldsymbol{v}} \textit{ and } C_{\{3,1\}|\boldsymbol{v}} \\ e_{0*0} &= e_{00*} + e_{01*} - e_{0*1} && \textit{from } C_{\{3,1\}|\boldsymbol{w}} \textit{ and } C_{\{3,0\}|\boldsymbol{w}} \end{aligned}$$

Next, we establish two auxiliary results. We say that a variable v *dominates* an equation e if it is the same as or appears before (according to order $\prec$) the minimal variable $\hat{v}_e$ of the equation. A variable v dominates a set of equations if it dominates each of its members.

Lemma 5. *Given an equation $e_{\boldsymbol{j}} \in C_J$ and a slice $\boldsymbol{\ell} \in \{0,1\}^L$ for some $L \subseteq J$ such that $\boldsymbol{j} = \boldsymbol{\ell}_{\uparrow J}$, then $\hat{v}_{e_j}$ dominates $C_{J|\boldsymbol{\ell}}$.*

Proof. $C_{J|\boldsymbol{\ell}}$ is defined as the set of equations $e_{\boldsymbol{r}} \in C_J$ such that $\boldsymbol{r}$ is consistent with $\boldsymbol{\ell}$. These equations consist of variables $v_{\boldsymbol{q}}$ such that any such $\boldsymbol{q}$ is consistent with $\boldsymbol{\ell}$ on dimensions in L. The vector $\boldsymbol{j}$ extends $\boldsymbol{\ell}$ by mapping all other dimensions in $J \setminus L$ to zero. The minimal variable $\hat{v}_{e_j}$ in equation $e_{\boldsymbol{j}}$ maps all other dimensions in $Q \setminus J$ to zero as well. No other variable $v_{\boldsymbol{q}}$ whose index $\boldsymbol{q}$ is consistent with $\boldsymbol{\ell}$ can come before $\hat{v}_{e_j}$ in the lexicographic order $\prec$. Obviously, $\hat{v}_{e_j}$ dominates all the $e_{\boldsymbol{r}}$. □

Lemma 6. *Given two equations $e_{\boldsymbol{s}} \in C_S$ and $e_{\boldsymbol{t}} \in C_T$ such that $\hat{v}_{e_s} = \hat{v}_{e_t}$, then*
(1) $\boldsymbol{s}_{\downarrow S \cap T} = \boldsymbol{t}_{\downarrow S \cap T}$ *, and* (2) $\boldsymbol{s} = \boldsymbol{\ell}_{\uparrow S}$ *and* $\boldsymbol{t} = \boldsymbol{\ell}_{\uparrow T}$ *for* $\boldsymbol{\ell} = \boldsymbol{s}_{\downarrow S \cap T}$.

Proof. Let $v_{\boldsymbol{q}}$ be the minimal variable in equations $e_{\boldsymbol{s}}$ and $e_{\boldsymbol{t}}$. Therefore, $\boldsymbol{q} \in \{0,1\}^Q$ is consistent with $\boldsymbol{s}$ on dimensions in S and $\boldsymbol{t}$ on dimensions in T.

Since $\boldsymbol{q}$ and $\boldsymbol{s}$ are consistent on dimensions in S, they are consistent on its subset $S \cap T$. Similarly, $\boldsymbol{q}$ and $\boldsymbol{t}$ are consistent on $S \cap T$ as well. Therefore, we have $\boldsymbol{s}_{\downarrow S \cap T} = \boldsymbol{q}_{\downarrow S \cap T} = \boldsymbol{t}_{\downarrow S \cap T}$, proving (1).

From Equations (2.1) and (2.2) we have, $\boldsymbol{s} = \boldsymbol{s}_{\downarrow S\cap T} \uplus \boldsymbol{s}_{\downarrow S\setminus T}$, and $\boldsymbol{\ell}_{\uparrow S} = \boldsymbol{\ell} \uplus \mathbf{0}$. Since $\boldsymbol{\ell} = \boldsymbol{s}_{\downarrow S\cap T}$, for (2), we only need to prove that $\boldsymbol{s}_{\downarrow S\setminus T} = \mathbf{0}$. In other words, we need to prove that $\boldsymbol{s}$ maps all dimensions in $S \setminus T$ to zero. We prove this by contradiction. Assume that some dimension $j \in S \setminus T$ exists that is mapped by $\boldsymbol{s}$ to 1. Since $\boldsymbol{q}$ is consistent with $\boldsymbol{s}$ on S, $\boldsymbol{q}$ maps dimension j to 1 as well. Let $\boldsymbol{r} \in \{0,1\}^Q$ be another vector that is identical to $\boldsymbol{q}$ except for dimension j, which it maps to 0. Since $\boldsymbol{q}$ is consistent with $\boldsymbol{t}$ on dimensions in T and $j \not\in T$, $\boldsymbol{r}$ is consistent with $\boldsymbol{t}$ on dimensions in T as well. Therefore, in addition to $v_{\boldsymbol{q}}$, the variable $v_{\boldsymbol{r}}$ is part of the equation $e_{\boldsymbol{t}}$. But this violates our assumption that $v_{\boldsymbol{q}}$ is the minimal variable in equation $e_{\boldsymbol{t}}$ since $v_{\boldsymbol{r}} \prec v_{\boldsymbol{q}}$. Therefore, $\boldsymbol{s}$ must map all dimensions in $S \setminus T$ to zero and $\boldsymbol{s} = \boldsymbol{\ell}_{\uparrow S}$. A symmetric argument can be made for $\boldsymbol{t} = \boldsymbol{\ell}_{\uparrow T}$ as well. □

Example 11. *Consider again our running example, and equations* $e_{*01} \in C_{\{1,0\}}$ *and* $e_{0*1} \in C_{\{3,0\}}$ *with* $\hat{v}_{e_{*01}} = \hat{v}_{e_{0*1}} = v_{001}$. *Then, for the slice* $\boldsymbol{\ell} = **1 \in \{0,1\}^{\{0\}}$, *we have* $C_{\{1,0\}|\boldsymbol{\ell}} = \{e_{*01}, e_{*11}\}$, *and* $C_{\{3,0\}|\boldsymbol{\ell}} = \{e_{0*1}, e_{1*1}\}$. *Variable* v_{001} *dominates both* $C_{\{1,0\}|\boldsymbol{\ell}}$ *and* $C_{\{3,0\}|\boldsymbol{\ell}}$, *as required by the combination of Lemmas 5 and 6.*

Theorem 7. *Given the system of linear equations yielded by a set of projections of a cuboid, any subset that contains exactly one equation from each equivalence class of* $\equiv$ *is a basis of the vector space spanned by the equations.*

Proof. Let E be an arbitrary set of linear equations constructed by picking one equation from each equivalence class of $\equiv$. This set is clearly linearly independent. Each row vector e has the leftmost nonzero element $\hat{v}_e$, and, by picking exactly one row vector from each equivalence class of $\equiv$, no two distinct row vectors have the same leftmost nonzero element (see Figure 6.9 for an illustration of this).

All that is left to be shown is that this set of linearly independent equations is also maximal. To prove this, we explicitly construct a basis and show that it has the same rank. Let $C_{J_1}, \ldots, C_{J_m}$ be the projections. The algorithm for the construction of the basis B works as follows. Initially, let B contain all equations from C_{J_1}. During step i, for each equation $e \in C_{J_i}$, processed in the order of increasing $\hat{v}_e$, do the following - if no element $d \in B$ exists with $d \equiv e$, add e to B; otherwise, do not.

We prove the correctness of this algorithm by induction, with the induction hypothesis that after each step, all equations processed so far are linearly dependent with B, and the elements of B are linearly independent. The equations of a single projection are linearly independent, so by initially setting B to all of C_{J_1}, we satisfy the induction hypothesis initially.

Let B be a basis for $C_{J_1} \cup \cdots \cup C_{J_i}$ after i steps. During step $i+1$, we process the equations e of $C_{J_{i+1}}$ in the reverse order of $\hat{v}_e$. We maintain the invariant that all equations e_0 of $C_{J_{i+1}}$ previously processed (i.e., with $\hat{x}_e \prec \hat{x}_{e_0}$) are linearly dependent with B. At the start of step $i+1$, this is true from the induction hypothesis. All equations of $C_{J_1} \cup \cdots \cup C_{J_i}$ can be obtained as linear combinations of the equations of B. If there is no $d \in B$ with $d \equiv e$, e is linearly independent of B, and we (may) add e to B. Otherwise, we do not add e, because it is linearly dependent with the existing equations in B.

Now, we prove that e is linearly dependent on B when there exists some $d \in B$ such that $d \equiv e$. Let C_{J_h} $(h \leq i)$ be the projection that contributed d. Let the equation e be indexed by $\boldsymbol{r} \in \{0,1\}^{J_{i+1}}$ in the cuboid $C_{J_{i+1}}$, the equation d be indexed by $\boldsymbol{s} \in \{0,1\}^{J_h}$ in the cuboid C_{J_h}. Also let $L = J_h \cap J_{i+1}$. Then, from Lemma 6 we have $\boldsymbol{\ell} = \boldsymbol{r}_{\downarrow L} = \boldsymbol{s}_{\downarrow L}$ and $\boldsymbol{\ell}_{\uparrow J_{i+1}} = \boldsymbol{r}$. From the definition of the slice of equations, we have $d \in C_{J_h|\boldsymbol{\ell}}$ and $e \in C_{J_{i+1}|\boldsymbol{\ell}}$. By Lemma 5, all the equations in $C_{J_{i+1}|\boldsymbol{\ell}} \setminus \{e\}$ are dominated by $\hat{v}_e$ and thus have been previously processed by the algorithm. By Lemma 4, e is linearly dependent with equations in $C_{J_h|\boldsymbol{\ell}} \cup C_{J_{i+1}|\boldsymbol{\ell}} \setminus \{e\}$, which is a subset of equations processed so far and, by the induction hypothesis, linearly dependent with B. □

Example 12. *If we execute the algorithm of the proof of Theorem 7 on the sales data cube, the first equation we encounter that is not added to B is* e_{01*} $(\boldsymbol{r} = 01*)$ *after step* $i = 1$ *for* $J_{i+1} = \{3,1\}$. *At this stage, B is equal to* $\{e_{*11}, e_{*10}, e_{*01}, e_{*00}, e_{11*}, e_{10*}\}$. *Here,* $d = e_{*11}$, $\hat{v}_d = v_{011}$, *and* $J_h = \{1,0\}$. *We also have* $J_h \cap J_{i+1} = \{1\}$, *and we construct* $\ell = \boldsymbol{r}_{\downarrow\{0\}} = *1*$. *Therefore,* $C_{J_h|\boldsymbol{\ell}} = \{e_{*10}, e_{*11}\}$ *and* $C_{J_{i+1}|\boldsymbol{\ell}} = \{e_{01*}, e_{11*}\}$. *Indeed,* $\hat{v}_{e_{01*}} = v_{010}$ *dominates* $C_{J_{i+1}|\boldsymbol{\ell}}$. $C_{J_h|\boldsymbol{\ell}} \cup (C_{J_{i+1}|\boldsymbol{\ell}} \setminus \{e\})$ *is a subset of B and therefore linearly dependent with it.*

Algorithm 4: Algorithm to construct a basis from the equations yielded by projections of query cuboid C_Q

```
input  : Query $Q$, cuboids $\boldsymbol{C}_I$ for every $I \in \mathscr{I}(Q)$
output : Matrix $\boldsymbol{A}$ and vector $\boldsymbol{b}$ containing the coefficients of equations that form a basis
         for the system of equations $\boldsymbol{A}\boldsymbol{v} = \boldsymbol{b}$
$\boldsymbol{A} \leftarrow$ empty $2^{|Q|} \times 2^{|Q|}$ matrix
$\boldsymbol{b} \leftarrow$ empty $2^{|Q|}$ column vector
foreach $I \in \mathscr{I}(Q)$ do
    foreach $\boldsymbol{i} \in \{0,1\}^I$ do
        Construct equation $e_{\boldsymbol{i}}$ from $C_I$
        $v_{\boldsymbol{s}} \leftarrow$ minimal variable $\hat{v}_{e_{\boldsymbol{i}}}$
        if row indexed by $\boldsymbol{s}$ is empty in $\boldsymbol{A}$ then
            $\boldsymbol{c_s} \leftarrow$ coefficients of $2^{|Q|}$ variables in $e_{\boldsymbol{i}}$
            $b_{\boldsymbol{s}} \leftarrow C_I(\boldsymbol{i})$
            add $\boldsymbol{c_s}$ as row vector indexed by $\boldsymbol{s}$ in $\boldsymbol{A}$
            add $b_{\boldsymbol{s}}$ to the component indexed by $\boldsymbol{s}$ in $\boldsymbol{b}$
return $\boldsymbol{A}$ and $\boldsymbol{b}$
```

Picking a set of linear equations according to Theorem 7 immediately yields a coefficient matrix in *row echelon form* – for each column, there is exactly one row that has a 1 in this column, and only zeroes to its left. The degree of freedom of the system of equations is the number of variables for which no equation has it as its minimal variable. If at least one such variable exists, we cannot answer the query without further constraints. These constraints could be obtained from additional cuboids or some other restriction, such as that the facts must be non-negative. In the following example, even though there is initially one degree of freedom, applying non-negativity constraints restricts the solution to be unique.

Example 13. *In Figure 6.9, we have marked the chosen witness of every equivalence class with a circle. According to Theorem 7,* $\{e_{*00}, e_{*01}, e_{*10}, e_{*11}, e_{10*}, e_{11*}, e_{1*1}\}$ *is a basis for the vector space spanned by the equations of the cuboids* $C_{\{1,0\}}$, $C_{\{3,1\}}$ *and* $C_{\{3,0\}}$. *Since there are eight variables and only seven independent equations, we have a single degree of freedom, and the query cannot be fully answered without further constraints. After Gaussian elimination on the coefficient matrix, we get the equation* $e_{*00} + e_{1*1} - e_{10*} = v_{000} + v_{111} = 0$. *If we impose a non-negativity constraint on all the* v *values, this equation gives us* $v_{000} = v_{111} = 0$, *and so we obtain the query result* $(v_{000}, v_{001}, \ldots, v_{111}) = (0, 1, 3, 1, 7, 2, 3, 0)$.

Algorithm 5: Simple algorithm to improve bounds of variables in an equation using existing bounds on other variables

```
input  : Basis $M$ for the system of equations, query $Q$
output : Bounds for variables in the result for query $Q$
Initialize lower bound $LB(v)$ for every variable $v$ in the result of query $Q$ as 0
Initialize upper bound $UB(v)$ for every variable $v$ in the result of query $Q$ as $\infty$
repeat
    foreach equation $e$ in the basis $M$ do
        foreach variable $v$ with a non-zero coefficient $s$ in $e$ do
            Rearrange $e$ as $e'$ in the form $v = \left(\frac{b}{c} + \sum \frac{-c_i}{c} v_i\right)$
            $P \leftarrow$ indexes $i$ of variables in $e'$ with positive values for coefficient $\frac{-c_i}{c}$
            $N \leftarrow$ indexes $i$ of variables in $e'$ with negative values for coefficient $\frac{-c_i}{c}$
            lower $\leftarrow \frac{b}{c} + \sum_{i \in P} \frac{-c_i}{c} LB(v_i) + \sum_{j \in N} \frac{-c_j}{c} UB(v_j)$
            upper $\leftarrow \frac{b}{c} + \sum_{i \in P} \frac{-c_i}{c} UB(v_i) + \sum_{j \in N} \frac{-c_j}{c} LB(v_j)$
            $LB(v) \leftarrow \max(LB(v), \text{lower})$
            $UB(v) \leftarrow \min(UB(v), \text{upper})$
until a fixed point is reached or a specified number of rounds is over
```

Even with the non-negativity restriction, queries may still have several degrees of freedom, and we cannot compute exact results. In such cases, we find the upper and lower bounds of every variable $v_{\boldsymbol{q}}$. Algorithm 5 shows a simple algorithm for improving the bounds on a particular variable using the tightest bounds on other variables with which it shares an equation. While the algorithm performs some cheap calculations in every iteration, there is no guarantee that it converges within a small number of steps. Furthermore, the bounds returned by this algorithm are not always tight, as it assumes that all variables in any given equation can simultaneously have their respective maximum or minimum values.

The correct method to obtain the tightest bounds on the values of any single variable is through linear programming [37]. We can construct linear programming problems for maximizing and minimizing objective functions comprising each query variable one at a time, subject to the set of linear equations obtained from the projections of the query. In fact, Algorithm 5 can be considered an incorrect version of the simplex algorithm that pivots on every variable without any regard to maintaining feasibility. Algorithm 6 sketches the high-level algorithm for using linear programming and the simplex algorithm for determining tight bounds on individual query variables.

However, the tight bounds for the values of the query variables given by the simplex algorithm do not come for free. There are no guarantees on the number of iterations required by simplex to reach the optimum value for the objective function, and running the simplex algorithm is expensive, especially when there are several degrees of freedom. Furthermore, any computation done by the simplex algorithm would have to be discarded after fetching another projection of the query that yields new equations that makes the previously optimum solution infeasible. For this reason, the linear programming solver in Sudokube uses the simple algorithm to quickly compute some bounds for the query variables initially when the degrees of freedom for the system of equation is high. As more projections of the query are processed and the degrees of freedom are few, the solver switches to the simplex algorithm to find tighter bounds.

Algorithm 6: Using linear programming to find bounds on query variables

```
input  : Basis for the system of equations for query $Q$
output : Bounds for every variable in the result of $Q$
$\boldsymbol{A} \leftarrow$ coefficients of equations that form basis
foreach variable $v_{\boldsymbol{q}}$ for $\boldsymbol{q} \in \{0,1\}^Q$ do
    $\boldsymbol{T}_1 \leftarrow$ simplex tableau from $\boldsymbol{A}$ with objective function to maximize $v_{\boldsymbol{q}}$
    run simplex algorithm on $\boldsymbol{T}_1$ until termination to find upper bound on $v_{\boldsymbol{q}}$
    $\boldsymbol{T}_2 \leftarrow$ simplex tableau from $\boldsymbol{A}$ with objective function to minimize $v_{\boldsymbol{q}}$
    run simplex algorithm on $\boldsymbol{T}_2$ until termination to find lower bound on $v_{\boldsymbol{q}}$
```

The linear programming solver spends more time finding tighter bounds when the degree of freedom is low so that it can identify variables that have the same lower and upper bound imposed on them by the non-negativity restriction on all the variables. In such cases, the variable is marked as solved, and its value is finalized. These scenarios lower the degrees of freedom even further, potentially all the way down to zero, in which case only a single solution satisfies all the constraints, and the query result can be answered exactly. Furthermore, when the degree of freedom is low, the simplex algorithm itself may not require a lot of iterations until it finds the optimum solution and could be cheap.

Our experiments show that this hybrid approach quickly yields tight bounds on query results for low-dimensional queries but does not scale well with query dimensionality. We use two different metrics to represent the quality of the bounds returned by the linear programming solver. The first metric, which we refer to as the normalized cumulative interval span, captures the precision by computing the total span of the bounds treated as intervals and then dividing it by the total aggregate for normalization. Formally, the normalized cumulative interval span for bounds $\ell(v_{\boldsymbol{q}})$ and $u(v_{\boldsymbol{q}})$ for all variables in the query result indexed by $\boldsymbol{q} \in \{0,1\}^Q$ is given by the following equation

$$\sum_{\boldsymbol{q} \in \{0,1\}^Q} \frac{u(v_{\boldsymbol{q}}) - \ell(v_{\boldsymbol{q}})}{total}.$$

The second metric captures the accuracy by computing the sum of absolute values of deviation of the true value of variable $v_{\boldsymbol{q}}$ from the average value of the upper and lower bounds and then dividing it by the total sum for normalization. Formally, we define the midpoint error as

$$\sum_{\boldsymbol{q} \in \{0,1\}^Q} \frac{|v_{\boldsymbol{q}} - \frac{\ell(v_{\boldsymbol{q}})+u(v_{\boldsymbol{q}})}{2}|}{total}.$$

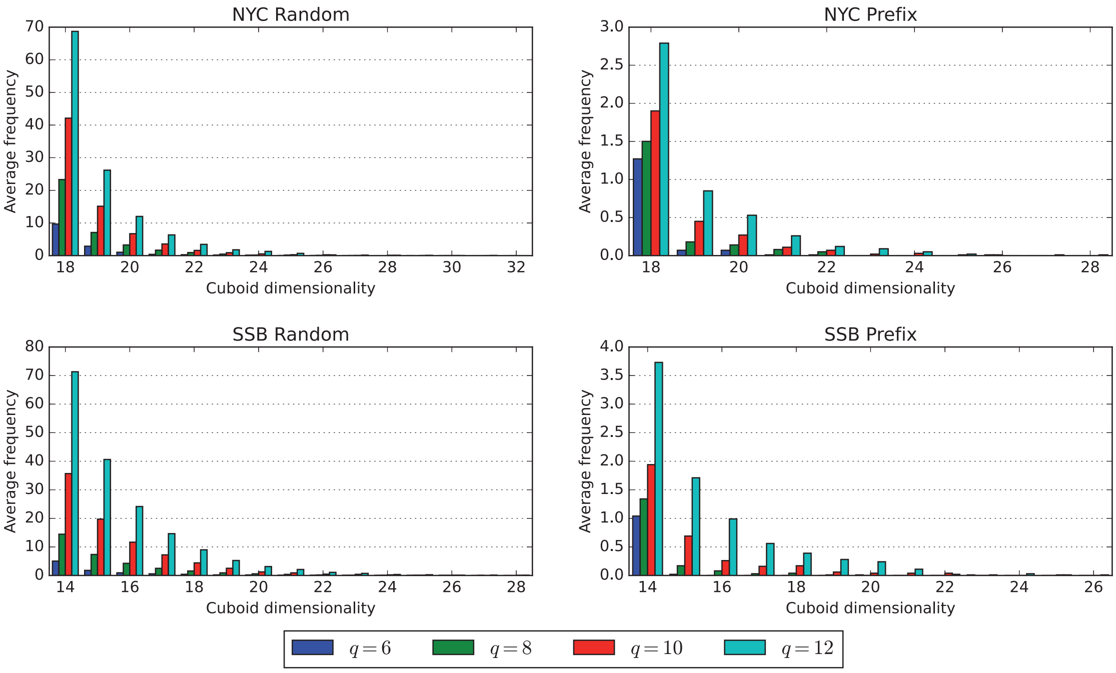
Experiment 6.4 *Varying Query Dimensionality on Linear Programming Solver in Batch Mode*

We submit 100 queries with various levels of dimensionality to the linear programming solver to run in batch mode, targeting data cubes constructed from NYC and SSB datasets using both Random and Prefix strategies. Each data cube contains 2^{15} materialized cuboids with a minimum dimensionality of 14 for SSB and 18 for NYC.

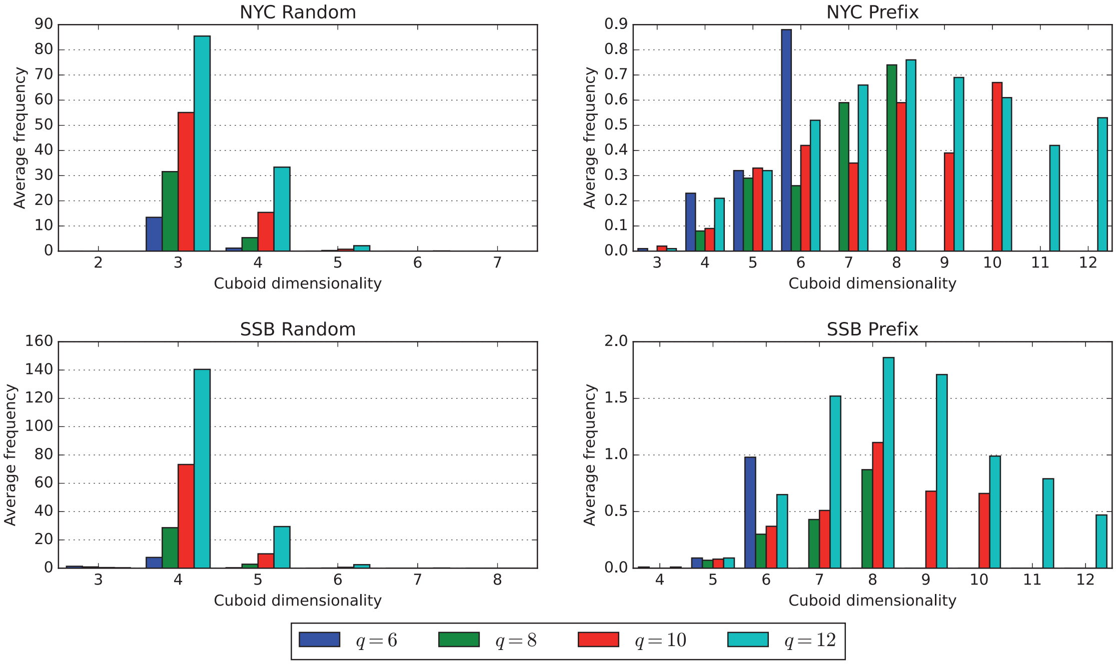
We first analyze the dimensionality of the cuboids employed by the linear programming solver to establish query result boundaries. We are interested in two different dimensionalities – the original dimensionality of the materialized cuboid and its dimensionality after projection to the dimensions present in the query. The original dimensionality of the cuboids impacts fetch time, while the dimensionality after projection affects the solver, both in terms of solving time as well as the uncertainty of the results. Figure 6.10 presents the average frequency count of dimensionality, accounting for both the original dimensionality prior to and the final dimensionality following the projection to the dimensions present in the query. We observe that only a few cuboids are selected for Prefix data cubes, and they have considerable overlap with queries. On the other hand, in the case of Random data cubes, at least 10 times more cuboids are selected, which have a very small overlap with queries.

As the query dimensionality increases, the total number of projections of its result increases exponentially. There is a corresponding increase in the number of cuboids fetched to answer the query, and more higher-dimensional cuboids are fetched. In the case of Prefix cubes, the rise in both the dimensionality of the query and that of the projected cuboids translates into an increase in the dimensionality after projection. However, even though higher dimensional cuboids are projected for data cubes using the Random strategy, there is minimal overlap with the query, keeping the dimensionality after projection relatively low.

Figure 6.11a displays the average prepare, fetch, and solve times when using the linear programming solver to answer queries of varying dimensionality. The prepare time experiences a slight increase as the query dimensionality grows. The escalation in both the quantity and

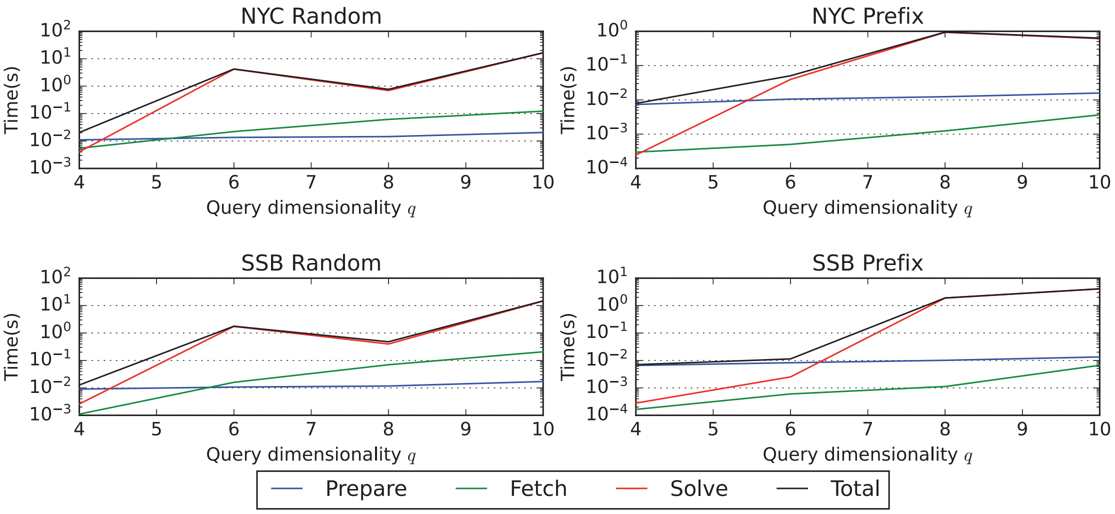


(a) Dimensionality before projection

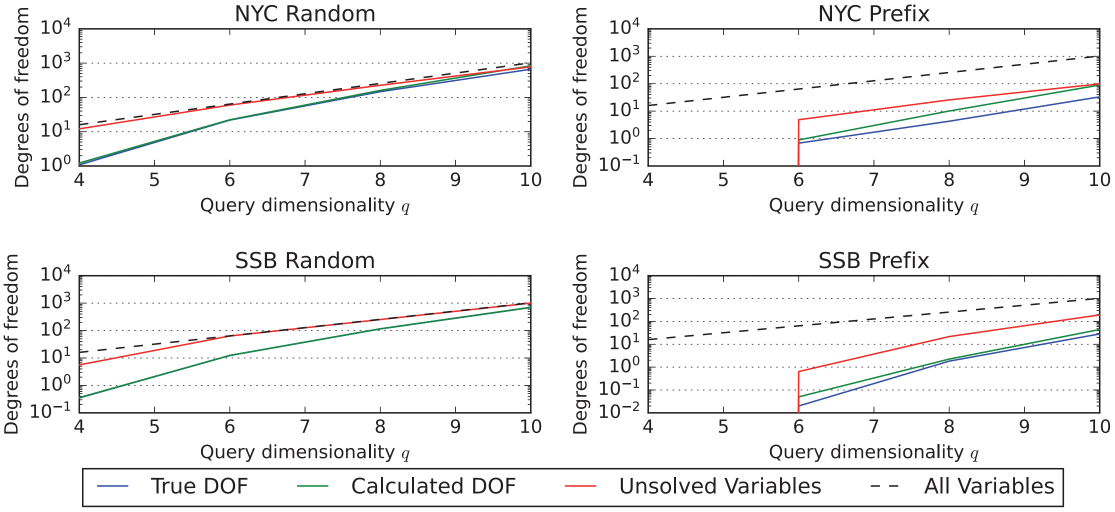


(b) Dimensionality after projection

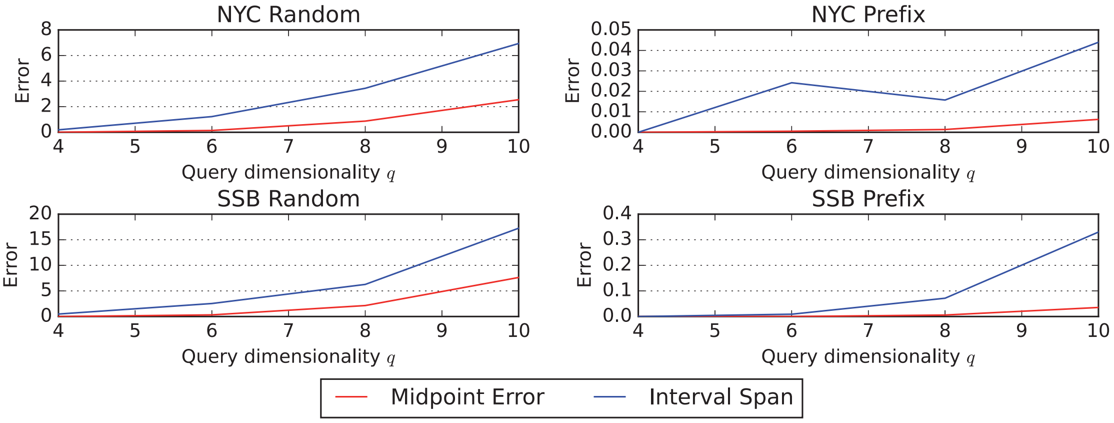
Figure 6.10: Average frequency count for cuboid dimensionality in Sudokube approach when the query dimensionality is varied. Histograms for both the original dimensionality of the materialized cuboid and the dimensionality after it is projected to a subset of the query dimensions are shown.



(a) Execution time



(b) Degrees of freedom



(c) Error

Figure 6.11: Average values for execution time, degrees of freedom, and error for the linear programming solver run in batch mode when the query dimensionality is varied

dimensionality of fetched cuboids results in an exponential boost in fetch time. The fetch time for Prefix cubes is lower compared to Random cubes due to fewer projected cuboids. Contrary to what we see with the naive solver, there is barely any difference between the fetch times for the NYC and SSB datasets, as the base cuboid is not being projected, and low-dimensional cuboids in both datasets have comparable sizes.

In general, the solve time increases with query dimensionality, as more instances of linear programming problems are needed to find bounds for the exponentially increasing variables associated with the rising query dimensionality, but this is not always the case. The linear programming solver utilizes a hybrid approach, employing the more computationally intensive simplex algorithm when the degrees of freedom are low, and a simple bound algorithm when the degrees of freedom are high. We set the threshold for the degrees of freedom at 30, which results in significantly higher solve times for cases where degrees of freedom are slightly below 30, compared to those slightly above 30. This explains why 6-dimensional queries have a longer solving time compared to 8-dimensional queries in the case of data cubes with the Random strategy. In the case of Prefix cubes, for most queries, there is at least one materialized cuboid that yields the entire query as its projection. Therefore, no simplex algorithm is required, and the solve time is lower.

The high value for the solve time, particularly beyond query dimensionality 8, makes this approach impractical for interactive-speed querying. One reason for the slow solve is the use of high-precision rational numbers to represent the coefficients of the variables in the linear equations. Without this, we observed that the simplex algorithm incorrectly concluded that the system of equations is infeasible even when our problem setup guarantees that there exists atleast one feasible solution.

Figure 6.11b shows how the degrees of freedom of the system of equations obtained from the projections of a query vary with the query dimensionality. The total number of variables, which is the maximum possible value for the degrees of freedom, increases exponentially with query dimensionality. The calculated degrees of freedom in the figure is the size of the basis for the system of equations obtained from the projections of the query. The true degree of freedom is lower than the calculated one when the non-negativity constraint restricts the size of the feasible space for some variables to contain just one value. This happens much more frequently for Prefix cubes, for which the cuboids are more sparse and contain zero as the aggregated measure for several cells compared to Random cubes.

Figure 6.11c shows the average cumulative span of the feasible interval for every variable in the query result, along with the average error comparing the midpoint of every interval with the true value. We observe that the error and the interval span are much smaller for the Prefix data cubes than the Random ones. Both metrics increase in response to an increase in the query dimensionality, reflecting the additional degrees of freedom. △

Experiment 6.5 *Varying Number of Materialized Cuboids on Linear Programming Solver*

Next, we examine the impact of the number of materialized cuboids on the histogram of cuboid dimensionality and how it affects the execution time as well as the accuracy of the bounds produced by the linear programming solver when run in batch mode. We run 100 queries of dimensionality 10 on four data cubes: NYC Random, NYC Prefix, SSB Random, and SSB Prefix. The minimum dimensionality parameter for selecting the cuboids to materialize is set to 14 for SSB and 18 for NYC.

Figure 6.12 shows the histogram for the cuboid dimensionality, both before and after the projection to dimensions present in the query. We observe that in the case of Random data cubes, the increase in the number of materialized cuboids leads to an increase in the cuboids selected during the prepare phase. On the other hand, we observe a different trend for Prefix data cubes – the number of prepared cuboids first increases with an increase in the total number of materialized cuboids up to 2^{12}, then it starts decreasing. The reason is clear from the histogram of dimensionality after projection in Figure 6.12b and degrees of freedom in Figure 6.13b. In the case of Random cubes, the overlap with the query is so low that little information is known about the query, and the degree of freedom is very high. On the other hand, in the case of Prefix cubes, there is significant overlap with the query that the additional materialized cuboids increased the number of projections with dimensionality close to the query, so there is no need to fetch the low-dimensional projections.

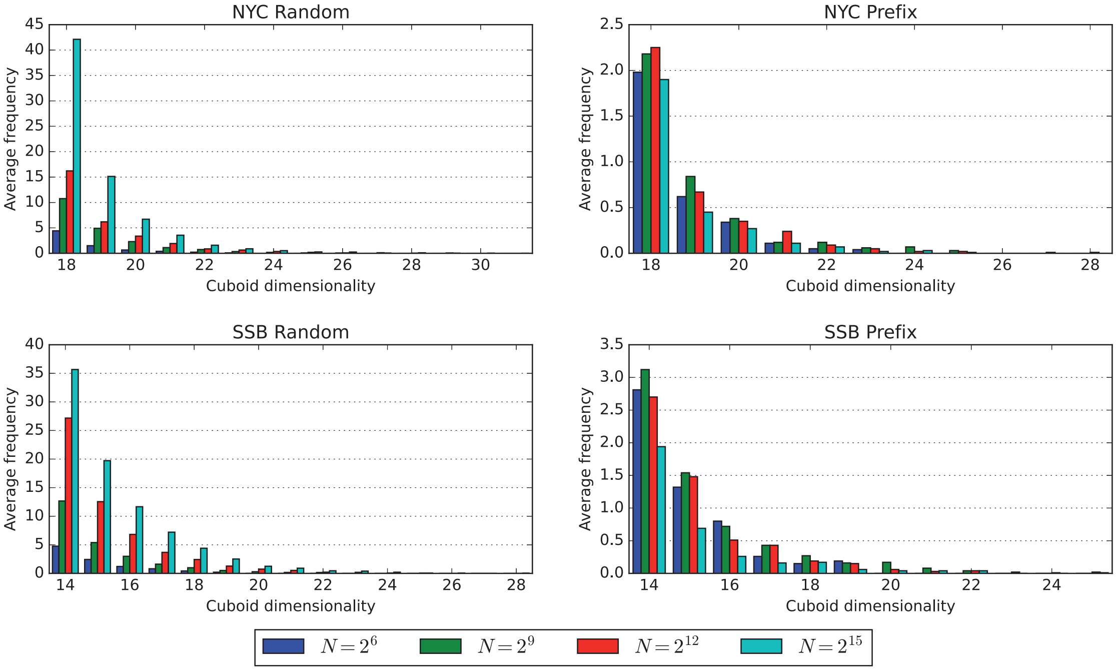
Figure 6.13a shows how the time spent by the linear programming solver on prepare, fetch and solve is affected by the number of materialized cuboids. The prepare time increases linearly with the number of materialized cuboids, as in the case of the naive solver. The fetch time increases in response to the greater number as well as the larger dimensionality of cuboids being fetched. The solve time is largely unaffected as it mostly depends on the number of variables in the query result, which is constant in this experiment.

With the additional information from the greater number of cuboids being fetched, the degree of freedom for the solution space comes down, as shown in Figure 6.13b. The reduction of the degrees of freedom is more significant in the Prefix cubes because the additional cuboids yield relatively high-dimensional projections compared of the query.

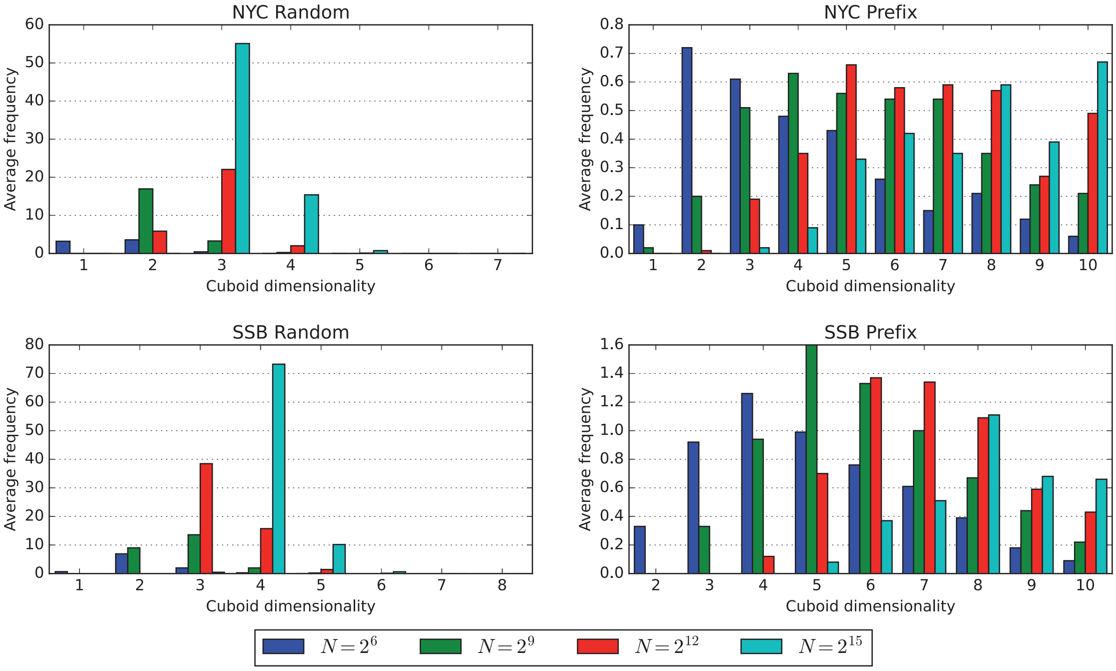
Finally, Figure 6.13c shows how the cumulative interval span and the error for the midpoints of the interval of feasible values for the query result compared to the true result. Both metrics reduce with an increase in the number of materialized cuboids in all four data cubes. △

Experiment 6.6 *Varying Minimum Dimensionality of Materialized Cuboids on Linear Programming Solver in Batch Mode*

We run experiments to study the impact of the minimum dimensionality parameter for materializing cuboids on the linear programming solver in batch mode. We run 100 10-dimensional queries on data cubes built on NYC and SSB datasets with Prefix and Random strategies. The

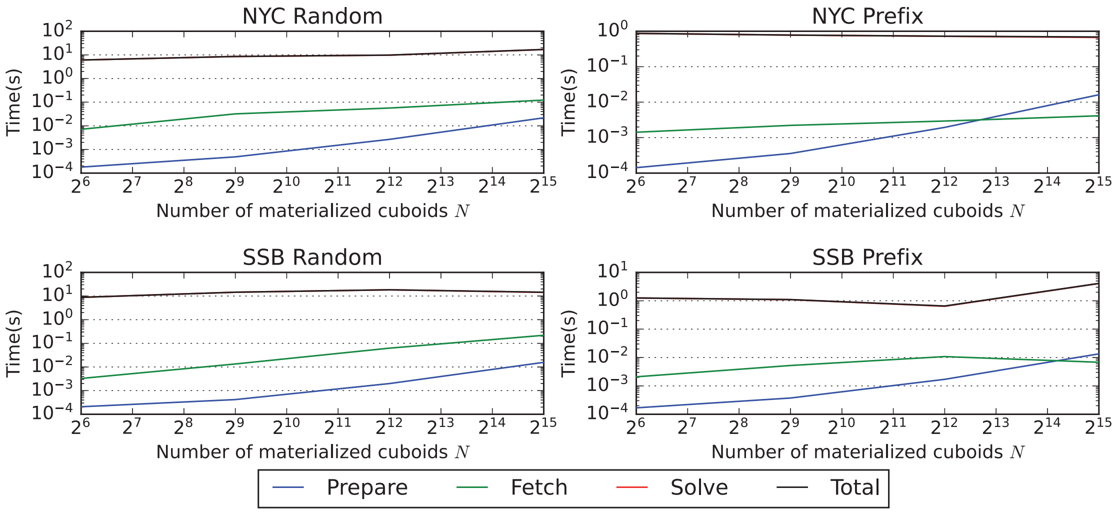


(a) Dimensionality before projection

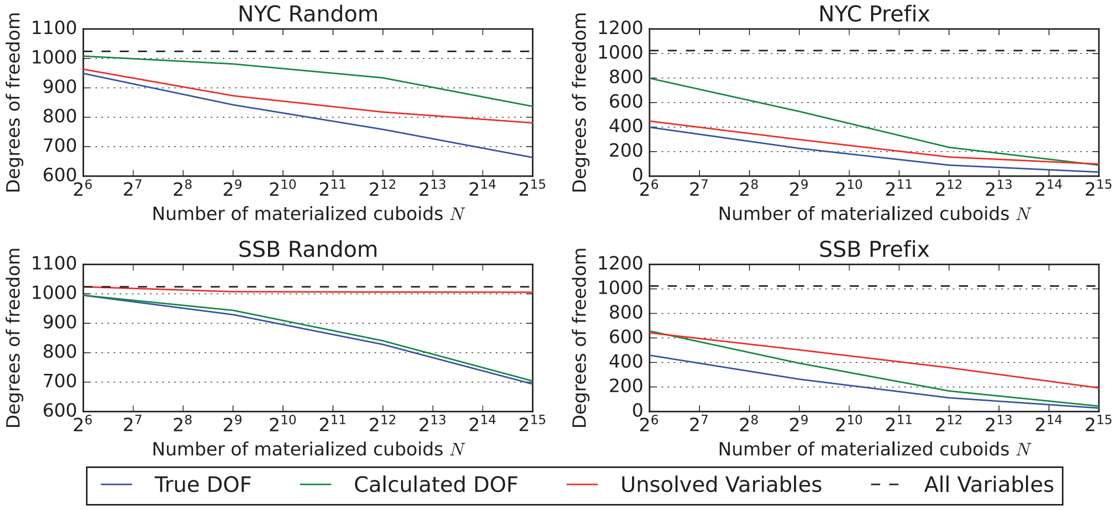


(b) Dimensionality after projection

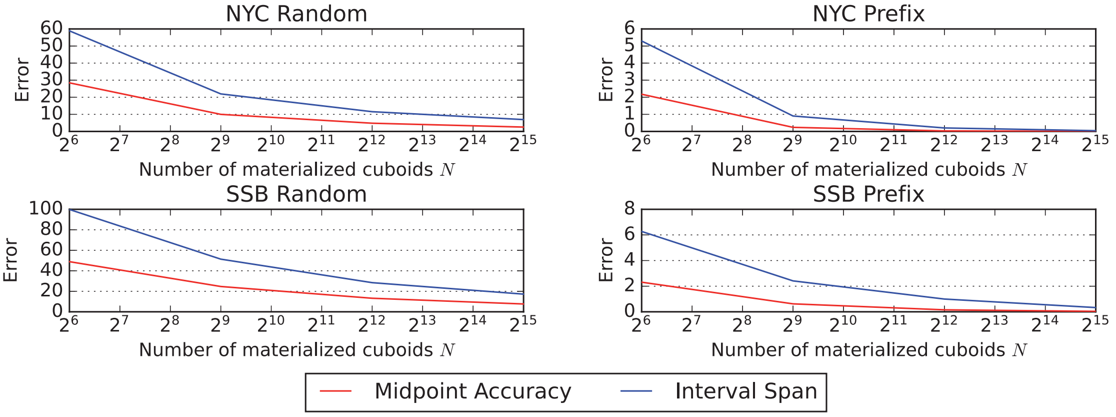
Figure 6.12: Average frequency count for cuboid dimensionality in Sudokube approach when the number of materialized cuboids is varied. Histograms for both the original dimensionality of the materialized cuboid and the dimensionality after it is projected to a subset of the query dimensions are shown.



(a) Execution time



(b) Degrees of freedom



(c) Error

Figure 6.13: Average values for execution time, degrees of freedom, and error for the linear programming solver run in batch mode when the number of materialized cuboids is varied

data cubes are built with the total number of cuboids set to 2^{15} and the experiment is run for different values of the minimum dimensionality d_{min}.

In Figure 6.14a, we observe that the histogram of dimensionality of the source cuboid is shifted whenever the minimum dimensionality is increased. The maximum number of cuboids are fetched from cuboids with dimensionality equal to d_{min} because this is also the dimensionality where the most number of cuboids are materialized. The number of fetched cuboids increases in the case of data cubes with the Random strategy and it decreases for data cubes with the Prefix strategy. The frequency of cuboids of some dimensionality k fetched for answering queries decreases following the same pattern as the total number of cuboids materialized with dimensionality k. The increased value of the minimum dimensionality translates to an increase in the dimensionality after projection due to the increased overlap in the Prefix strategy, eliminating the need for fetching lower-dimensional projections. Figure 6.14b confirms the increase in the dimensionality after the intersection with the query when the dimensionality of materialized cuboids increases.

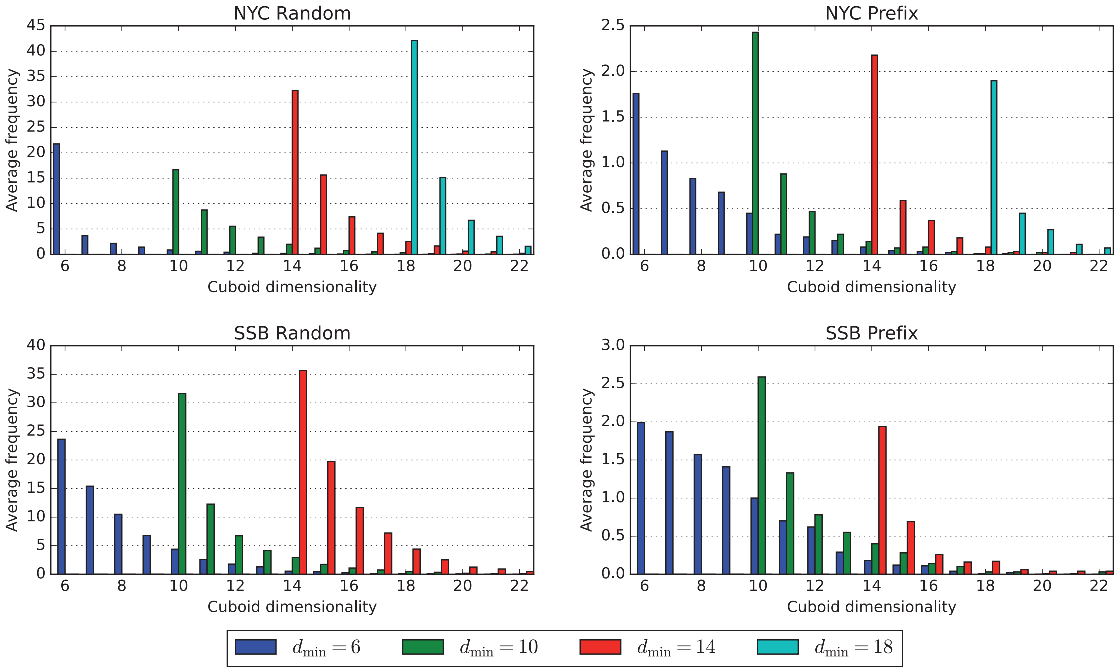
We observe the impact of minimum dimensionality of materialized cuboids on the execution time for linear programming solver in Figure 6.15a. The minimum dimensionality hardly affects the prepare time, but the fetch time increases as larger cuboids are fetched. The solve time also is largely unaffected except for the SSB Prefix data cube, where the average solve time increases due to some outlier queries with a long running time for the simplex algorithm.

Increasing the minimum dimensionality reduces the degrees of freedom, as can be seen in Figure 6.15b. The increased dimensionality of the materialized cuboids leads to an increase in the size of the intersection of their dimensions with the query, which yields more equations that contribute to the increase in the size of the basis.

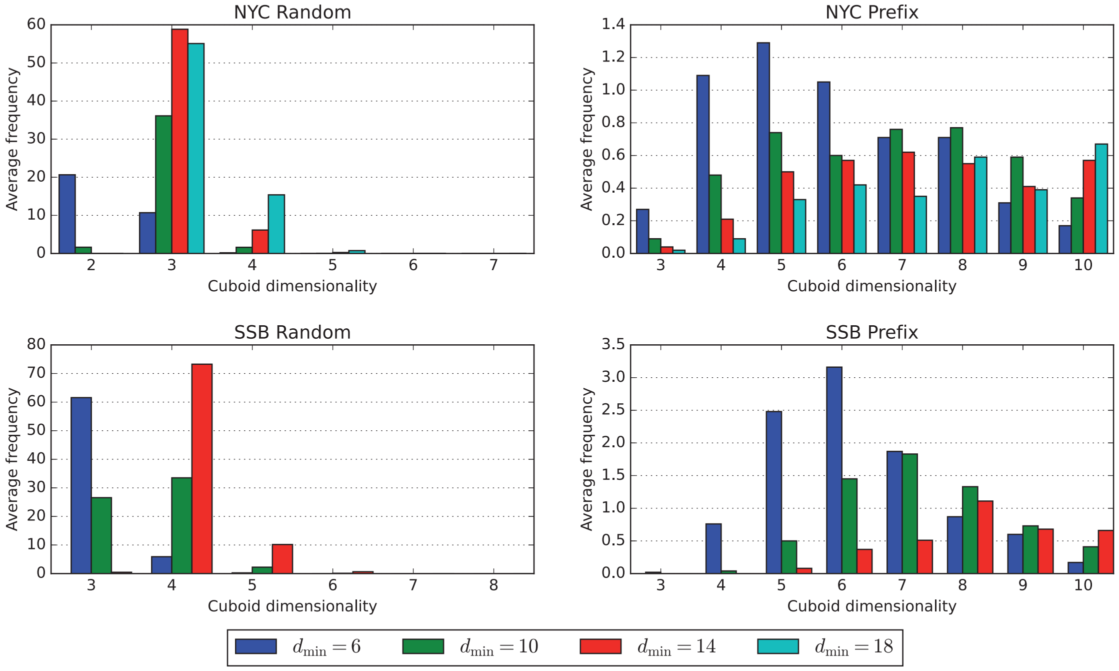
The impact of increasing the dimensionality of materialized cuboids on the accuracy of the result can be seen in Figure 6.15c. The reduced degree of freedom reduces the span of the feasible interval for each query variable and also reduces the error of the midpoint compared to the true solution. △

Experiment 6.7 *Studying Improvement of Error over Time for Linear Programming Solver in Online Mode While Varying Query Dimensionality*

Finally, we study how the approximation quality improves over time when queries are answered using the linear programming solver in the online mode. We pick 100 queries each of various dimensionality and run them on data cubes built on NYC and SSB data sets using Random and Prefix strategies. We use data cubes with 2^{15} materialized cuboids with minimum dimensionality 14 for SSB and 18 for NYC. For each query, we note down the time and the most up-to-date bounds returned by the linear programming solver after processing every fetched cuboid. We use the cumulative span for the bounds returned by the solver for assessing the quality of the approximation.

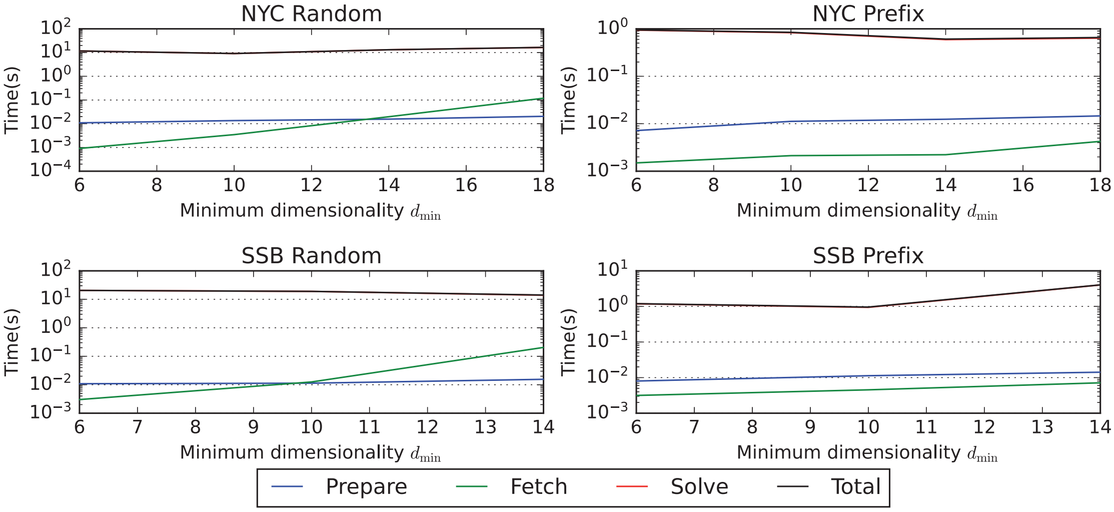


(a) Dimensionality before projection

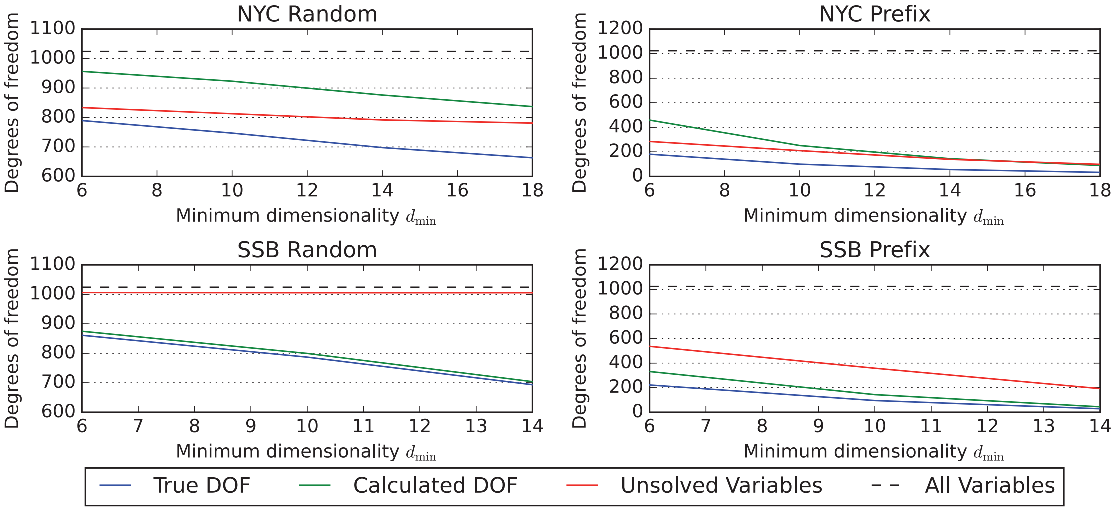


(b) Dimensionality after projection

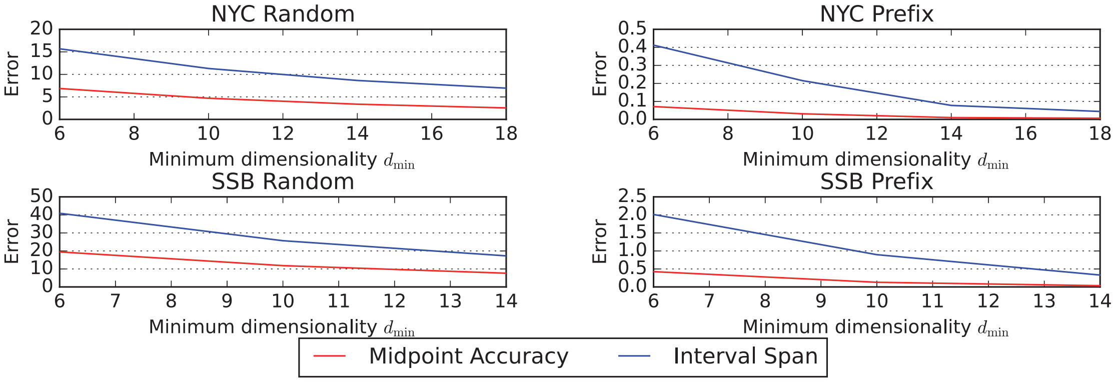
Figure 6.14: Average frequency count for cuboid dimensionality in Sudokube approach when the minimum dimensionality of materialized cuboids is varied. Histograms for both the original dimensionality of the materialized cuboid and the dimensionality after it is projected to a subset of the query dimensions are shown.



(a) Execution time



(b) Degrees of freedom



(c) Error

Figure 6.15: Average values for execution time, degrees of freedom, and error for the linear programming solver run in batch mode when the minimum dimensionality of materialized cuboids is varied

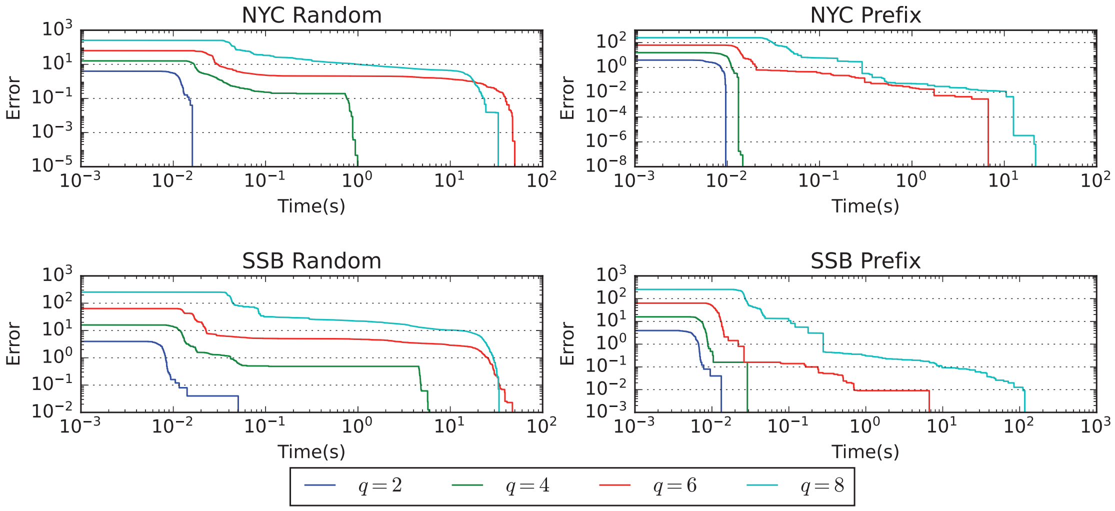


Figure 6.16: Improvement of error over time for linear programming solver in online mode for various query dimensionality

We plot the average error at various times during the execution for various query dimensionality in Figure 6.16. First of all, we observe that the total execution time is much more in online mode compared to the batch mode for the same query dimensionality. Unlike the batch mode, in online mode, the solving algorithm is invoked after every cuboid is fetched instead of just once at the end. When solve time is the dominating cost, as is the case here, invoking solve multiple times will increase the total execution time proportionally as well. The figure also shows there is really no point in running queries in online mode as it is currently implemented when solve time is the dominant cost. It is much cheaper to wait until more cuboids are fetched before solving the query. As a consequence, the linear programming solver in the online mode is interactive only for queries up to dimensionality 4. △

To encapsulate, the performance of the linear programming solver has been comprehensively examined in four distinct experiments. The first three experiments delve into the influence of query dimensionality, the quantity of materialized cuboids, and the minimum dimensionality of these materialized cuboids on the solver's execution time and the accuracy of its result, when operated in batch mode. Another experiment provides insight into the solver's online mode operation, illustrating the evolving accuracy over time for varied query dimensionality.

Across all experiments, we observe the solver's superior performance in terms of speed and precision for the Prefix cubes. In contrast, the solver exhibited longer execution times and reduced accuracy for Random cubes, due to a smaller fraction of materialized cuboids compared to the space of all possible query cuboids.

The solver effectively leveraged the non-negativity constraints in multiple instances, reducing degrees of freedom and subsequently enhancing the accuracy of the results, particularly in the case of Prefix cubes. When the number of materialized cuboids increased, the results displayed enhanced precision without significantly impacting the total execution time. This was largely due to the fact that the increased cuboid count primarily affected the prepare time, which only constitutes a minor portion of the total execution time.

Similarly, increasing the minimum dimensionality of the materialized cuboids led to improved result accuracy without substantially affecting the overall execution time, as this primarily influenced the fetch time. Notably, except for queries of very low dimensionality, the solve time remained the most dominant factor contributing to the total execution time.

Nonetheless, the combination of the high computational complexity of the simplex algorithm and the requirement for high-precision numerical operations hinders the solver's practicality for queries with dimensionality beyond 8. Overall, the linear programming solver demonstrates promise in certain scenarios, but faces challenges for general applicability.

6.3 Solving Queries Using Moments

The linear programming approach discussed in the previous section gives tight bounds for values of the query cuboid from its projections. However, when the number of degrees of freedom of the solution is high, the intervals are usually large and may not provide helpful insights for the given query. We now discuss an alternative approach that returns the most likely values for the cuboid even when we have numerous degrees of freedom. This approach assumes that the extreme values allowed by the many degrees of freedom are possible but highly unlikely.

In this approach, we characterize a cuboid C by its moments. We define a moment for every projection of C, and the complete set of moments uniquely determines the cuboid. In particular, for some query Q, when only some projections of the query result C_Q are known, we can only compute the moments for those projections, and the cuboid C_Q cannot be precisely reconstructed. We approximate the cuboid instead by extrapolating unknown moments from the known ones.

6.3.1 Moments of a Cuboid

In Section 4.1.2, we saw how the binary cuboids in Sudokube resemble probability distributions. The main difference between cuboids and probability distributions is that the entries in a probability distribution are always non-negative and sum up to 1, whereas the entries in cuboids don't necessarily satisfy these properties. However, in this thesis, we are focusing only on non-negative measure values, and the other difference exists only as a normalization factor. Therefore, we generalize Equation (2.5) to define raw moments for binary cuboids by scaling them up by the total measure value *total*, as shown below.

$$\mu(\boldsymbol{x}) = total \cdot \mathbb{E}\left[\prod_{i \in \mathbb{1}_{\boldsymbol{x}}} X_i\right] \tag{6.1}$$

The generalized moment associated with the zero vector $\mu(\mathbf{0})$ is equal to the total sum of all entries *total*, and we shall use them interchangeably. These moments are related to the measure values in cuboids, as the following proposition states.

Proposition 8. *For any* $\boldsymbol{x} \in \{0,1\}^I$, $\mu_I(\boldsymbol{x}) = C_J(\boldsymbol{x}_{\downarrow J})$, *where* $J = \mathbb{1}_{\boldsymbol{x}}$.

Proof. From its definition, $\mu_I(\boldsymbol{x})$ is the expected value of a product of some random variables multiplied by a scaling factor, which translates to a sum over measure values of cells that map all the dimensions in J to 1. All these vectors are consistent with each other on dimensions in J, and the sum of their measure values forms the measure value of a cell in the cuboid C_J, as shown below.

$$\begin{aligned}
\mu_I(\boldsymbol{x}) &= total \cdot \mathbb{E}\left[\prod_{i \in \mathbb{1}_{\boldsymbol{x}}} X_i\right] \\
&= total \cdot \sum_{\boldsymbol{y} \in \{0,1\}^I} \left(\prod_{i \in \mathbb{1}_{\boldsymbol{x}}} y_i\right) p_I(\boldsymbol{y}) \\
&= \sum_{\boldsymbol{y} \in \{0,1\}^I} \left(\prod_{i \in \mathbb{1}_{\boldsymbol{x}}} y_i\right) C_I(\boldsymbol{y}) \\
&= \sum_{\mathbb{1}_{\boldsymbol{y}} \supseteq \mathbb{1}_{\boldsymbol{x}}} C_I(\boldsymbol{y}) \\
&= \sum_{\boldsymbol{y}_{\downarrow J} = \boldsymbol{x}_{\downarrow J}} C_I(\boldsymbol{y}) = C_J(\boldsymbol{x}_{\downarrow J})
\end{aligned}$$

☐

Proposition 8 allows us to compute moments of some cuboid from its projections. All moments of a cuboid can also be efficiently computed directly from the cuboid as established in the following proposition.

Proposition 9. *For any* $I \subseteq [n]$, *the vectorized cuboid* $\boldsymbol{C}_I$ *and its moments* $\boldsymbol{\mu}_I$ *are related to each other by the following two equations, where* $\otimes$ *denotes the Kronecker product (applied* $|I|$ *times in the equations) and* $\boldsymbol{M} = \left[\begin{smallmatrix} 1 & 1 \\ 0 & 1 \end{smallmatrix}\right]$ *and* $\boldsymbol{W} = \boldsymbol{M}^{-1} = \left[\begin{smallmatrix} 1 & -1 \\ 0 & 1 \end{smallmatrix}\right]$:

$$(1)\ \boldsymbol{\mu}_I = \boldsymbol{M}^{\otimes |I|} \boldsymbol{C}_I \quad \text{and} \quad (2)\ \boldsymbol{C}_I = \boldsymbol{W}^{\otimes |I|} \boldsymbol{\mu}_I$$

Proof. Statement (2) immediately follows from Statement (1) because of the properties of Kronecker product given that $\boldsymbol{W} = \boldsymbol{M}^{-1}$. We only need to prove Statement (1), which we do by induction on the size of the cuboid $|I|$. Let the elements of I in step k with $|I| = k$ be labeled $i_1, \ldots, i_k$.

Base Case : For $|I| = 1$ from the definition of moments, we know that $\mu_I(\mathbf{0}) = C_I(\mathbf{0}) + C_I(\mathbf{1})$ and $\mu_I(\mathbf{1}) = C_I(\mathbf{1})$. Thus, $\boldsymbol{\mu}_I = \boldsymbol{M}\,\boldsymbol{C}_I$.

Inductive Case : We assume that the statements are true for $|I| = k-1$ and prove them for $|I| = k$. The k-dimensional cuboid can be sliced along the last dimension i_k into two $(k-1)$

dimensional cuboids. Let $\boldsymbol{C}_{I|\boldsymbol{i}}$ denote the two slices and $\boldsymbol{\mu}_{I|\boldsymbol{i}}$ their respective moments for $\boldsymbol{i} = \boldsymbol{0}, \boldsymbol{1} \in \{0,1\}^{\{i_k\}}$. For some $\boldsymbol{x}, \boldsymbol{y} \in \{0,1\}^{I\setminus\{i_k\}}$, we have

$$\begin{aligned} C_{I|\boldsymbol{i}}(\boldsymbol{y}) &= C_I(\boldsymbol{y} \uplus \boldsymbol{i}), && \text{and} \\ \mu_{I|\boldsymbol{i}}(\boldsymbol{x}) &= \sum_{\mathbb{1}_{\boldsymbol{y}} \supseteq \mathbb{1}_{\boldsymbol{x}}} C_{I|\boldsymbol{i}}(\boldsymbol{y}) = \sum_{\mathbb{1}_{\boldsymbol{y}} \supseteq \mathbb{1}_{\boldsymbol{x}}} C_I(\boldsymbol{y} \uplus \boldsymbol{i}). \end{aligned}$$

Furthermore, let $\boldsymbol{\mu}_I^{(\boldsymbol{i})}$ and $\boldsymbol{C}_I^{(\boldsymbol{i})}$ denote the first and second halves of the moments and values of the k-dimensional cuboid for $\boldsymbol{i} = \boldsymbol{0}$ and $\boldsymbol{i} = \boldsymbol{1}$ respectively. Because of the ordering of entries in the vectors $\boldsymbol{C}_I$ and $\boldsymbol{\mu}_I$, for some $\boldsymbol{x}, \boldsymbol{y} \in \{0,1\}^{I\setminus\{i_k\}}$ and $\boldsymbol{j} \in \{0,1\}^{\{i_k\}}$, we have

$$\begin{aligned} C_I^{(\boldsymbol{i})}(\boldsymbol{y}) &= C_I(\boldsymbol{y} \uplus \boldsymbol{i}), && \text{and} \\ \mu_I^{(\boldsymbol{i})}(\boldsymbol{x}) &= \mu_I(\boldsymbol{x} \uplus \boldsymbol{i}) = \sum_{\substack{\mathbb{1}_{\boldsymbol{y}} \supseteq \mathbb{1}_{\boldsymbol{x}} \\ \mathbb{1}_{\boldsymbol{j}} \supseteq \mathbb{1}_{\boldsymbol{i}}}} C_I(\boldsymbol{y} \uplus \boldsymbol{j}). \end{aligned}$$

Clearly, $\boldsymbol{C}_{I|\boldsymbol{i}} = \boldsymbol{C}_I^{(\boldsymbol{i})}$. Furthermore, $\boldsymbol{\mu}_I^{(\boldsymbol{0})} = \boldsymbol{\mu}_{I|\boldsymbol{0}} + \boldsymbol{\mu}_{I|\boldsymbol{1}}$ and $\boldsymbol{\mu}_I^{(\boldsymbol{1})} = \boldsymbol{\mu}_{I|\boldsymbol{1}}$. Using the inductive property, we have $\boldsymbol{\mu}_{I|\boldsymbol{i}} = \boldsymbol{M}^{\otimes(k-1)} \boldsymbol{C}_{I|\boldsymbol{i}}$. Combining them all together, we have

$$\begin{aligned} \boldsymbol{\mu}_I &= \begin{bmatrix} \boldsymbol{\mu}_I^{(\boldsymbol{0})} \\ \boldsymbol{\mu}_I^{(\boldsymbol{1})} \end{bmatrix} = \begin{bmatrix} \boldsymbol{M}^{\otimes(k-1)} & \boldsymbol{M}^{\otimes(k-1)} \\ \boldsymbol{0} & \boldsymbol{M}^{\otimes(k-1)} \end{bmatrix} \begin{bmatrix} \boldsymbol{C}_I^{(\boldsymbol{0})} \\ \boldsymbol{C}_I^{(\boldsymbol{1})} \end{bmatrix} \\ &= \left(\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \otimes \boldsymbol{M}^{\otimes(k-1)} \right) \begin{bmatrix} \boldsymbol{C}_I^{(\boldsymbol{0})} \\ \boldsymbol{C}_I^{(\boldsymbol{1})} \end{bmatrix} = \boldsymbol{M}^{\otimes k} \boldsymbol{C}_I. \end{aligned}$$

□

Example 14. *The moments of the query cuboid* C_Q *for the query* $Q = \{3,1,0\}$ *with* $\boldsymbol{C}_Q = (C_Q(000) \ldots C_Q(111))$ *on the sales data cube are shown below. For every* $\boldsymbol{x} \in \{0,1\}^Q$*, the moment* $\mu(\boldsymbol{x})$ *equal to the last entry in the cuboid from Figure 6.2 containing only the dimensions mapped to* 1 *by* $\boldsymbol{x}$*, as stated in Proposition 8.*

$$\begin{bmatrix} \mu_Q(000) \\ \mu_Q(001) \\ \mu_Q(010) \\ \mu_Q(011) \\ \mu_Q(100) \\ \mu_Q(101) \\ \mu_Q(110) \\ \mu_Q(111) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 3 \\ 1 \\ 7 \\ 2 \\ 3 \\ 0 \end{bmatrix} = \begin{bmatrix} 17 \\ 4 \\ 7 \\ 1 \\ 12 \\ 2 \\ 3 \\ 0 \end{bmatrix} = \begin{bmatrix} C_{\emptyset}(***) \\ C_{\{0\}}(**1) \\ C_{\{1\}}(*1*) \\ C_{\{1,0\}}(*11) \\ C_{\{3\}}(1**) \\ C_{\{3,0\}}(1*1) \\ C_{\{3,1\}}(11*) \\ C_{\{3,1,0\}}(111) \end{bmatrix}$$

6.3.2 Cuboid Transformations

Proposition 9 establishes a 1-to-1 correspondence between the cuboid $\boldsymbol{C}_I$ and its moments $\boldsymbol{\mu}_I$ using transformation matrices $\boldsymbol{M}^{\otimes|I|}$ and $\boldsymbol{W}^{\otimes|I|}$. Specifically, $\boldsymbol{\mu}_I$ can be taken as an alternative representation of the cuboid. In the general case, we are interested in transformations represented by matrices that have a very convenient structure in the form of Kronecker products

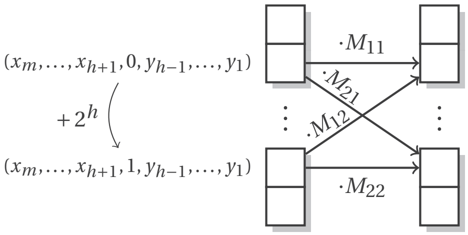

$(x_m, \dots, x_{h+1}, 0, y_{h-1}, \dots, y_1)$
$+2^h$
$(x_m, \dots, x_{h+1}, 1, y_{h-1}, \dots, y_1)$
$\cdot M_{11}$
$\cdot M_{21}$
$\cdot M_{12}$
$\cdot M_{22}$

Figure 6.17: Data flow on array **A** in Algorithm 7, innermost loop, shown for $I = \{1, \dots, m\}$ and $\boldsymbol{M} = \begin{pmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{pmatrix}$

of 2×2 matrices, one for each dimension in I. This is key to being able to carry out such transformations in an efficient way.

As established in Proposition 3, a Kronecker product of several matrices can be expressed as a product of expressions involving each matrix separately. We can use this result to split some linear transformation between two representations having the above-mentioned structure into a series of linear transformations for each dimension in I. This property allows us to implement fast versions of these transformations, much like the Fast Fourier Transform [24] and similar transformations [32], [33]. Furthermore, because of the commutativity for the product of expressions for each dimension, the transformations for all the dimensions can be performed in any order.

Algorithm 7 shows the algorithm for the `Transform` function that applies a 2×2 linear transformation for some generic matrix $\boldsymbol{M}$ with respect to some dimension h (with ordinal position k within the set I) on an array **A**. This corresponds to the transformation of **A** by the k-th term of the matrix product on the right-hand side of (2.8). In the algorithm, we iterate over pairs of elements in the array partitioned along dimension h. For efficient implementation of this algorithm using integer encodings of the binary vectors indexing the array **A**, we describe the iteration in terms of two loops iterating over contiguous ranges of vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ whose sum indexes the first half of the pair of elements being considered. The second half of the pair is obtained by adding the unit vector in dimension h to the sum of $\boldsymbol{x}$ and $\boldsymbol{y}$.

Algorithm 7: Efficient implementation of the transformation algorithm

```
input  : Array A indexed over $\{0,1\}^I$ on which transformation $\boldsymbol{M} \in \mathbb{R}^{2\times 2}$ is to be applied
         corresponding to dimension $h \in I$.
output : Same array A after in-place transformation
def Transform(A, M, h):
    Let $\boldsymbol{e}_h \in \{0,1\}^I$ be the unit vector along $h$
    foreach $\boldsymbol{x} \in \{0,1\}^I$ with $x_i = 0$ for all $i \geq h$ do
        foreach $\boldsymbol{y} \in \{0,1\}^I$ with $y_i = 0$ for all $i \leq h$ do
            $\begin{pmatrix} \mathbf{A}[\boldsymbol{x}+\boldsymbol{y}] \\ \mathbf{A}[\boldsymbol{x}+\boldsymbol{y}+\boldsymbol{e}_h] \end{pmatrix} \leftarrow \boldsymbol{M} \cdot \begin{pmatrix} \mathbf{A}[\boldsymbol{x}+\boldsymbol{y}] \\ \mathbf{A}[\boldsymbol{x}+\boldsymbol{y}+\boldsymbol{e}_h] \end{pmatrix}$
    return A
```

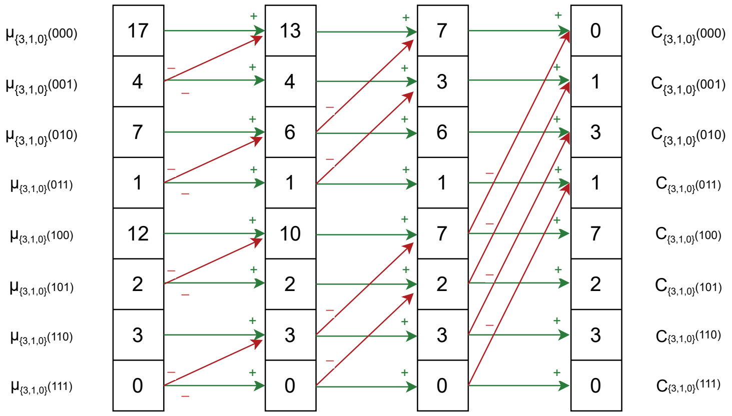


Figure 6.18: Computations involved in Algorithm 7 when applied to transform $\boldsymbol{\mu}_Q$ back to $\boldsymbol{C}_Q$

The execution of Algorithm 7 for the set $I = \{1, \ldots, m\}$ is simulated in Figure 6.17. For this set I, the ordinal position of any dimension h is h. The figure also illustrates how Algorithm 7 implements the expression $\boldsymbol{I}_2^{\otimes(h-1)} \otimes \boldsymbol{M} \otimes \boldsymbol{I}_2^{\otimes(m-h)}$ for the transformation matrix $\boldsymbol{M}$.

Algorithm 8 shows a function `CuboidToRawMoments` that uses the `Transform` function to efficiently convert a cuboid $\boldsymbol{C}_I$ to its representation using raw moments $\boldsymbol{\mu}_I$ following Proposition 9. A similar function `RawMomentsToCuboid` can be obtained for transforming the moments $\boldsymbol{\mu}_I$ back to cuboid $\boldsymbol{C}_I$ by passing $\boldsymbol{W}$ instead of $\boldsymbol{M}$ as the argument to the `Transform` function.

Algorithm 8: Transformation of $\boldsymbol{C}_I$ into $\boldsymbol{\mu}_I$

```
input  : Cuboid $\boldsymbol{C}_I$
output : Array $\mathbf{A}$ containing the values of the vector $\boldsymbol{\mu}_I$
def CuboidToRawMoments($I$, $\boldsymbol{C}_I$):
    Initialize array $\mathbf{A}$ with the contents of $\boldsymbol{C}_I$
    $\boldsymbol{M} \leftarrow \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$
    foreach $h \in I$ do
        $\mathbf{A} \leftarrow$ Transform($\mathbf{A}$, $\boldsymbol{M}$, $h$)
    return $\mathbf{A}$ as $\boldsymbol{\mu}_I$
```

Example 15. *Given the vector* $\boldsymbol{\mu}_Q$ *comprising all 8 moments of the query* $Q = \{3, 1, 0\}$*, Figure 6.18 shows the computations involved in applying the fast transform algorithm to transform* $\boldsymbol{\mu}_Q$ *to obtain the result* C_Q.

Proposition 10. *The run-time complexity of either transformation in Proposition 9 using the fast transformation algorithm from Algorithm 7, for some set* $I \subseteq [n]$ *of size* m*, is* $\mathcal{O}(m \cdot 2^m)$.

Proof. A single invocation of `Transform` has a run-time complexity that is linear in the size of the array, i.e., $\mathcal{O}(2^m)$. This can be inferred from Algorithm 7 where each element of the array is read twice and written once. The transformations described in Proposition 9 invokes `Transform` once per dimension in I as shown in Algorithm 8. Thus, the overall run-time complexity is $\mathcal{O}(m \cdot 2^m)$. □

6.3.3 Cuboid Approximation from Projections

Given a query Q and a set of known projections C_I for several $I \in \mathscr{I}(Q)$ of the query cuboid C_Q, we shall now see how C_Q can be reconstructed from its projections using their moments. The moment solver approximates the result C_Q from the moments of the available projections. It makes statistical assumptions about the underlying data and extrapolates the missing moments of C_Q from the known ones. Algorithm 9 describes a high-level algorithm for the same. Each available projection $\boldsymbol{C}_I$ is first transformed into its moments $\boldsymbol{\mu}_I$ in Line 3. These moments are copied into the appropriate slot among the moments of the query result $\boldsymbol{\mu}_Q$ following Proposition 1 in Line 5. After all the projections are processed, any missing slot in $\boldsymbol{\mu}_Q$ is filled using some extrapolation technique in Line 8. Finally, the approximate query result $\widetilde{C}_Q$ is obtained by applying the reverse transformation on the extrapolated moments in Line 9.

Algorithm 9: Algorithm for reconstructing C_Q from moments of its projections

```
input  : Query $Q$, cuboids $\boldsymbol{C}_I$ for every $I \in \mathscr{I}(Q)$
output : Reconstructed cuboid $\widetilde{C}_Q$
[1]  $\mathsf{A} \leftarrow$ new array of size $2^{|Q|}$ // for $\boldsymbol{\mu}_Q$
[2]  foreach $I \in \mathscr{I}(Q)$ do
[3]      $\boldsymbol{\mu}_I \leftarrow$ CuboidToRawMoments($I, \boldsymbol{C}_I$)
[4]      foreach $\boldsymbol{y} \in \{0,1\}^I$ do
[5]          $\mathsf{A}[\boldsymbol{y}_{\uparrow Q}] \leftarrow \mu_I(\boldsymbol{y})$ // from Proposition 1
[6]          mark index $\boldsymbol{y}_{\uparrow Q}$ as known
[7]  foreach $\boldsymbol{q} \in \{0,1\}^Q$ that is not marked as known do
[8]      $\mathsf{A}[\boldsymbol{q}] \leftarrow$ extrapolated value of $\mu_Q(\boldsymbol{q})$ using some method
[9]  $\widetilde{C}_Q \leftarrow$ RawMomentsToCuboid($Q, \mathsf{A}$)
[10] return $\widetilde{C}_Q$
```

Example 16. *7 moments of the query $Q = \{3,1,0\}$ can be computed from its projections $\{1,0\}$, $\{3,0\}$ and $\{3,1\}$ as shown in Figure 6.19 following Proposition 1. The moment $\mu_Q(111)$ is unknown and has to be extrapolated from the other moments by some means.*

Let us now examine some methods for extrapolating moments of the query cuboid that cannot be computed from any of the materialized cuboids. We denote the extrapolated moment indexed by some vector $\boldsymbol{q} \in \{0,1\}^Q$ using $\tilde{\mu}_Q(\boldsymbol{q})$. If this value is computable from the available projections of the query, then we set it to its exact value $\tilde{\mu}_Q(\boldsymbol{q}) = \mu_Q(\boldsymbol{q})$ and otherwise, we set it using one of the following approaches.

ZeroRawMoment

In this approach, we simply set the extrapolated value of the unknown raw moments $\mu_Q(\boldsymbol{q})$ as zero. This is the simplest extrapolation method, and produces an approximate query result with all cells $\widetilde{C}_Q(\boldsymbol{r})$ with $\mathbb{1}_{\boldsymbol{r}} \supseteq \mathbb{1}_{\boldsymbol{q}}$ set to zero.

$$\tilde{\mu}_Q(\boldsymbol{q}) = 0$$

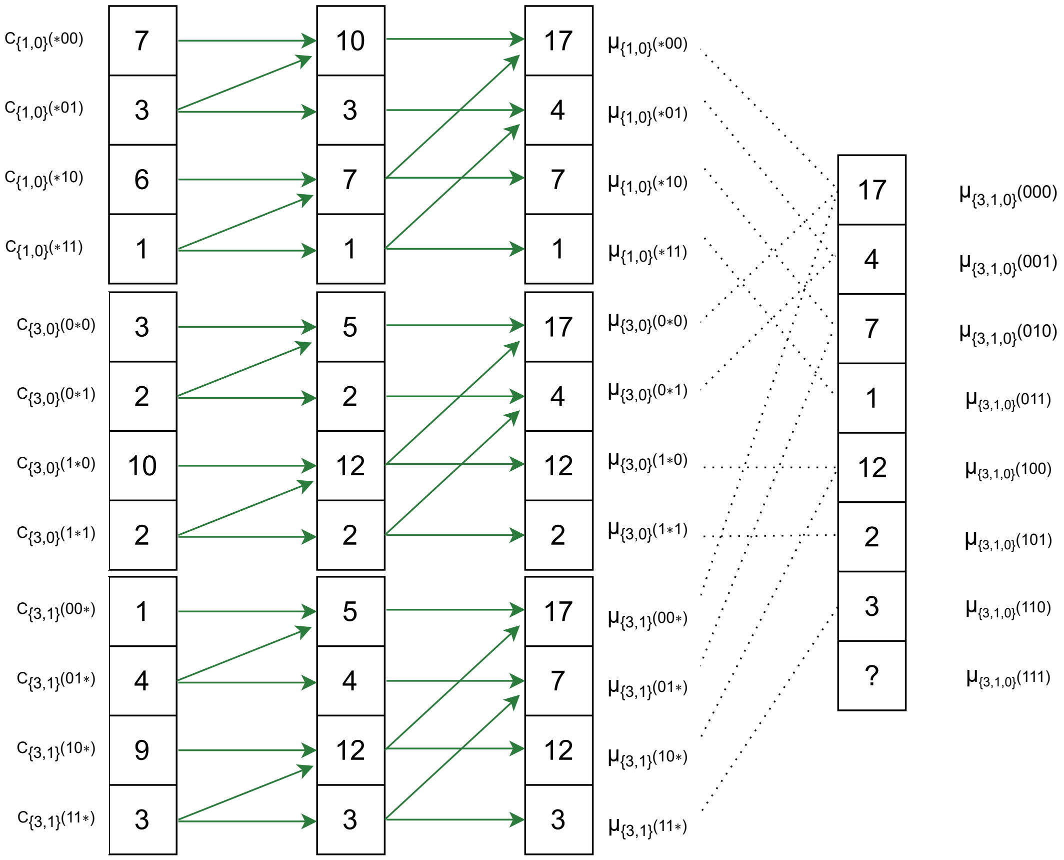


Figure 6.19: Computing seven out of eight moments μ_Q of the query cuboid C_Q from its three projections. The last moment is not known and has to be extrapolated from the others.

HalfPowerK

This approach sets the extrapolated values of unknown moments to be equal to the corresponding moments in a uniformly distributed cuboid. In this approach, for any unknown moment indexed by $\boldsymbol{q} \in \{0,1\}^Q$, we set

$$\widetilde{\mu}_Q(\boldsymbol{q}) = \frac{total}{2^{|\mathbb{1}_{\boldsymbol{q}}|}}$$

ParentAverage

In this approach, we process the unknown moments in ascending order of $\boldsymbol{q}$. We set the value of any unknown $\mu(\boldsymbol{q})$ to be half of the average values of moments $\mu(\boldsymbol{r})$ such that $\boldsymbol{r} \in \{0,1\}^Q$ has exactly one less dimension mapping to 1 compared to $\boldsymbol{r}$ as shown below

$$\widetilde{\mu}_Q(\boldsymbol{q}) = \frac{1}{|\mathbb{1}_{\boldsymbol{q}}|} \sum_{\substack{\mathbb{1}_{\boldsymbol{r}} \subset \mathbb{1}_{\boldsymbol{q}} \\ |\mathbb{1}_{\boldsymbol{r}}| = |\mathbb{1}_{\boldsymbol{q}}|-1}} \frac{\widetilde{\mu}_Q(\boldsymbol{r})}{2}$$

AncestorAverage

This follows a similar approach to ParentAverage, but takes a weighted average of all preceding moments for any unknown moment indexed by $\boldsymbol{q} \in \{0,1\}^Q$.

$$\widetilde{\mu}_Q(\boldsymbol{q}) = \frac{1}{2^{|\mathbb{1}_{\boldsymbol{q}}|} - 1} \sum_{k=1}^{|\mathbb{1}_{\boldsymbol{q}}|} \sum_{\substack{\mathbb{1}_{\boldsymbol{r}} \subset \mathbb{1}_{\boldsymbol{q}} \\ |\mathbb{1}_{\boldsymbol{r}}| = |\mathbb{1}_{\boldsymbol{q}}| - k}} \frac{\widetilde{\mu}_Q(\boldsymbol{r})}{2^k}$$

ParentMin

This approach is similar to ParentAverage, but takes the minimum value of all the moments instead of half the average value. This is the upper bound for the possible values of any unknown $\mu(\boldsymbol{q})$ based on Fréchet's inequality [35].

$$\widetilde{\mu}_Q(\boldsymbol{q}) = \min_{\substack{\mathbb{1}_{\boldsymbol{r}} \subset \mathbb{1}_{\boldsymbol{q}} \\ |\mathbb{1}_{\boldsymbol{r}}| = |\mathbb{1}_{\boldsymbol{q}}| - 1}} \widetilde{\mu}_Q(\boldsymbol{r})$$

FrechetMid

In this approach, we set the value of any unknown moment as the average of its Fréchet upper and lower bounds.

$$\widetilde{\mu}_Q(\boldsymbol{q}) = \frac{1}{2}\left(\min_{\substack{\mathbb{1}_{\boldsymbol{r}} \subset \mathbb{1}_{\boldsymbol{q}} \\ |\mathbb{1}_{\boldsymbol{r}}| = |\mathbb{1}_{\boldsymbol{q}}| - 1}} \widetilde{\mu}_Q(\boldsymbol{r}) \quad + \quad \max\left\{ 0, \left(1 - |\mathbb{1}_{\boldsymbol{q}}|\right) total + \sum_{\substack{\mathbb{1}_{\boldsymbol{r}} \subset \mathbb{1}_{\boldsymbol{q}} \\ |\mathbb{1}_{\boldsymbol{r}}| = |\mathbb{1}_{\boldsymbol{q}}| - 1}} \widetilde{\mu}_Q(\boldsymbol{r}) \right\} \right)$$

ZeroCentralMoment

We extrapolate the value of any unknown raw moment $\mu(\boldsymbol{q})$ by assuming that the corresponding central moment $\sigma(\boldsymbol{q})$ is zero. The central moments for probability distributions defined in Equation (2.6) can be generalized to cuboids in the same way raw moments were generalized in Equation (6.1). The relationship between μ_I and σ_I as proposed in Lemma 2 holds for the generalized moments as well. We extrapolate unknown moments in ascending order of $\boldsymbol{q}$ by setting all unknown central moments to zero based on Lemma 2 as

$$\widetilde{\mu}_Q(\boldsymbol{q}) = \sum_{\substack{\mathbb{1}_{\boldsymbol{r}} \subseteq \mathbb{1}_{\boldsymbol{q}} \\ \mathbb{1}_{\boldsymbol{r}} \in \mathscr{I}(Q)}} \sigma_Q(\boldsymbol{r}) \prod_{i \in \mathbb{1}_{\boldsymbol{q}} \setminus \mathbb{1}_{\boldsymbol{r}}} \theta_i \tag{6.2}$$

Algorithm 10 shows an algorithm for extrapolating the moments by setting unknown central moments to zero. It starts with the state where the 0-D moment that is equal to the total sum and all the 1-D moments equal to each θ_i, are known. It computes the products of various combinations of θ_i in the array $\mathbf{P}$ and initializes $\mathbf{A}(\boldsymbol{q})$, which stores $\mu_Q(\boldsymbol{q})$, as $total \cdot \mathbf{P}(\boldsymbol{q})$. Then for each known projection C_I, the moments $\mu_I(\boldsymbol{y})$ are processed in order $\prec$ for $\boldsymbol{y}$. From Proposition 1, this is the true value for the moment $\mu_Q(\boldsymbol{s})$, where $\boldsymbol{s} = \boldsymbol{y}_{\uparrow Q}$. The value of $\sigma_Q(\boldsymbol{s})$ is then computed as the difference between the true value and the current value of $\mu_Q(\boldsymbol{s})$, which is then used to update the values of moments $\mu_Q(\boldsymbol{q})$ with $\mathbb{1}_{\boldsymbol{q}} \supseteq \mathbb{1}_{\boldsymbol{s}}$. The correctness of this algorithm is given by the following theorem.

Algorithm 10: Algorithm for reconstructing C_Q by assuming unknown central moments are zero

```
input  : Query $Q$, cuboids $\boldsymbol{C}_I$ for every $I \in \mathscr{I}(Q)$
output : Reconstructed cuboid $\widetilde{C}_Q$
$\mathbf{A} \leftarrow$ new array of size $2^{|Q|}$ // for $\tilde{\boldsymbol{\mu}}_Q$
$\mathbf{P} \leftarrow$ new array of size $2^{|Q|}$
mark $\mathbf{0}$ and unit vectors $\boldsymbol{e}_h \in \{0,1\}^Q$ as known
foreach $\boldsymbol{q} \in \{0,1\}^Q$ do
    $\mathbf{P}[\boldsymbol{q}] \leftarrow \prod_{i \in \mathbb{1}_{\boldsymbol{q}}} \theta_i$
    $\mathbf{A}[\boldsymbol{q}] \leftarrow total \cdot \mathbf{P}[\boldsymbol{q}]$
foreach $I \in \mathscr{I}(Q)$ do
    $\boldsymbol{\mu}_I \leftarrow$ CuboidToRawMoments($I, \boldsymbol{C}_I$)
    foreach $\boldsymbol{y} \in \{0,1\}^I$ in ascending order of $\prec$ do
        $\boldsymbol{s} \leftarrow \boldsymbol{y}_{\uparrow Q}$
        $\mu \leftarrow \mu_I(\boldsymbol{y})$ // true value of $\mu_Q(\boldsymbol{s})$
        if $\boldsymbol{s}$ is not marked as known then
            $\sigma \leftarrow \mu - \mathbf{A}[\boldsymbol{s}]$ // difference of true and current value of $\mu_Q(\boldsymbol{s})$
            foreach $\boldsymbol{q} \in \{0,1\}^Q$ such that $\mathbb{1}_{\boldsymbol{q}} \supseteq \mathbb{1}_{\boldsymbol{s}}$ do
                $\mathbf{A}[\boldsymbol{q}] \leftarrow \mathbf{A}[\boldsymbol{q}] + \sigma \cdot \mathbf{P}[\boldsymbol{q} - \boldsymbol{s}]$
            mark index $\boldsymbol{s}$ as known
$\widetilde{C}_Q \leftarrow$ RawMomentsToCuboid($Q, \mathbf{A}$)
return $\widetilde{C}_Q$
```

Theorem 11. *Algorithm 10 computes the extrapolated moments $\tilde{\mu}_Q(\boldsymbol{q})$ of cuboid C_Q representing the result of the query Q according to Equation* (6.2) *as more moments of C_Q are computed from its projections.*

Proof. We use induction on the number of moments of C_Q that become known to prove the correctness of Algorithm 10.

Base Case : Initially, only the total sum referenced by $\mu_Q(\mathbf{0})$ and the 1-D marginals referenced by $\mu(\boldsymbol{e}_h)$ for each $h \in Q$ are known. From its definition, $\sigma_Q(\mathbf{0}) = \mu_Q(\mathbf{0})$ and $\sigma_Q(\boldsymbol{e}_h) = 0$ for any $h \in Q$. Plugging these values into Equation (6.2) gives the following equation for every moment indexed by $\boldsymbol{q} \in \{0,1\}^Q$.

$$\tilde{\mu}_Q(\boldsymbol{q}) = total * \prod_{i \in \mathbb{1}_{\boldsymbol{q}}} \theta_i$$

Inductive Case: We assume that the algorithm is correct after processing a certain number of cuboids and prove that it remains correct after one more step. Let the new moment that is now being added be indexed by $\boldsymbol{s}$. Consider some arbitrary index $\boldsymbol{q} \in \{0,1\}^Q$. In Equation (6.2), the moment $\mu_Q(\boldsymbol{q})$ depends only on the central moments $\boldsymbol{r}$ such that $\mathbb{1}_{\boldsymbol{r}} \subseteq \mathbb{1}_{\boldsymbol{q}}$. Therefore, after computing a new moment $\mu_Q(\boldsymbol{s})$, the only moments that need to be updated to maintain Equation (6.2) are those indexed by specific $\boldsymbol{q}$ such that $\mathbb{1}_{\boldsymbol{s}} \subseteq \mathbb{1}_{\boldsymbol{q}}$. For one such $\widetilde{\mu}_Q(\boldsymbol{q})$, the incremental change to it upon computing the new moment $\mu_Q(\boldsymbol{s})$ is given by

$$\sigma_Q(\boldsymbol{s}) \prod_{i \in \mathbb{1}_{\boldsymbol{q}} \setminus \mathbb{1}_{\boldsymbol{s}}} \theta_i.$$

The only thing left to be proved is the correctness of the value of $\mu_Q(\boldsymbol{s})$ computed in Line 13. Since we process moments in ascending order of the indices, all moments $\mu_Q(\boldsymbol{r})$ with $\mathbb{1}_{\boldsymbol{r}} \subsetneq \boldsymbol{s}$ have been computed and processed before. Therefore, the latest extrapolated value $\widetilde{\mu}_Q(\boldsymbol{s})$ before computing the true value is given by Equation (6.3), according to the induction hypothesis. We also have Equation (6.4) that separates the term for $\boldsymbol{s}$ from the other terms in the summation in the formula for $\mu_Q(\boldsymbol{s})$ in Lemma 2.

$$\widetilde{\mu}_Q(\boldsymbol{s}) = \sum_{\mathbb{1}_{\boldsymbol{r}} \subsetneq \mathbb{1}_{\boldsymbol{s}}} \sigma_Q(\boldsymbol{r}) \cdot \prod_{i \in \mathbb{1}_{\boldsymbol{s}} \setminus \mathbb{1}_{\boldsymbol{r}}} \theta_i \tag{6.3}$$

$$\mu_Q(\boldsymbol{s}) = \sigma_Q(\boldsymbol{s}) \cdot 1 + \sum_{\mathbb{1}_{\boldsymbol{r}} \subsetneq \mathbb{1}_{\boldsymbol{s}}} \sigma_Q(\boldsymbol{r}) \cdot \prod_{i \in \mathbb{1}_{\boldsymbol{s}} \setminus \mathbb{1}_{\boldsymbol{r}}} \theta_i \tag{6.4}$$

Combining these equations, we get $\sigma_Q(\boldsymbol{s}) = \mu_Q(\boldsymbol{s}) - \widetilde{\mu}_Q(\boldsymbol{s})$, thus completing the proof. □

Example 17. *Consider the query $Q = \{3,1,0\}$ on the sales data cube. Initially only values of total$= 17$, $\theta_0 = \frac{4}{17}$, $\theta_1 = \frac{7}{17}$ and $\theta_2 = \frac{12}{17}$. The initial values of the extrapolated moments $\widetilde{\mu}_Q(011)$ and $\widetilde{\mu}_Q(111)$ are given by*

$$\widetilde{\mu}_Q(011) = \textit{total} \cdot \theta_0\theta_1 = \tfrac{28}{17}, \textit{ and } \widetilde{\mu}_Q(111) = \textit{total} \cdot \theta_0\theta_1\theta_2 = \tfrac{336}{289}.$$

Once the cuboid $C_{\{1,0\}}$ is fetched, we set $\widetilde{\mu}_Q(011)$ to its true value $\mu_Q(011)$, which is same as $\mu_{\{1,0\}}(*11)$. The updated moments are

$$\widetilde{\mu}_Q(011) = 1, \textit{ and } \widetilde{\mu}_Q(111) = \tfrac{336}{289} + \left(1 - \tfrac{28}{17}\right) \cdot \tfrac{12}{17} = \tfrac{12}{17}.$$

After processing all three projections $C_{\{1,0\}}$, $C_{\{3,1\}}$ and $C_{\{3,0\}}$,

$$\widetilde{\mu}_Q(111) = \tfrac{336}{289} + \left(1 - \tfrac{28}{17}\right) \cdot \tfrac{12}{17} + \left(3 - \tfrac{84}{17}\right) \cdot \tfrac{4}{17} + \left(2 - \tfrac{48}{17}\right) \cdot \tfrac{7}{17} = \tfrac{-26}{289}.$$

From this point on, we ignore the distinction between the true and extrapolated moments of a cuboid and simply refer to them as moments. Once the moment vector is computed through extrapolation, the values of the cuboid can be obtained using the equation using a procedure `RawMomentsToCuboid` that is similar to the procedure `CuboidToRawMoments` described in Algorithm 8. We estimate the quality of the approximated result in terms of an *error.* For any approximate result $\widetilde{C}_Q$ of some query Q with true result C_Q, we define the error in the approximation as the sum of the absolute values of the deviation between the true and the approximated value for each cell divided by the total sum as shown below.

$$error = \sum_{\boldsymbol{q} \in \{0,1\}^Q} \frac{|\widetilde{C}_Q(\boldsymbol{q}) - C_Q(\boldsymbol{q})|}{total} \tag{6.5}$$

Experiment 6.8 *Comparing Moment Extrapolation Techniques in Batch Mode*

We run 100 queries each of varying dimensionality in batch mode on the four data cubes – NYC Random, NYC Prefix, SSB Random, and SSB Prefix, and compare the performance of various techniques to extrapolate the moments we have discussed so far. We fix the total number of materialized cuboids to be 2^{15} and set the minimum dimensionality $d_{\min}$ to be 14 for SSB and 18 for NYC. Figure 6.20 shows the average error for each solving technique for various query dimensionality. The figure shows that the approach setting the unknown moments to match that of a uniformly distributed cuboid yields the worst approximation, while the approach that sets the central moment to zero appears to be the best approach. Furthermore, the error for many of these extrapolation techniques is very high, which we investigate next. △

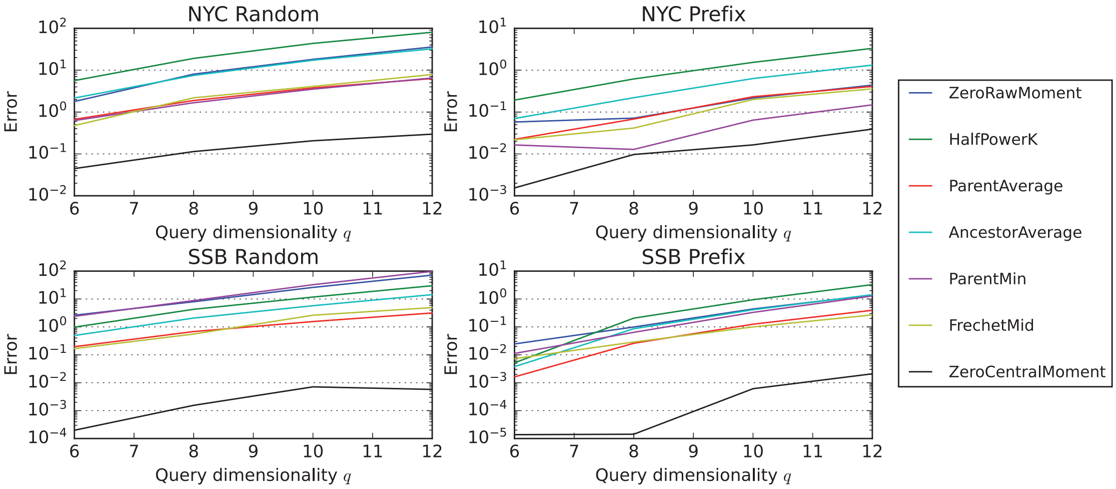


Figure 6.20: Average error for different moment extrapolation techniques run in batch mode for various query dimensionality when no heuristics are applied to improve the error

There are constraints on the feasible values for all moments based on the values of other moments so that the measures in the cuboid they represent are non-negative. Our extrapolation methods may result in extrapolated moments that do not respect these constraints. Since our measure values are all assumed to be non-negative, having negative measure values in an approximated query result lowers its accuracy. Computing exact bounds on the unknown moments to mitigate this problem can be expensive as it would involve solving linear programming problems, as was the case with our linear programming solver in Section 6.2. Instead, a simple approach would be to set the negative values in an approximated cuboid to zero after the reverse transformation from its moments. This can only improve the error as the true values are all assumed to be non-negative.

Another method to improve the error is to apply some simple bounds locally during the transformation from the extrapolated moments to obtain the approximate cuboid. We know from Fréchet's inequality that a moment cannot be greater than another moment that is indexed by a vector containing fewer dimensions mapped to 1, and the moments themselves are non-negative. We, therefore, implemented a function `TransformWithBounds` that is similar to `Transform`, but applies these bounds first. Algorithm 11 shows the function `TransformWithBounds` that is used by `RawMomentsToCuboid` in Algorithm 12 to transform extrapolated moments into values.

Algorithm 11: Transformation algorithm that also applies bounds locally

```
input  : Array $\mathsf{A}$ indexed over $\{0,1\}^I$ on which transformation $\boldsymbol{M} \in \mathbb{R}^{2\times 2}$ is to be applied
         corresponding to dimension $h \in I$.
output : Same array $\mathsf{A}$ after in-place transformation
def TransformWithBounds($\mathsf{A}$, $\boldsymbol{M}$, $h$):
    Let $\boldsymbol{e}_h \in \{0,1\}^I$ be the unit vector along $h$
    foreach $\boldsymbol{x} \in \{0,1\}^I$ with $x_i = 0$ for all $i \geq h$ do
        foreach $\boldsymbol{y} \in \{0,1\}^I$ with $y_i = 0$ for all $i \leq h$ do
            $\mathsf{A}[\boldsymbol{x}+\boldsymbol{y}+\boldsymbol{e}_h] \leftarrow \max(0, \min(\mathsf{A}[\boldsymbol{x}+\boldsymbol{y}+\boldsymbol{e}_h], \mathsf{A}[\boldsymbol{x}+\boldsymbol{y}]))$
            $\begin{pmatrix} \mathsf{A}[\boldsymbol{x}+\boldsymbol{y}] \\ \mathsf{A}[\boldsymbol{x}+\boldsymbol{y}+\boldsymbol{e}_h] \end{pmatrix} \leftarrow \boldsymbol{M} \cdot \begin{pmatrix} \mathsf{A}[\boldsymbol{x}+\boldsymbol{y}] \\ \mathsf{A}[\boldsymbol{x}+\boldsymbol{y}+\boldsymbol{e}_h] \end{pmatrix}$
    return $\mathsf{A}$
```

Algorithm 12: Reverse Transform

```
input  : Array $\mathsf{A}$ containing the values of the vector $\widetilde{\boldsymbol{\mu}}_I$
output : Approximated Cuboid $\widetilde{\boldsymbol{C}}_I$
def RawMomentsToCuboid($I$, $\mathsf{A}$):
    $\boldsymbol{W} \leftarrow \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}$
    foreach $h \in I$ do
        $\mathsf{A} \leftarrow$ TransformWithBounds($\mathsf{A}$, $\boldsymbol{W}$, $h$)
    return $\mathsf{A}$ as $\boldsymbol{\mu}_I$
```

Experiment 6.9 *Comparing Moment Extrapolation Techniques With Bounds in Batch Mode*

We repeat the same experiment with 100 queries and four data cubes and investigate the error for different moment extrapolation techniques when the local bound is applied during the reverse transformation of moments to construct the query cuboid. Figure 6.21 shows the improved error for each extrapolation technique. Applying the bounds significantly reduces the error for every extrapolation technique, and extrapolating moments so that the central moment is zero is still the best approach by far. △

Among all the approaches we have considered in these experiments, the best approach for extrapolating raw moments is to set the corresponding central moments to zero.

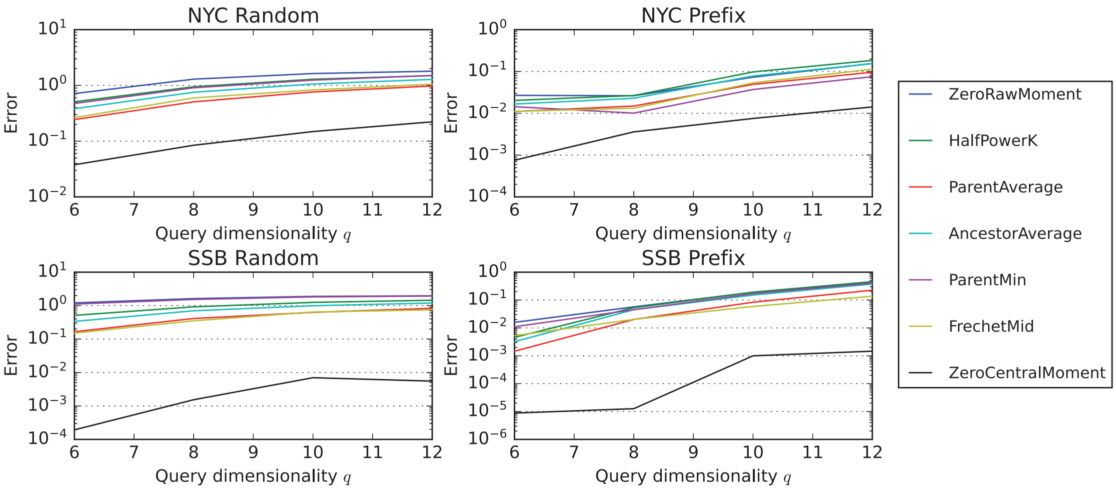


Figure 6.21: Average error for different moment extrapolation techniques run in batch mode for various query dimensionality after applying heuristic bounds to improve the error

6.3.4 Improved Moment Extrapolation Algorithm

In the previous section, an algorithm was described for extrapolating the moments of a cuboid based on the assumption that the corresponding central moments are zero. But the main drawback of the algorithm is that it extrapolates each unknown moment individually, with the worst case complexity of $\mathcal{O}(2^{|Q|})$ for each unknown moment, and there can be $\mathcal{O}(2^{|Q|})$ unknown moments, bringing the overall complexity to $\mathcal{O}(4^{|Q|})$. A more efficient algorithm that extrapolates all unknown moments together in $\mathcal{O}(|Q|)$ steps is described next. Before going into the details of the algorithm, we will establish the relationship between $\boldsymbol{\mu}_I$, $\boldsymbol{\sigma}_I$ and $\boldsymbol{C}_I$, where $\boldsymbol{\mu}_I$ and $\boldsymbol{\sigma}_I$ are the generalized raw and central moments of the vectorized cuboid $\boldsymbol{C}_I$. The following theorem extends Proposition 9 and is a modified version of [99, Theorem 1].

Theorem 12. *For any $I \subseteq [n]$ with $I = \{i_1, \ldots, i_m\}$, the vectors $\boldsymbol{\mu}_I$, $\boldsymbol{\sigma}_I$ and $\mathbf{C}_I$ are related to each other by the following equations:*

(1) $\boldsymbol{\mu}_I = \left(\begin{smallmatrix} 1 & 1 \\ 0 & 1 \end{smallmatrix}\right)^{\otimes m} \boldsymbol{C}_I$

(2) $\boldsymbol{\sigma}_I = \left(\begin{pmatrix} 1 & 0 \\ -\theta_{i_m} & 1 \end{pmatrix} \otimes \ldots \otimes \begin{pmatrix} 1 & 0 \\ -\theta_{i_1} & 1 \end{pmatrix}\right) \boldsymbol{\mu}_I$

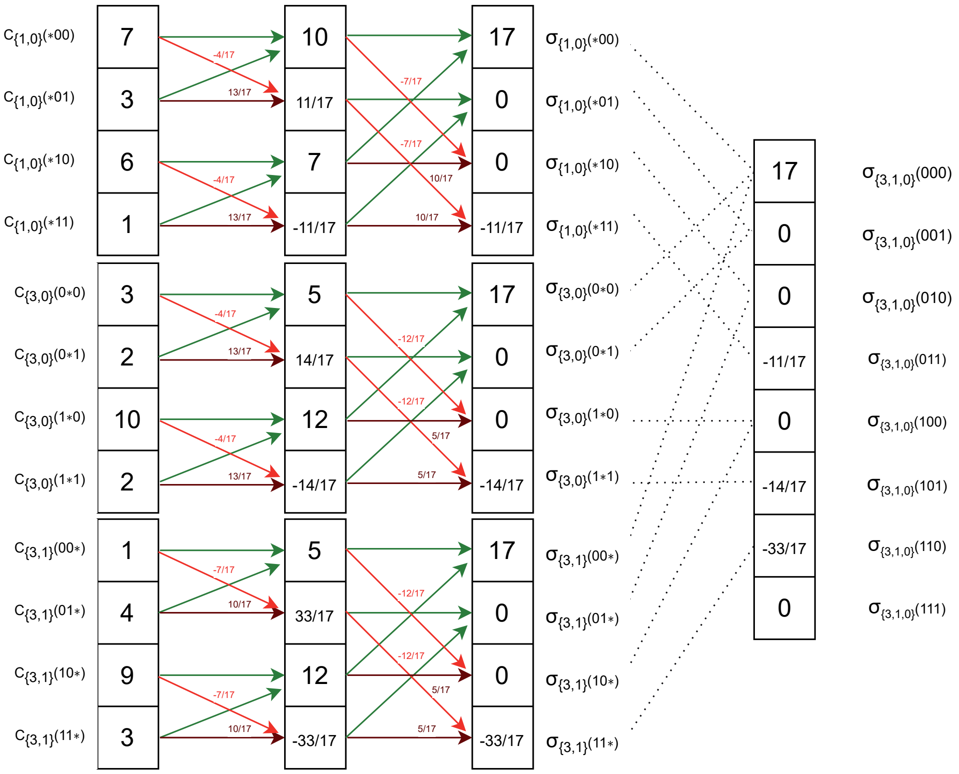
(3) $\boldsymbol{\sigma}_I = \Bigg(\begin{pmatrix} 1 & 1 \\ -\theta_{i_m} & \bar{\theta}_{i_m} \end{pmatrix}\Bigg) \otimes \ldots \otimes \begin{pmatrix} 1 & 1 \\ -\theta_{i_1} & \bar{\theta}_{i_1} \end{pmatrix}\Bigg) \boldsymbol{C}_I$

(4) $\boldsymbol{C}_I = \left(\begin{smallmatrix} 1 & -1 \\ 0 & 1 \end{smallmatrix}\right)^{\otimes m} \boldsymbol{\mu}_I$

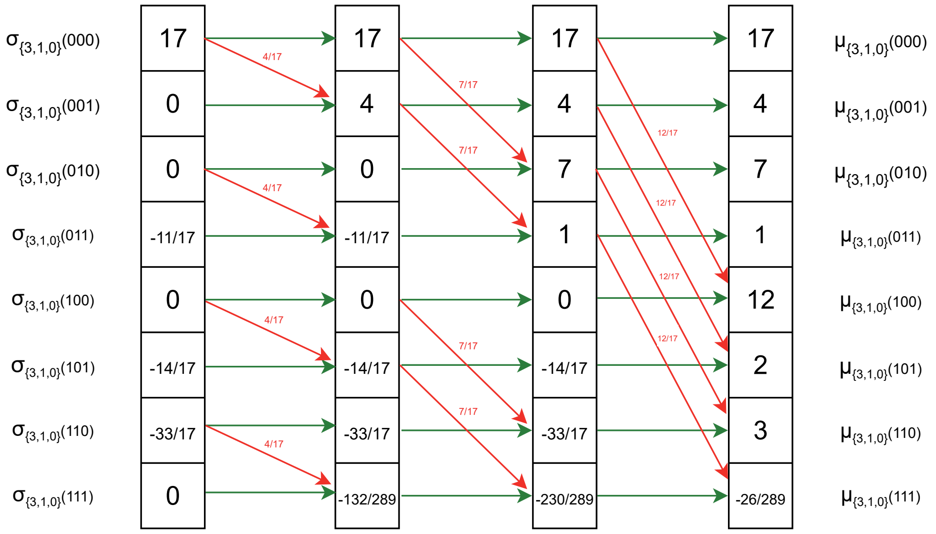
(5) $\boldsymbol{\mu}_I = \left(\begin{pmatrix} 1 & 0 \\ \theta_{i_m} & 1 \end{pmatrix} \otimes \ldots \otimes \begin{pmatrix} 1 & 0 \\ \theta_{i_1} & 1 \end{pmatrix}\right) \boldsymbol{\sigma}_I$

(6) $\boldsymbol{C}_I = \left(\begin{pmatrix} \bar{\theta}_{i_m} & -1 \\ \theta_{i_m} & 1 \end{pmatrix} \otimes \ldots \otimes \begin{pmatrix} \bar{\theta}_{i_1} & -1 \\ \theta_{i_1} & 1 \end{pmatrix}\right) \boldsymbol{\sigma}_I$

This theorem establishes that the vectorized cuboid $\boldsymbol{C}_I$, the vector for raw moments $\boldsymbol{\mu}_I$, and the vector for central moments $\boldsymbol{\sigma}_I$ all represent the same information in different formats and can be interchanged using the fast transformation algorithm described in Algorithm 7. Algorithm 13 describes our new implementation of moment extrapolation to reconstruct the approximate joint distribution $\widetilde{\boldsymbol{C}}_Q$ for query Q given projection cuboids for some sets $I \in \mathcal{I}(Q)$.


C{1,0}(*00) 7 10 17 σ{1,0}(*00)
C{1,0}(*01) 3 11/17 0 σ{1,0}(*01)
C{1,0}(*10) 6 7 0 σ{1,0}(*10)
C{1,0}(*11) 1 -11/17 -11/17 σ{1,0}(*11)
C{3,0}(0*0) 3 5 17 σ{3,0}(0*0)
C{3,0}(0*1) 2 14/17 0 σ{3,0}(0*1)
C{3,0}(1*0) 10 12 0 σ{3,0}(1*0)
C{3,0}(1*1) 2 -14/17 -14/17 σ{3,0}(1*1)
C{3,1}(00*) 1 5 17 σ{3,1}(00*)
C{3,1}(01*) 4 33/17 0 σ{3,1}(01*)
C{3,1}(10*) 9 12 0 σ{3,1}(10*)
C{3,1}(11*) 3 -33/17 -33/17 σ{3,1}(11*)
-4/17
13/17
-7/17
10/17
-12/17
5/17
17 σ{3,1,0}(000)
0 σ{3,1,0}(001)
0 σ{3,1,0}(010)
-11/17 σ{3,1,0}(011)
0 σ{3,1,0}(100)
-14/17 σ{3,1,0}(101)
-33/17 σ{3,1,0}(110)
0 σ{3,1,0}(111)

(a) Extracting central moments from projections


σ{3,1,0}(000) 17 17 17 17 μ{3,1,0}(000)
σ{3,1,0}(001) 0 4 4 4 μ{3,1,0}(001)
σ{3,1,0}(010) 0 0 7 7 μ{3,1,0}(010)
σ{3,1,0}(011) -11/17 -11/17 1 1 μ{3,1,0}(011)
σ{3,1,0}(100) 0 0 0 12 μ{3,1,0}(100)
σ{3,1,0}(101) -14/17 -14/17 -14/17 2 μ{3,1,0}(101)
σ{3,1,0}(110) -33/17 -33/17 -33/17 3 μ{3,1,0}(110)
σ{3,1,0}(111) 0 -132/289 -230/289 -26/289 μ{3,1,0}(111)
4/17
7/17
12/17

(b) Transforming central moments to raw moments

Figure 6.22: Approximating query result from projections using the improved moment solver. The improved moment solver extracts central moments directly from projections and approximates query results assuming unknown central moments to be zero. Instead of transforming extrapolated central moments directly to the query cuboid, it first transforms the central moments to raw moments and then transforms raw moments to the query cuboid while applying the heuristic bounds to improve the error.

Algorithm 13: Improved algorithm to approximate query cuboid from projections using moments

```
input  : Query $Q$ for which joint distribution is to be approximated, cuboids $\boldsymbol{C}_I$ for several
         $I \in \mathscr{I}(Q)$, 1-D marginals $\boldsymbol{\theta} = (\theta_h)_{h \in I}$
output : Approximated query cuboid $\widetilde{\boldsymbol{C}}_Q$
Initialize array $\mathsf{A}$ of size $2^{|Q|}$ with zeroes to represent $\boldsymbol{\sigma}_Q$.
foreach $I \in \mathscr{I}(Q)$ do
    $\boldsymbol{\sigma}_I \leftarrow$ ExtractCentralMoments($I, \boldsymbol{p}_I, \boldsymbol{\theta}$)
    foreach $\boldsymbol{y} \in \{0,1\}^I$ do
        $\mathsf{A}[\boldsymbol{y}_{\uparrow Q}] \leftarrow \boldsymbol{\sigma}_I(\boldsymbol{y})$ // using Proposition 1
// Transform $\boldsymbol{\sigma}$ to $\boldsymbol{\mu}$ first
foreach $h \in Q$ do
    $\boldsymbol{M} \leftarrow \left(\begin{smallmatrix} 1 & 0 \\ \theta_h & 1 \end{smallmatrix}\right)$
    $\mathsf{A} \leftarrow$ Transform($\mathsf{A}, \boldsymbol{M}, h$)
// Transform $\boldsymbol{\mu}$ to $\boldsymbol{p}$ while applying bounds
foreach $h \in Q$ do
    $\boldsymbol{M} \leftarrow \left(\begin{smallmatrix} 1 & -1 \\ 0 & 1 \end{smallmatrix}\right)$
    $\mathsf{A} \leftarrow$ TransformWithBounds($\mathsf{A}, \boldsymbol{M}, h$)
return $\mathsf{A}$ as $\widetilde{\boldsymbol{C}}_Q$
```

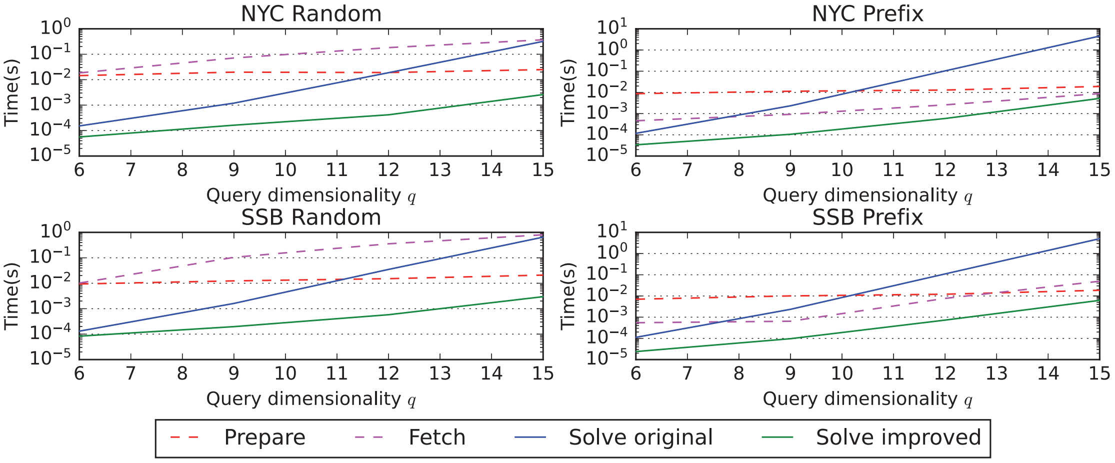
We first extract the central moments $\sigma_I(\boldsymbol{y})$ from each projection C_I and copy their values into the appropriate slot for $\sigma_Q(\boldsymbol{y}_{\uparrow Q})$, using Proposition 1. We set all remaining unknown central moments in $\boldsymbol{\sigma}_Q$ to zero and perform the reverse transformation to obtain the (approximate) cuboid $\boldsymbol{C}_Q$ from the (extended) central moments $\boldsymbol{\sigma}_Q$. In order to apply the heuristics we describe in the previous section, we don't transform central moments $\boldsymbol{\sigma}_Q$ directly into cuboid $\boldsymbol{C}_Q$. We first transform the central moments $\boldsymbol{\sigma}_Q$ into raw moments $\boldsymbol{\mu}_Q$. Then, the raw moments $\boldsymbol{\mu}_Q$ are transformed into the cuboid $\boldsymbol{C}_Q$ while applying the local bounds, similar to how it was done in Algorithm 12.

Example 18. *Consider the query $Q = \{3,1,0\}$ again. Figure 6.22a shows how the improved moment solver extracts the central moments $\boldsymbol{\sigma}_I$ instead of raw moments $\boldsymbol{\mu}_I$ from cuboid $\boldsymbol{C}_I$ and fills the central moments of the query $\boldsymbol{\sigma}_Q$. The sole unknown moment $\sigma_Q(111)$ is set to 0. Then, Figure 6.22b shows the transformation converting $\boldsymbol{\sigma}_Q$ to $\boldsymbol{\mu}_Q$. After this step, $\boldsymbol{\mu}_Q$ is transformed to $\boldsymbol{C}_Q$ using the transformation with bounds. When the lower bound is applied, $\boldsymbol{\mu}_Q(111)$ is set as 0, and the computation proceeds as in Figure 6.18.*

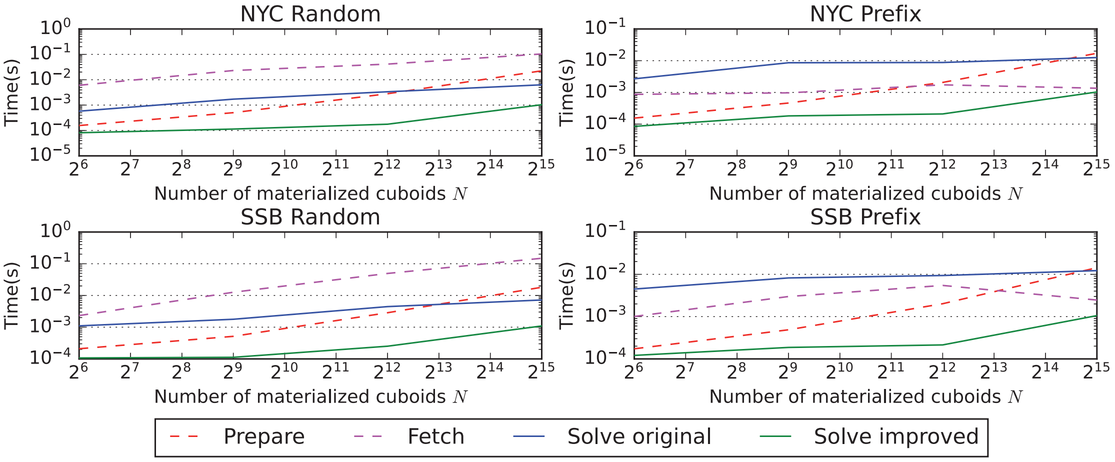
We now run some experiments evaluating the performance of the improved moment algorithm in Algorithm 13 vis-à-vis the original moment algorithm described in Algorithm 10.

Experiment 6.10 *Varying Query Dimensionality on Improved Moment Solver in Batch Mode*

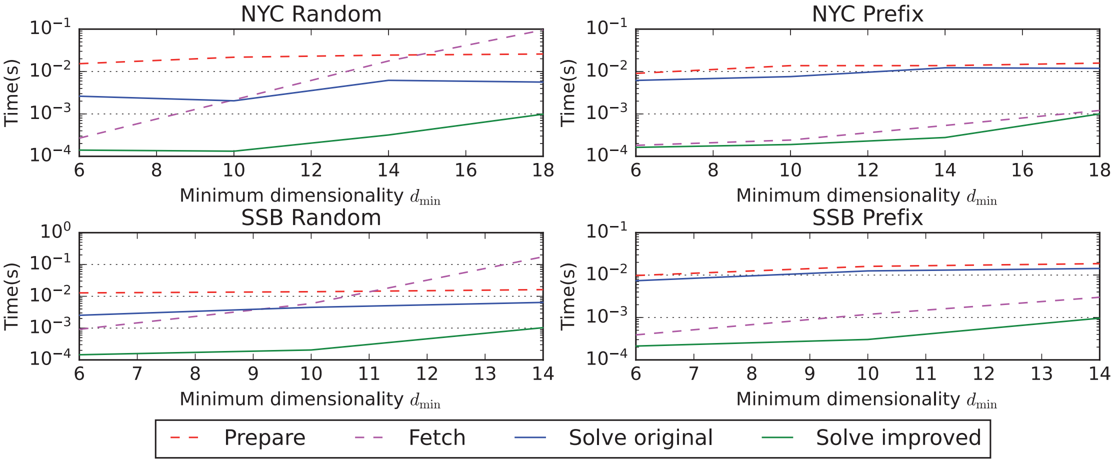
To study the impact of query dimensionality, we run 100 queries each of various dimensionality in batch mode on data cubes built on NYC and SSB datasets using Random and Prefix strategies. We fix parameters $N = 2^{15}$ for all four data cubes and $d_{\min} = 14$ for SSB and



(a) Varying query dimensionality

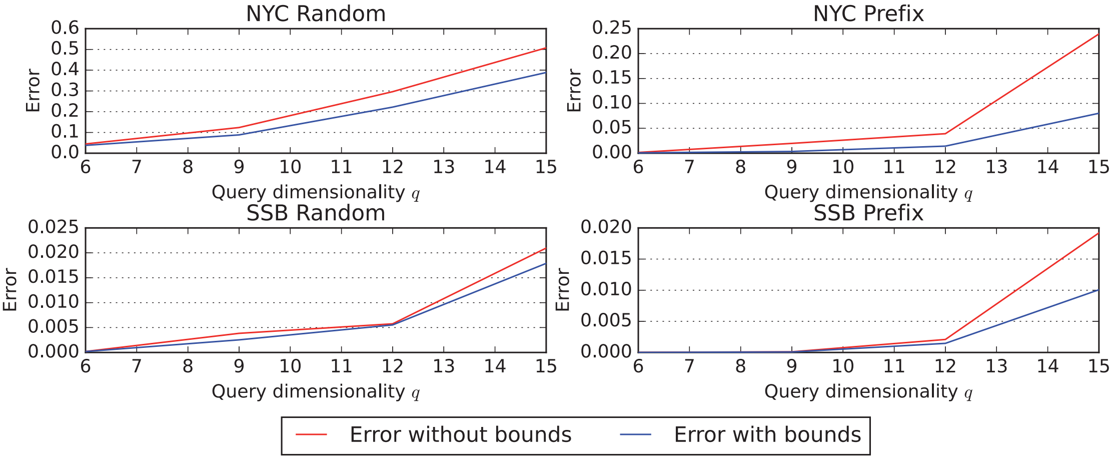


(b) Varying number of materialized cuboids

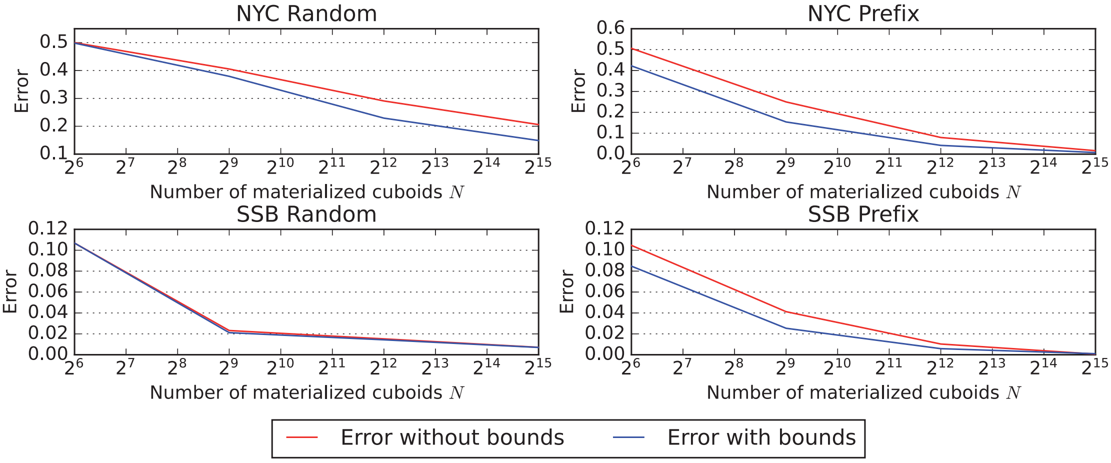


(c) Varying minimum dimensionality

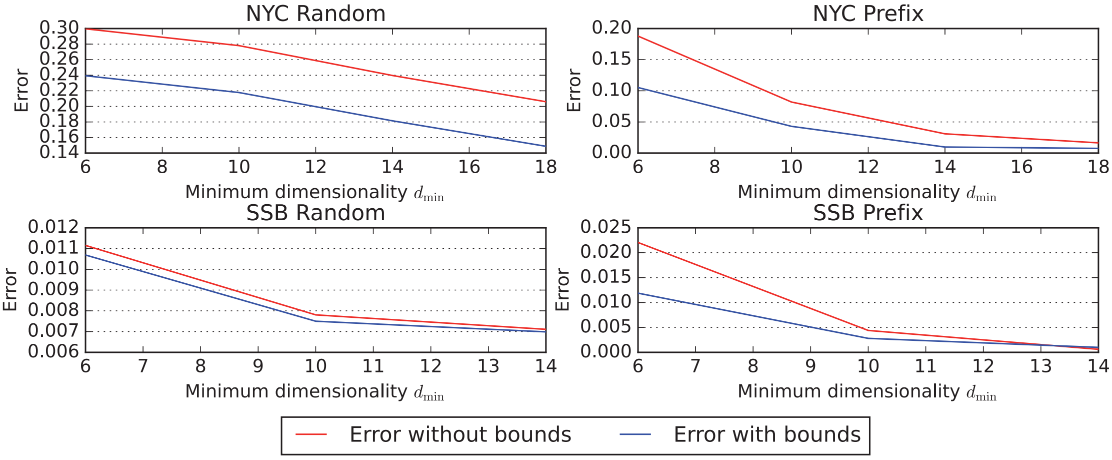
Figure 6.23: Average time spent by the original and the improved moment solvers in each phase of query execution in batch mode while varying the query dimensionality, the number of materialized cuboids and the minimum dimensionality of materialized cuboids



(a) Varying query dimensionality



(b) Varying number of materialized cuboids



(c) Varying minimum dimensionality

Figure 6.24: Average error for the improved moment solver run in batch mode with and without applying the heuristic bounds for improving the error while varying the query dimensionality, the number of materialized cuboids, and the minimum dimensionality of materialized cuboids

$d_{\min} = 18$ for NYC. Figure 6.23a shows the average time spent on solving for both techniques along with the prepare and fetch time. It is clear from the figure that the improved algorithm solves asymptotically faster compared to the original algorithm. The distribution for the dimensionality of cuboids before and after projection was shown in Figure 6.10, and their impact on prepare and fetch time was discussed in Experiment 6.4.

We also study the impact of query dimensionality on the error. Note that both the original and improved moment solver yield identical results as they both approximate query results from their projections with the assumption that all unknown central moments are zero. We show the average error when extrapolating moments by setting unknown central moments to zero for various query dimensionality in Figure 6.24a. We plot errors for both cases when the local bounds are applied during the reverse transformation to cuboids and when they are not. We observe that the error increases exponentially with query dimensionality. We also observe that the error is very low for prefix queries, especially when the bounds are applied. △

Experiment 6.11 *Varying Number of Materialized Cuboids on Improved Moment Solver in Batch Mode*

We fix the query dimensionality to 10 and run 100 queries in batch mode on data cubes with varying the number of materialized cuboids. We build data cubes NYC Random and NYC Prefix with minimum dimensionality set to 18, and SSB Random and SSB Prefix data cubes with minimum dimensionality set to 14.

We show the histogram for the cuboid dimensionality before and after projection in Figure 6.12 and discuss its impact on prepare and fetch times in Experiment 6.5. Figure 6.23b shows the average prepare, fetch and solve times using both the original and the improved algorithms for various numbers of materialized cuboids. The solve time increases with the number of materialized cuboids for both versions of the moment solver due to an increase in the number of cuboids to process in the case of the Random data cubes and due to an increase in the dimensionality of the projected cuboid in the case of Prefix data cubes.

Figure 6.24b shows the impact of the number of materialized cuboids on error before and after applying the bounds for the improved moment solver. The additional information available to the solver due to the increased number of materialized cuboids leads to a decrease in the errors for both variants of the solving algorithm in all four data cubes. △

Experiment 6.12 *Varying Minimum Dimensionality of Materialized Cuboids on Improved Moment Solver in Batch Mode*

In this experiment, we fix the query dimensionality to 10 and run 100 queries on data cubes built on NYC and SSB datasets with Random and Prefix strategies with a total number of cuboids 2^{15}. The histogram for the dimensionality of cuboids for this experiment is shown in Figure 6.14, and its impact on prepare and fetch times is discussed in Experiment 6.6. The solve time for the Random data cubes increases slightly as more cuboids are processed, as shown in Figure 6.23b. On the other hand, the solve time increases for Prefix cubes at a lower rate as the increase in dimensionality is counterbalanced by a decrease in the number of cuboids to be processed.

Figure 6.24c shows the impact of the dimensionality of cuboids on the error for the improved moment solver. The increased dimensionality of the projected cuboids yields many more moments and reduces the error of the approximated result. △

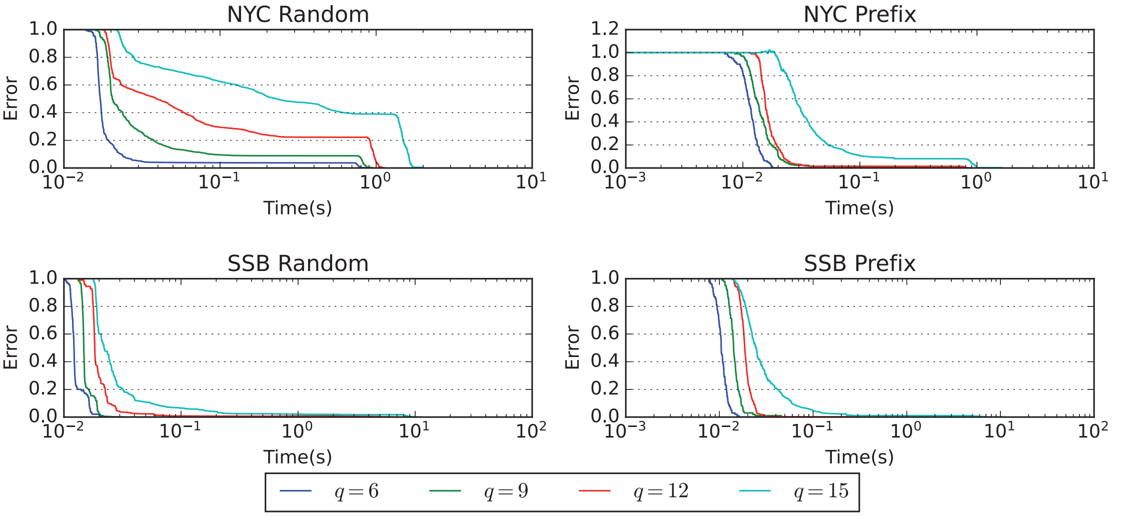


Figure 6.25: Improvement of error over time for the improved moment solver in online mode for various query dimensionality

Experiment 6.13 *Studying Improvement of Error over Time for Improved Moment Solver in Online mode while Varying Query Dimensionality*

Finally, we run queries on the moment solver in online mode and study how the error drops with time. We run 100 queries each of various dimensionality on the NYC Random, NYC Prefix, SSB Random, and SSB Prefix data cubes with 2^{15} cuboids materialized. The minimum dimensionality of the materialized cuboids is set to 18 for NYC cubes and 14 for SSB cubes. The approximate solution calculated by the moment solver after processing each fetched cuboid is stored along with the cumulative time elapsed since the submission of the query. The average error for the solutions returned at various times for various query dimensionality is shown in Figure 6.25.

We observe that the error drops very quickly when the moment solver approximates the query results from small low-dimensional projections, and then the error remains steady. At this stage, all projections of the query that can be obtained from materialized cuboids other than the base cuboid have been processed, and the moment solver returns its most accurate approximation, equivalent to the one returned in batch mode. Then, if the query has not yet been answered exactly by some other cuboid, the base cuboid is projected to answer the query, and the error drops to zero.

In the case of the NYC Random data cube, the average error after 1 second of execution time is nearly 0 for query dimensionality 9 and below, and it is around 13% for query dimensionality 12 and 38% for query dimensionality 15. In all other cuboids, the average error is less than or close to 1% for query dimensionality up to 15. △

In summary, we explored techniques to approximate query results that extract several of its moments from its projections and extrapolate unknown moments from known ones. The most effective approach resulted in corresponding central moments being set to zero, leading to low error rates. The moment solver demonstrated its capacity to deliver precise approximations swiftly, given sufficient information from the projections.

However, there were some limitations to the moment solver. Setting the unknown central moments to zero is not always feasible and can result in negative values for approximated cuboid measures, contrary to our assumption of non-negative measure values. The application of heuristic bounds locally, however, has been discussed as a potential countermeasure to these drawbacks, showing a significant decrease in error without any detrimental impact on runtime.

Furthermore, we studied an enhanced version of the moment solver that maintains the zero assumption for unknown central moments while performing the extrapolation more rapidly. Its performance was rigorously evaluated under varying parameters, including query dimensionality, the number of materialized cuboids, and minimum dimensionality of materialized cuboids in a batch mode setting. Moreover, its online mode implementation exhibited remarkable results: for three out of four data cubes, the average error plummeted to less than 1% in under a second, highlighting its efficacy.

6.4 Iterative Proportional Fitting

Iterative proportional fitting (IPF) is an algorithm with a long history. Starting from [28], there has been a lot of work on the properties of the algorithm [25], [52], on optimizations [58], [75], [98], generalizations [26] and on using it in a wide range of applications (see [73], for example).

The algorithm was initially described in [28] for fitting contingency tables. *Contingency tables* [17] contain cross-tabulated data with respect to different variables. The cells of such tables contain count data. Thus, contingency tables can essentially be seen as a particular kind of data cube. The original application was to estimate the joint count distribution over all variables by taking into consideration smaller known tables that are available from sampling and surveying. This is the data synthesis scenario mentioned in our introduction. Later on, IPF was also picked up specifically for the reconstruction of probability distributions [65].

6.4.1 The Base Algorithm

We use a modified version of the IPF algorithm to reconstruct a cuboid C_Q from its known projections C_I for $I \in \mathscr{I}(Q)$. Algorithm 14 shows the base version of IPF. Starting from a cuboid that uniformly distributes the total aggregate among all cells, the algorithm *iterates* through the known projections of the target query cuboid. Whenever a projection of the query cuboid is fetched, the current values are *fitted* to satisfy the corresponding constraints by a *proportional* adjustment simultaneously.

Algorithm 14: Iterative Proportional Fitting (IPF)

```
input  : Cuboids $C_I$ for $I \in \mathscr{I}(Q)$
output : Approximated query cuboid $\widetilde{C}_Q$
Initialize $\widetilde{C}_Q$ with $\frac{total}{2^{|Q|}}$ for all cells $C_Q(\boldsymbol{q})$ with $\boldsymbol{q} \in \{0,1\}^Q$
repeat
    foreach $I \in \mathscr{I}(Q)$ do
        $\widetilde{C}_I \leftarrow$ projection of $\widetilde{C}_Q$ down to $I$
        foreach $\boldsymbol{x} \in \{0,1\}^Q$ do
            $\boldsymbol{y} \leftarrow \boldsymbol{x}_{\downarrow I}$
            $\widetilde{C}_Q(\boldsymbol{x}) \leftarrow \widetilde{C}_Q(\boldsymbol{x}) \cdot \frac{C_I(\boldsymbol{y})}{\widetilde{C}_I(\boldsymbol{y})}$
until convergence criterion reached
return $\widetilde{C}_Q$
```

The innermost loop in the algorithm contains the *IPF update* for fitting the cuboid C_I. The convergence condition in Algorithm 14 requires (for example) that the sum of all perturbations made by the updates is at most ε.

Example 19. *Consider the example query $Q = \{3,1,0\}$ on the sales data cube. When we run IPF, initially, we set all values of approximated cuboid $\widetilde{C}_Q$ to $^{17}/_{8}$. Suppose the first cuboid to fetched is C_I with $I = \{1,0\}$. By projecting $\widetilde{C}_Q$ to I, we have $\widetilde{C}_I$ with all values equal to $^{17}/_{4}$ Then, we update values in $\widetilde{C}_Q$ as shown below.*

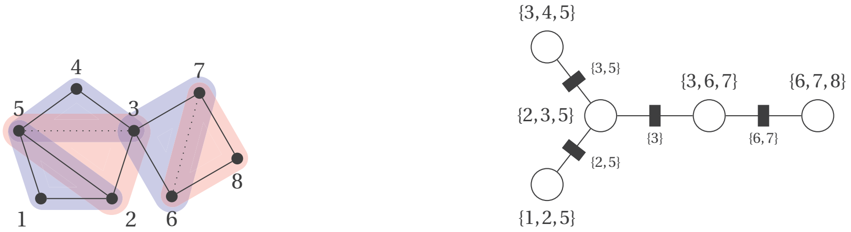
$$\widetilde{C}_Q(000) \leftarrow {}^{17}/_{8} \cdot \frac{7}{^{17}/_{4}} = {}^{7}/_{2} \qquad \widetilde{C}_Q(100) \leftarrow {}^{17}/_{8} \cdot \frac{7}{^{17}/_{4}} = {}^{7}/_{2}$$

$$\widetilde{C}_Q(001) \leftarrow {}^{17}/_{8} \cdot \frac{3}{^{17}/_{4}} = {}^{3}/_{2} \qquad \widetilde{C}_Q(101) \leftarrow {}^{17}/_{8} \cdot \frac{3}{^{17}/_{4}} = {}^{3}/_{2}$$

$$\widetilde{C}_Q(010) \leftarrow {}^{17}/_{8} \cdot \frac{6}{^{17}/_{4}} = {}^{6}/_{2} \qquad \widetilde{C}_Q(110) \leftarrow {}^{17}/_{8} \cdot \frac{6}{^{17}/_{4}} = {}^{6}/_{2}$$

$$\widetilde{C}_Q(011) \leftarrow {}^{17}/_{8} \cdot \frac{1}{^{17}/_{4}} = {}^{1}/_{2} \qquad \widetilde{C}_Q(111) \leftarrow {}^{17}/_{8} \cdot \frac{1}{^{17}/_{4}} = {}^{1}/_{2}$$

Remark 1. *The only situation in which the IPF update could attempt to divide by zero is when the value $\widetilde{C}_I(\boldsymbol{x}_{\downarrow I})$ is zero, which is possible only when $\widetilde{C}_Q(\boldsymbol{x})$ is already zero to begin with. In this case $\widetilde{C}_Q(\boldsymbol{x}) = 0$ is left unchanged [57].*



(a) Triangulation and cliques (b) A junction tree

Figure 6.26: Junction tree for Example 20

6.4.2 Convergence and Maximum Entropy

If the IPF update is executed on a cuboid $\widetilde{C}_Q$ that already agrees with C_I when projected to dimensions I, the update has no effect. Otherwise, the update causes the updated values in the cuboid to satisfy the projection constraint for I. In general, fitting a cuboid C_I may cause previously satisfied projection constraints to be violated again.

Call a cuboid $\widetilde{C}_Q$ a *fixed point* of IPF, if for all $I \in \mathscr{I}(Q)$, applying the IPF update does not change $\widetilde{C}_Q$. Note that $\widetilde{C}_Q$ is a fixed point of IPF if and only if it satisfies all projection constraints. Since the IPF update can violate other constraints again, it can happen that the algorithm runs forever without reaching a fixed point. Yet, it is well-known that, as long there exists a solution, IPF not only always converges, but converges pointwise to the cuboid that *maximizes entropy* among all cuboids that satisfy all the constraints [25, Theorem 3.2].

When multiple solutions to the query cuboid are possible, picking the one having maximum entropy is guided by the principle of maximum entropy [53].

In IPF, for fitting any projection, the entire query cuboid $\widetilde{C}_Q$ needs to be updated, entailing high time and space consumption. For this reason, a number of approaches for making IPF less resource-intensive have been proposed [8], [58], [75], [98].

6.4.3 Graphical Models

We will later discuss, and experiment with an optimization of IPF that uses machinery from the field of *probabilistic graphical models* [63], [101]. Such an algorithm has first been described in [58]. To the best of our knowledge, the state-of-the-art version of this algorithm is the one presented in [98]. In order to describe the algorithm, let us introduce some background, following [101].

A *Markov random field (MRF)* for $\boldsymbol{X} = (X_n, \dots, X_1)$ is an undirected graph G with vertices $[n]$, where edges indicate interactions between the corresponding random variables. More precisely, an MRF G for $\boldsymbol{X}$ models that the joint distribution of $\boldsymbol{X}$ can be written as a normalized product of real-valued functions, called *factors*, defined on the (maximal) cliques of G.

Example 20. *Suppose* $Q = [8]$ *and we know the following projection cuboids:*

$$\mathscr{I}(Q) = \big\{\{1,2,5\},\{2,3\},\{3,4\},\{3,6\},\{3,7\},\{4,5\},\{6,8\},\{7,8\}\big\}.$$

We construct the graph on vertices Q *where every* $I \in \mathscr{I}(Q)$ *has been turned into a clique (Figure 6.26a). Adding the dotted edges makes the graph triangulated. The shaded areas indicate the resulting cliques. These cliques "become" the vertices of the junction tree Figure 6.26b. Tentatively, two vertices* v, w *get connected by an edge* $e = \{v, w\}$ *if* $\mathrm{sep}(e) = \mathrm{cliq}(v) \cap \mathrm{cliq}(w) \neq \emptyset$*. Every such edge is weighted by* $|\mathrm{sep}(e)|$*. In the resulting edge-weighted graph, we find a spanning tree of maximum weight. Figure 6.26b is the traditional depiction of a possible resulting junction tree, with the subsets of* Q *corresponding to cliques and separators highlighted.*

Junction trees provide a representation of graphical models that is used, for example, for performing inference on MRFs. In order to build a junction tree for graph G with vertex set $[n]$ we first *triangulate* the graph, by adding edges if necessary (see Figure 6.26a). For a triangulated graph G, a junction tree is an undirected tree $T = (V, E)$ whose vertices correspond to the maximal cliques of G. That is, every $v \in V$ is associated with a maximal *clique* $\mathrm{cliq}(v) \subseteq [n]$ in G. Every edge $e = \{v, w\}$ can be associated with the intersection $\mathrm{sep}(e) = \mathrm{cliq}(v) \cap \mathrm{cliq}(w)$ called *separators*. Valid junction trees additionally need to satisfy the *running intersection property*, that is, for all $v, w \in V$, whenever u is a vertex on the path from v to w, then $\mathrm{cliq}(v) \cap \mathrm{cliq}(w) \subseteq \mathrm{cliq}(u)$. In terms of factors, every original factor needs to be associated with one of the cliques of the junction tree. Amongst others, the construction of a junction tree is illustrated in the next section.

6.4.4 Junction Tree IPF

The main idea we take away from [58], [98] is to represent the cuboid C_Q of some query Q by a junction tree, where the known projections $(C_I)_{I \in \mathscr{I}(Q)}$ take the role of the factors. Treating (C_I) this way comes down to considering the graph with a vertex set Q in which every $I \in \mathscr{I}(Q)$ forms a clique. We show the preparation of the junction tree for this graph in a bigger example.

Assume, that a junction tree of $\mathscr{I}(Q)$ has already been constructed, and that we have associated every $I \in \mathscr{I}(Q)$ with a vertex v_I such that $I \subseteq C(v_I)$. In a nutshell, the algorithm of [98] now consists of a repeated sequence of IPF updates within a clique, followed by a *propagation* to a neighboring clique.

In order to describe the algorithm, we need some more notation. We work with cuboids projected to the sets belonging to the cliques and separators: For every vertex v and every edge e, we let $\widetilde{C}_v$ and $\widetilde{C}_e$ be the corresponding distributions with dimensions $\mathrm{cliq}(v)$ and $\mathrm{sep}(e)$. Accordingly, we use $\boldsymbol{x}_v$ and $\boldsymbol{x}_e$ to denote the projections of a vector $\boldsymbol{x}$ to these sets. Unlike the case when cuboids are exact, projecting approximations of different cuboids down to the same set of dimensions shared by all of them need not yield the same result. For this reason, we explicitly denote which (approximated) cuboid is projected using "↓" notation: For example,

$\widetilde{C}_{v\downarrow e}$ is the cuboid obtained by projecting the approximated cuboid $\widetilde{C}_v$ associated with clique $\mathrm{cliq}(v)$ down to dimensions in the separator $\mathrm{sep}(e)$.

With this setup, Junction Tree IPF is given by Algorithm 15. The starting vertex may be chosen arbitrarily. The first for-loop contains the IPF updates for a clique. Subsequently, the update is propagated to the next vertex w over the connecting edge. The choice of the next vertex to go to is only required to be *fair*, meaning that every vertex is treated periodically. In our implementation, we follow the proposal of [98] and use a depth-first search traversal.

Algorithm 15: Junction Tree IPF

```
input  : Junction tree $(V, E)$ for $\mathscr{I}(Q)$, cuboid $C_I$ for all $I \in \mathscr{I}(Q)$
output : Approximated query cuboid $\widetilde{C}_Q$
Initialize all $\widetilde{C}_v$, $\widetilde{C}_e$ to uniform distributions.
$v \leftarrow$ some initial vertex from $V$
repeat
    foreach $I \in \mathscr{I}(Q)$ with $v_I = v$ do                         // IPF updates
        foreach $\boldsymbol{x} \in \{0,1\}^{\mathrm{cliq}(v)}$ do
            $\widetilde{C}_v(\boldsymbol{x}) \leftarrow \widetilde{C}_v(\boldsymbol{x}) \cdot \frac{C_I(\boldsymbol{x}_I)}{\widetilde{C}_{v\downarrow I}(\boldsymbol{x}_I)}$
    $e \leftarrow$ an edge $\{v, w\}$ to a neighboring vertex $w$
    foreach $\boldsymbol{x} \in \{0,1\}^{\mathrm{cliq}(w)}$ do                       // Clique prop.
        $\widetilde{C}_w(\boldsymbol{x}) \leftarrow \widetilde{C}_w(\boldsymbol{x}) \cdot \frac{\widetilde{C}_{v\downarrow e}(\boldsymbol{x}_e)}{\widetilde{C}_e(\boldsymbol{x}_e)}$
    foreach $\boldsymbol{x} \in \{0,1\}^{\mathrm{sep}(e)}$ do                     // Separator prop.
        $\widetilde{C}_e(\boldsymbol{x}) \leftarrow \widetilde{C}_{v\downarrow e}(\boldsymbol{x}_e)$
    $v \leftarrow w$
until convergence criterion reached
foreach $\boldsymbol{x} \in \{0,1\}^Q$ do
    $\widetilde{C}_Q(\boldsymbol{x}) \leftarrow \frac{\prod_{v\in V} \widetilde{C}_v(\boldsymbol{x}_v)}{\prod_{e\in E} \widetilde{C}_e(\boldsymbol{x}_e)}$
return $\widetilde{C}_Q$
```

Example 21. *In Figure 6.26b, the association of sets $I \in \mathscr{I}(Q)$ is uniquely determined. For example, $I = \{3,6\}$ has to be associated with the clique $\{3,6,7\}$. The algorithm could start with the top left vertex and traverse the graph in the way of a depth-first search, where for each vertex, the IPF update is made for all relevant $I \in \mathscr{I}(Q)$, followed by a propagation to an outgoing separator and a next clique. The propagation (interpreted as* message passing*) is illustrated in [98] for the junction tree of Figure 6.26b.*

The correctness of Algorithm 15 in particular relies on the fact that for a junction tree, the maximum entropy distribution decomposes precisely as in the last step of the algorithm [57, Theorem 1].

6.4.5 Moment-based IPF

The Junction Tree IPF presented in the previous section speeds up the iterative update process by partitioning the query dimensions into smaller sets, running iterative updates separately on each partition, and then combining the results. If we cannot split the query dimensions into several small partitions satisfying the requirements for the junction tree, then there would be no speed-up for the iterative update. In this section, we explore a more efficient implementation for the vanilla IPF algorithm taking advantage of our knowledge of the moment transformations from Section 6.3.2.

The most expensive part of the algorithms for Vanilla and Junction Tree IPF described earlier in this section is the projection of the current approximation of the query cuboid $\widetilde{\boldsymbol{C}}_Q$ to scale it to the cuboid currently being processed. There are several optimizations we can apply to make the projection more efficient. First, instead of iterating over all $\boldsymbol{x} \in \{0,1\}^Q$, projecting each $\boldsymbol{x}$ to $\boldsymbol{x}_{\downarrow I}$, and computing the value of projection $\widetilde{\boldsymbol{C}}_I(\boldsymbol{x}_{\downarrow I})$, we could split the iteration into two loops. We could first iterate over $\boldsymbol{y} \in \{0,1\}^I$, compute the scale factor for $\boldsymbol{C}_I(\boldsymbol{y})$ divided by $\widetilde{\boldsymbol{C}}_I(\boldsymbol{y})$, then loop over all $\boldsymbol{z} \in \{0,1\}^{Q\setminus I}$ to update the value of $\widetilde{\boldsymbol{C}}_Q(\boldsymbol{x})$ with $\boldsymbol{x}$ constructed as $\boldsymbol{y} \uplus \boldsymbol{z}$. Secondly, the entire projection $\widetilde{\boldsymbol{C}}_I$ can be computed simultaneously, not just for individual entries in the projection. This allows us to reuse some of the computation involved in finding cells in $\widetilde{\boldsymbol{C}}_Q$ for which the measures should be aggregated to compute the measure for which cell in $\widetilde{\boldsymbol{C}}_I$. Moreover, we can formulate the projection of the array by applying the moment transformation partially, as the following theorem states.

Theorem 13. *Given a cuboid C_Q for some $Q \subseteq [n]$, projecting C_Q to dimensions in set $I \subseteq Q$ is equivalent to applying the moment transform partially on dimensions $Q \setminus I$ and then slicing the result for any $\boldsymbol{x} \in \{0,1\}^I$ as $C_I(\boldsymbol{x}) = D_{Q|\boldsymbol{0}}(\boldsymbol{x})$, where $\boldsymbol{0}$ is the zero vector from $\{0,1\}^{Q\setminus I}$ and*

$$\boldsymbol{D}_Q = \left(\boldsymbol{M}^{\otimes|Q\setminus I|} \otimes \boldsymbol{I}_2^{\otimes|I|}\right)\boldsymbol{C}_Q$$

Algorithm 16 describes the algorithm that applies the optimizations described above to the Vanilla IPF algorithm described in Algorithm 14. While processing cuboid C_I for any $I \in \mathscr{I}(Q)$, the algorithm transforms $\widetilde{\boldsymbol{C}}_Q$ to another representation that contains entries of $\widetilde{\boldsymbol{C}}_I$ by applying the moment transform on dimensions $Q \setminus I$, scales that representation to fit C_I and then transforms it back to $\widetilde{\boldsymbol{C}}_Q$ by applying the reverse moment transformation on dimensions $Q \setminus I$. There is still scope for improvement here. When processing two successive sets $I, J \in \mathscr{I}(Q)$, we apply the reverse transformation at the end of the iteration on $Q \setminus I$ only to apply the forward moment transform on $Q \setminus J$ at the beginning of the next iteration. If the sets I and J have considerable overlap in dimensions, so too would $Q \setminus I$ and $Q \setminus J$, and we can speed up the computation by not doing the reverse transformation at the end of processing I and forward transformation at the start of processing J for the dimensions common to both sets I and J. We can process the sets in the order that maximizes the overlap between two successive sets and minimizes their difference.

We formalize this optimal ordering problem on a graph. Given the set of dimension sets $\mathscr{I}(Q)$ relevant to answering queries, we construct a graph G with each set $I \in \mathscr{I}(Q)$ forming a vertex. Every pair of vertices I and J share an edge which is weighted by the size of their symmetric difference $|I \triangle J|$. Then an optimal ordering of sets I from $\mathscr{I}(Q)$ is given by any minimum weight tour yielded as a solution to the traveling-salesman problem [38] on this graph. Computing an optimal tour is computationally hard, but an approximation can be obtained by constructing a minimum spanning tree [64] of the graph [46], [47]. A preorder traversal of this minimum spanning tree yields an order that is guaranteed to have a cost at most twice that of the minimum weight tour. Algorithm 17 describes an algorithm that uses this order for processing sets from $\mathscr{I}(Q)$ and avoids forward and backward transformations on dimensions shared between successive sets that cancel each other.

Algorithm 16: Moment-based IPF

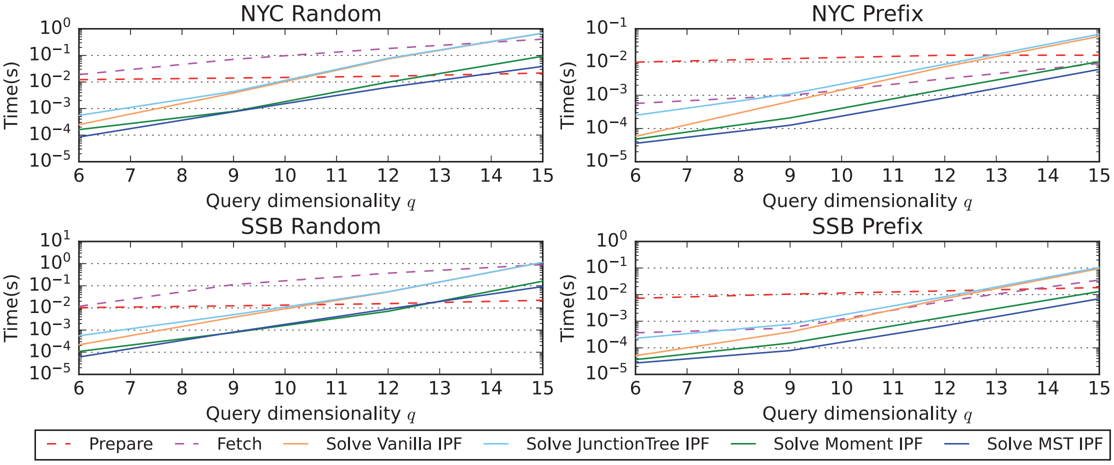
```
input  : Query $Q$, Cuboid $C_I$ for all $I \in \mathscr{I}(Q)$
output : Approximate query cuboid $\widetilde{C}_Q$
Initialize $\mathbf{A}$ of size $2^{|Q|}$ with every value $\frac{total}{2^{|Q|}}$
$M \leftarrow \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$      $W \leftarrow \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}$
foreach $I \in \mathscr{I}(Q)$ do
    foreach $h \in Q \setminus I$ do
        $\mathbf{A} \leftarrow$ Transform($\mathbf{A}$, $M$, $h$)
    foreach $\boldsymbol{i} \in \{0,1\}^I$ do
        $factor \leftarrow \frac{C_I(\boldsymbol{i})}{\mathbf{A}[\boldsymbol{i} \uplus \mathbf{0}]}$
        foreach $\boldsymbol{j} \in \{0,1\}^{Q \setminus I}$ do
            $\mathbf{A}[\boldsymbol{i} \uplus \boldsymbol{j}] \leftarrow factor \cdot \mathbf{A}[\boldsymbol{i} \uplus \boldsymbol{j}]$
    foreach $h \in Q \setminus I$ do
        $\mathbf{A} \leftarrow$ Transform($\mathbf{A}$, $W$, $h$)
return $\mathbf{A}$ as $\widetilde{C}_Q$
```

We run several experiments to compare the performance of the various IPF solvers. We use the error metric defined in Equation (6.5) for moment solvers to measure the quality of the approximations provided by the various IPF solvers as well.

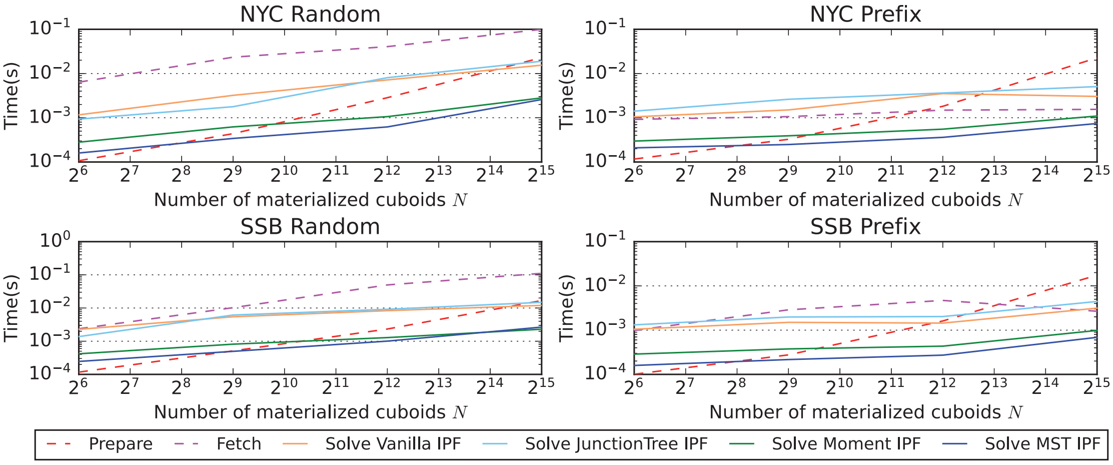
Experiment 6.14 *Varying Query Dimensionality on IPF Solvers in Batch Mode*

We study the impact of varying the query dimensionality on the execution time as well as the error for the approximate query answers returned by various IPF solvers. We run 100 queries each of various dimensionality on NYC Random, NYC Prefix, SSB Random, and SSB Prefix data cubes. The total number of materialized cuboids is set to 2^{15} in all four data cubes and the minimum dimensionality is set to 14 for SSB and 18 for NYC.

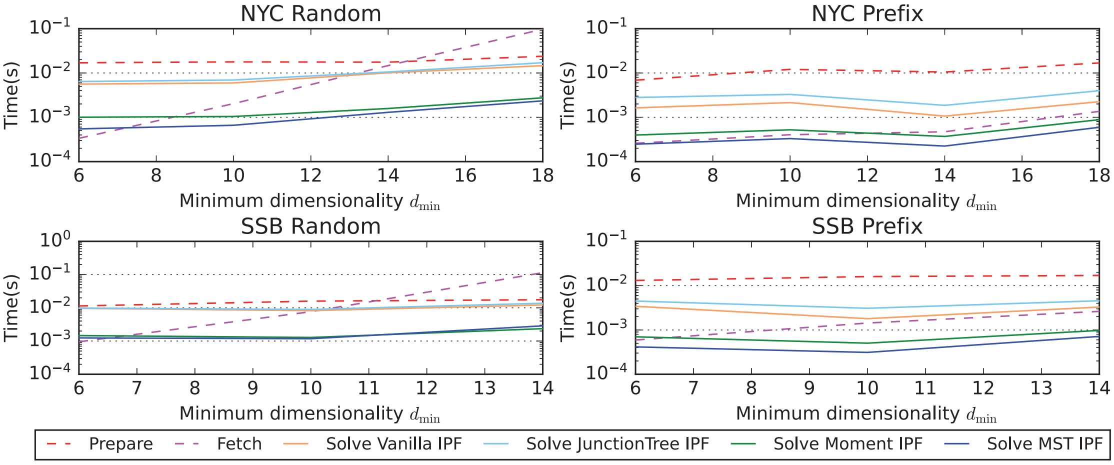
Figure 6.27a shows the average solve time for various IPF solvers in addition to prepare and fetch time for various query dimensionality. We observe that the Junction Tree variant of the IPF algorithm does not significantly reduce solve time, but increases it due to the initial



(a) Varying query dimensionality

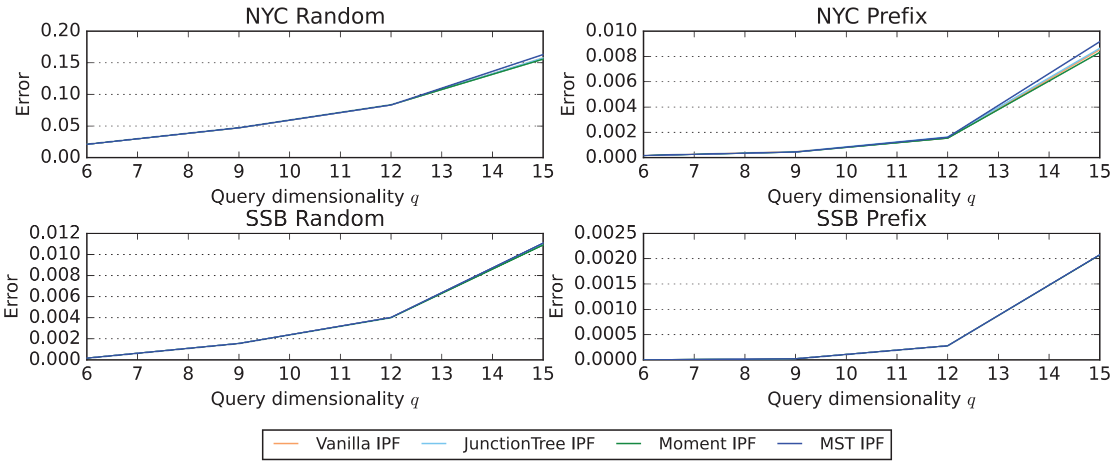


(b) Varying number of materialized cuboids

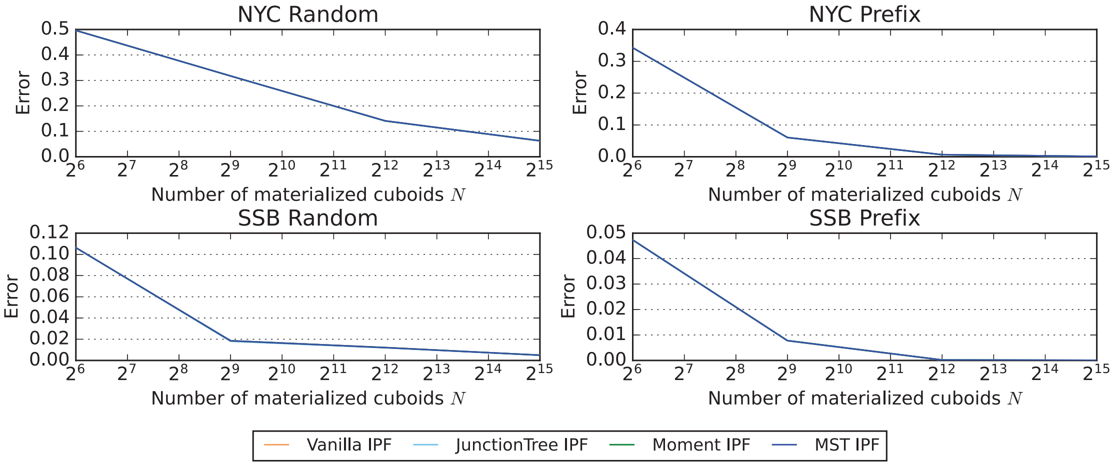


(c) Varying minimum dimensionality

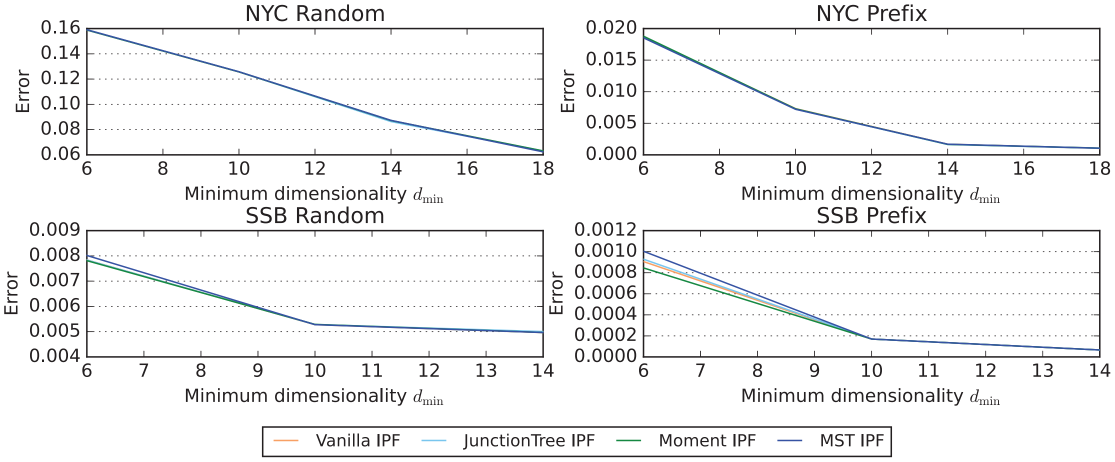
Figure 6.27: Average time spent by each IPF solver in each phase of query execution in batch mode while varying the query dimensionality, the number of materialized cuboids and the minimum dimensionality of materialized cuboids



(a) Varying query dimensionality



(b) Varying number of materialized cuboids



(c) Varying minimum dimensionality

Figure 6.28: Average error for each IPF solver run in batch mode while varying the query dimensionality, the number of materialized cuboids, and the minimum dimensionality of materialized cuboids

Algorithm 17: Minimum Spanning Tree Moment-Based IPF

```
input  : Query $Q$, Cuboid $C_I$ for all $I \in \mathscr{I}(Q)$
output : Approximate query cuboid $\widetilde{C}_Q$
Initialize $\mathbf{A}$ of size $2^{|Q|}$ with every value $\frac{total}{2^{|Q|}}$
$\boldsymbol{M} \leftarrow \left(\begin{smallmatrix} 1 & 1 \\ 0 & 1 \end{smallmatrix}\right)$     $\boldsymbol{W} \leftarrow \left(\begin{smallmatrix} 1 & -1 \\ 0 & 1 \end{smallmatrix}\right)$
Construct undirected graph $G$ with $\mathscr{I}(Q)$ forming the vertex set and every edge $(I, J)$ is
  weighted by the symmetric difference $|I \triangle J|$
$T \leftarrow$ minimum spanning tree of $G$
$S \leftarrow Q$
foreach $I \in$ preorder traversal of $T$ do
    foreach $h \in S \setminus I$ do
        $\mathbf{A} \leftarrow$ Transform($\mathbf{A}$, $\boldsymbol{M}$, $h$)
    foreach $h \in I \setminus S$ do
        $\mathbf{A} \leftarrow$ Transform($\mathbf{A}$, $\boldsymbol{W}$, $h$)
    $S \leftarrow I$
    foreach $\boldsymbol{i} \in \{0,1\}^I$ do
        $factor \leftarrow \frac{C_I(\boldsymbol{i})}{\mathbf{A}[\boldsymbol{i} \uplus \mathbf{0}]}$
        foreach $\boldsymbol{j} \in \{0,1\}^{Q \setminus I}$ do
            $\mathbf{A}[\boldsymbol{i} \uplus \boldsymbol{j}] \leftarrow factor \cdot \mathbf{A}[\boldsymbol{i} \uplus \boldsymbol{j}]$
foreach $h \in Q \setminus S$ do
    $\mathbf{A} \leftarrow$ Transform($\mathbf{A}$, $\boldsymbol{W}$, $h$)
return $\mathbf{A}$ as $\widetilde{C}_Q$
```

preprocessing cost for constructing the junction tree. The impact of the optimizations for efficient iterations can be seen in the reduced solving time for the moment-based IPF solver. The MST IPF solver reduces it even further by optimizing the cuboid processing order. The impact of the ordering can be seen more in the case of prefix queries on data cubes built using the Prefix strategy as there is significant overlap between various cuboids, which can be verified from Figure 6.10b. The solve time increases exponentially with the query dimensionality for all solvers due to the greater number of projections in the case of Random data cubes and larger projection sizes in the case of Prefix data cubes.

Figure 6.28a shows the average error for various IPF solvers changes when the query dimensionality is increased. All the different IPF solvers produce nearly identical results, and therefore the errors are similar to each other. We also observe that the errors are lower for Prefix cubes compared to Random cubes, in agreement with the reduced degrees of freedom and larger projection sizes. We also observe that lower error for SSB cubes compared to NYC cubes, as the latter consists of randomly generated data with higher entropy, and, therefore, closely matches the maximum entropy solution returned by the IPF solvers. △

Experiment 6.15 *Varying Number of Materialized Cuboids on IPF Solvers in Batch Mode*

We fix the query dimensionality to 10 and run 100 queries on NYC and SSB data cubes built using Random and Prefix strategies with varying the number of materialized cuboids. These cuboids are selected with minimum dimensionality set to 14 for the SSB dataset and 18 for the NYC dataset.

Figure 6.27b shows the impact of the number of materialized cuboids on the average solve time for various IPF solvers run in batch mode. The fetch and prepare time increases as described in Experiment 6.5 affected by the change in the distribution of cuboid dimensionality shown in Figure 6.12. We observe that the solve time increases following an increase in the number of fetched cuboids as a consequence of the increased number of materialized cuboids in Random cubes. In the case of Prefix cubes, the total number of fetched cuboids does not change significantly when the number of materialized cuboids is increased; only the dimensionality of projection increases, as can be seen in Figure 6.12b. The dimensionality of the projections does not significantly impact the solve time of IPF solvers, and therefore, we don't observe a significant impact on solve times in the case of Prefix cubes.

Figure 6.28b shows the impact of the number of materialized cuboids on the approximation quality of the result represented by the error. As before, all IPF solvers produce nearly identical results; therefore, the errors are also nearly identical. The error decreases with an increase in the total number of materialized cuboids due to the additional information available from the increased number of fetched cuboids in the case of Random cubes and the increased dimensionality of projections in the case of Prefix cubes. △

Experiment 6.16 *Varying Minimum Dimensionality of Materialized Cuboids on IPF Solvers in Batch Mode*

We now study the impact of the dimensionality of materialized cuboids on the execution time as well as the approximation quality of the IPF solvers run in batch mode. We fix query dimensionality to 10 and run 100 queries on four data cubes – NYC Random, NYC Prefix, SSB Random, and SSB Prefix. The total number of materialized cuboids is set to 2^{15}, and the minimum dimensionality of the materialized cuboids is varied.

Figure 6.27c plots the average solve time for various IPF solvers in batch mode for different values of minimum dimensionality of materialized cuboids. The histogram for cuboid dimensionality for this experiment is shown in Figure 6.14, and its impact on prepare and fetch time is discussed in Experiment 6.6. We also observe in Figure 6.14b that there is only a slight increase in the number of fetched cuboids as the minimum dimensionality is increased. Consequently, the solve time too increases only slightly, and the increased dimensionality of the projections has a negligible impact on them.

The impact of the dimensionality of materialized cuboids on the batch mode error of various IPF solvers is shown in Figure 6.28c. The increased projection sizes available for the solvers to approximate the query as a result of increased minimum dimensionality of the materialized cuboids leads to a decrease in the error for all four data cubes. △

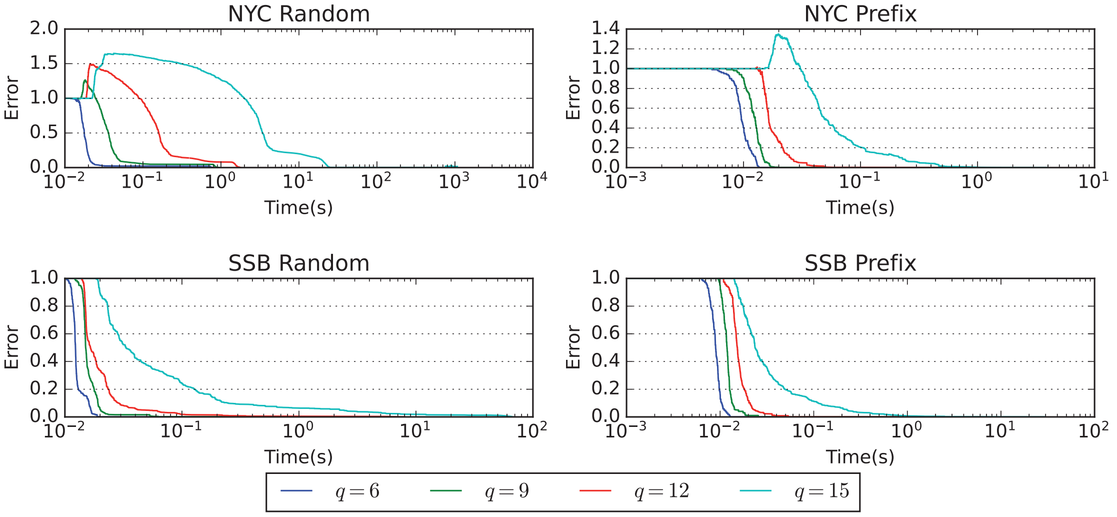


Figure 6.29: Improvement of error over time for moment-based IPF solver in online mode for various query dimensionality

Experiment 6.17 *Online Experiment for IPF Solver Varying Query Dimensionality*

Finally, we study how the error changes with time when queries are run on the IPF solver in online mode. For this experiment, we choose the moment-based IPF solver without the optimization that uses a minimum spanning tree to find the optimal cuboid processing order. The cuboids are simply processed in the order in which they are fetched. We run 100 queries each of various dimensionality on NYC Random, NYC Prefix, SSB Random, and SSB Prefix data cubes. These data cubes are built with parameters 2^{15} for the number of materialized cuboids and 14 and 18, respectively, for the minimum dimensionality of materialized cuboids for SSB and NYC data sets. After each cuboid is fetched, the iterative update is performed to scale the approximate query result using all cuboids fetched until that point. Once convergence is achieved, the cumulative time since the start of the experiment and a copy of the approximate query result is stored.

Figure 6.29 shows the average errors of the approximate answers yielded by the IPF solver at various times throughout the experiment. The results are comparable to those from running these queries on the moment solver in online mode for the Prefix data cubes. The average error is less than 1% in under a second for query dimensionality as high as 15. However, for the Random data cubes, the IPF solvers take longer and also produce poorer approximations. The order in which cuboids are processed plays a very significant role in determining both the total execution time as well as the error for IPF solvers. Because the expected size of the intersection of dimensions of the fetched cuboids with the query is very low, it is possible that some dimensions are never seen among the first few cuboids. This causes a very poor initial approximation for the queries on Random data cubes. For query dimensionality 15, the expected average error is worse than 100% for the NYC Random cube and around 8% for the SSB Random cube after 1 second of execution. △

6.5 Comparing Different Solvers

In this chapter so far, we have seen four different classes of solving techniques. We will now compare and contrast the best variant from each class of solving techniques. Among the linear programming solvers, we choose the hybrid technique that uses the simplex solver (Algorithm 6) when the degree of freedom is low and the simple bounds solver (Algorithm 5) when it is high. We choose the improved moment solver from the class of solvers that use moment extrapolation (Algorithm 13). Finally, among the IPF solvers, we choose the one that constructs a minimum spanning tree (Algorithm 17).

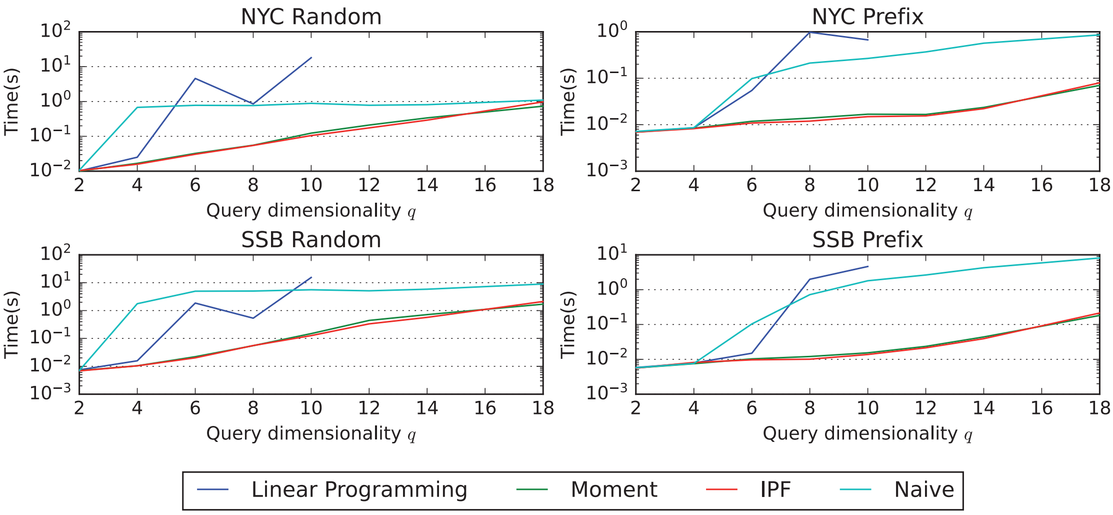


Figure 6.30: Average execution time in batch mode for each solving technique for various query dimensionality

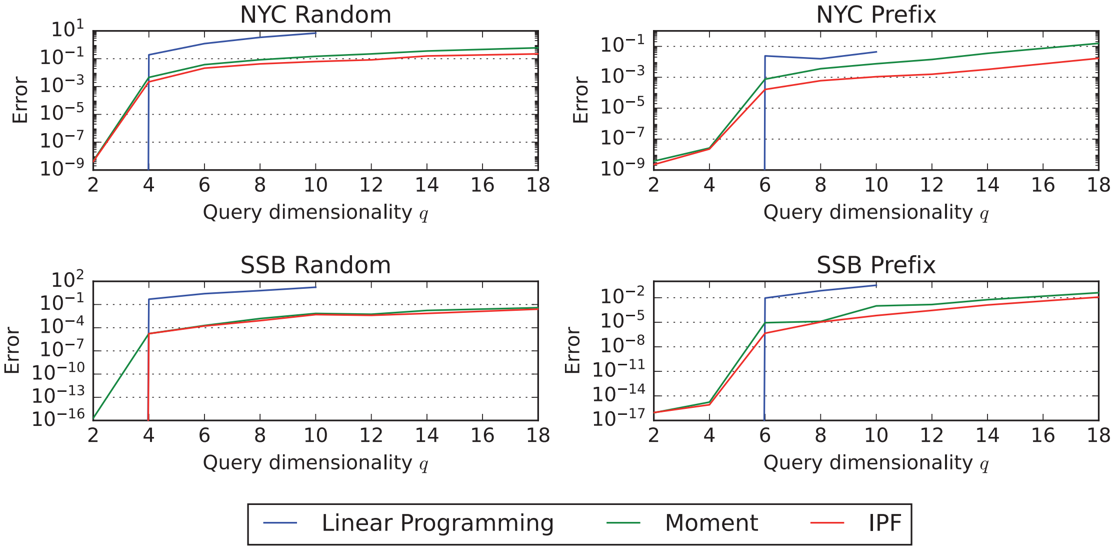


Figure 6.31: Average approximation error in batch mode for each solving technique for various query dimensionality

Experiment 6.18 *Varying Query Dimensionality on all Solvers in Batch Mode*

We will compare execution time and the approximation error for the chosen solvers for various query dimensionality. We run 100 queries of different dimensionality on the four data cubes we have used throughout this chapter. For the NYC data set, we run queries on data cubes constructed with both Prefix and Random strategy with minimum dimensionality set to 18 and the total number of cuboids set to 2^{15}. Similarly, for the SSB data set, we run queries on data cubes constructed using both strategies with minimum dimensionality 14 and the total number of cuboids 2^{15}. We limit the linear programming solver to run on only queries with dimensionality up to 10, beyond which it simply takes too much time.

Figure 6.30 shows the average execution time that includes prepare, fetch, and solve times for each solver. In the data cubes built using the Random strategy, the total time for the naive solver increases with the query dimensionality first and then plateaus as the base cuboid is projected to answer every single query. The linear programming solver is able to find bounds for the query results before the naive solver can answer queries exactly for query dimensionality near $4-6$ but is futile for queries with greater dimensionality. Both moment and IPF solvers take a similar amount of time much lower than the time the naive solver takes. In the case of the NYC Random data cube, the average execution times for these solvers become comparable to that for the naive solver around query dimensionality 18, but for the SSB Prefix data cube, they both are nearly two orders of magnitude lower than the naive solver time even for query dimensionality 18.

Figure 6.31 shows the approximation errors for the chosen solvers. The naive solver answers queries exactly and always has error 0. In the case of the linear programming solver, we use the normalized cumulative span of the intervals for each query variable as the stand-in for error. The figure shows that the linear programming solver is very good at finding tight bounds for the query variables when the query dimensionality is lower than 4 for Random cubes and 6 for Prefix cubes, but the bounds become very wide very quickly as the query dimensionality increases. Both moment and IPF solvers approximate query results extremely well, with the IPF solver being the better one, even producing results with one magnitude lower error compared to the moment solver in the case of NYC Prefix cubes. △

Experiment 6.19 *Online Experiments Varying Query Dimensionality*

Next, we compare the performance of the solvers in online mode. We skip the linear programming solver and use the moment-based IPF solver without the minimum spanning tree optimization for the online experiments. We fix query dimensionality to 12 and run 100 queries in the online mode of various solvers on NYC Random, NYC Prefix, SSB Random, and SSB Prefix data cubes. The number of materialized cuboids in each data cube is set to 2^{15} and the minimum dimensionality is set to 14 for SSB and 18 for NYC data cubes. After fetching any cuboid, the solution returned by each solver and the time are recorded for each query.

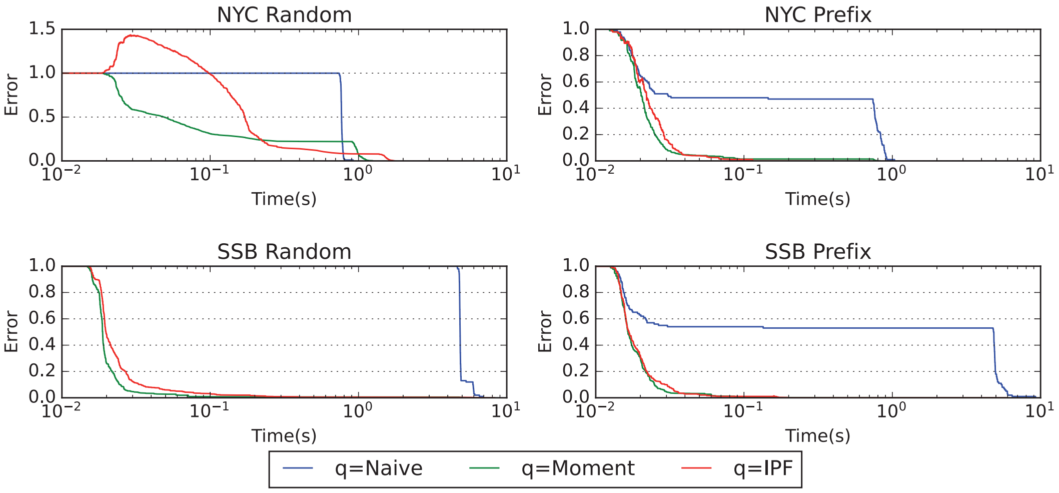


Figure 6.32: Average error at various times for each solver in online mode

Figure 6.31 shows the average error for each solver at various times throughout the experiment. We observe that the moment and IPF solvers behave nearly identically in all cases except the NYC Random data cube. For this data cube, the IPF solver performs very poorly initially due to no information on some of the dimensions of the query. This problem does not arise for the moment solver as it includes the information from all 1-D marginals in its transformations. The naive solver yielded answers by projecting the base cuboid nearly always in the case of Random cubes. The average time for answering a query using the naive solver is around 1 second for NYC cubes and 7 seconds for SSB cubes. In the case of NYC Random, within the average time the naive solver takes to return an exact answer, the other solvers have an average error between 10% and 20%. On the other hand, in the case of SSB Random, the other solvers have an average error very close to zero within a few hundred milliseconds.

In Prefix cubes, the naive solver can answer a few queries by projecting a cuboid other than the base cuboid. This means that, for some queries, the naive solver error goes down to zero very quickly, and the average error goes down as well. The other two solvers have their average errors go down to nearly zero within the first few hundred milliseconds, which is two orders of magnitude faster than the time it takes for the naive solver in the case of the SSB Prefix cube, and nearly one order of magnitude faster in case of NYC Prefix cube. △

This chapter described and evaluated four classes of solving techniques implemented in Sudokube. The experiments show that both the naive and linear programming solvers cannot answer queries within an interactive response time for queries with more than 4–6 dimensions, but the moment and IPF solvers can give very accurate approximations from the materialized projections very quickly. We also observe that the approximation quality is better for SSB data cubes than NYC data cubes, as the former has a higher budget for materialization while having fewer dimensions in total. Similarly, the approximations are better for queries run on Prefix data cubes compared to those run on Random data cubes owing to the smaller space of cuboids and queries to choose from.

6.6 Related Work

This section explores previous studies that are connected to the solving techniques discussed in this chapter.

In the context of probability distributions, the characterization of all joint distributions that satisfy some specified moments is studied by Fontana et al. in [34]. They construct a set of linear equations similar to ours and propose a characterization of the solution space using the extremal rays of a cone representing this space. While this characterization is more elegant from a theoretical perspective compared to our approach for finding maximum and minimum values for each query variable individually, it is not feasible for interactive-time querying. Computing extremal rays is computationally expensive [1], [27] and would not scale well to queries more than a few dimensions as our experience with linear programming solver shows.

Our characterization of cuboids in terms of their moments and the relationship between them and the aggregated values were first established in the context of probability distributions in [99]. Bahadur [9] proposed a similar representation for joint probability distributions in terms of generalized correlations that are normalized versions of central moments. Specifically, the central moment $\sigma(\boldsymbol{x})$ and the corresponding Bahadur coefficient differ only by a factor involving products of variances $\theta_i\overline{\theta}_i$ of Bernoulli random variables X_i for $i \in \mathbb{1}_{\boldsymbol{x}}$. Similarly, the central moments are closely related to Fourier coefficients of boolean functions [80], [81] as well. Therefore, setting unknown higher-order central moments of a cuboid to zero produces the same effect as ignoring the corresponding coefficients in these alternative representations.

Concerning graphical models, [76] solves a similar problem, but they rely on low-treewidth properties that we typically do not have in our scenario. Notably, there also exist methods beyond graphical models and moment-based reconstructions. For example, [51], [61] describe a method for reconstructing joint distributions from only two- or three-dimensional marginals based on tensor decompositions.

7 Materialization Strategy

Given a base cuboid containing hundreds, if not thousands, of binary dimensions, precomputing and materializing all its projections is infeasible in terms of time and space. The purpose of a materialization strategy is to decide, given a storage budget, what cuboids are ideal for materialization. Sudokube relies on having a good materialization strategy that yields the best information to approximate queries of some expected pattern quickly and accurately. In this chapter, we explore what a good materialization strategy is.

7.1 Number of Materializable Cuboids

We will now examine how many cuboids of various levels of dimensionality can be materialized when given a storage budget.

To quantify the storage budget, we introduce the concept of a budget factor. Given a budget factor b and a base cuboid containing 2^d non-zero cells, we are interested in the number of cuboids we can materialize so that the total storage cost does not exceed that for 2^{d+b} cells. As we shall see in the next chapter, all cells do not have the same storage cost. In dense storage formats, we store only measure values for every cell, whereas, in sparse storage formats, we store the keys along with the measure values for every cell. We ignore this difference in this chapter to simplify the analysis.

We established in Section 3.3 that the expected support size for a random k-dimensional projection of an n-dimensional binary cuboid with 2^d non-zero cells is $2^k(1 - e^{2^{d-k}})$. In Figure 3.2, we compare the expected size with the actual value from experiments for $d = 20$ and $n = 64$. Figure 7.1 plots the expected values for relative size and density of a random k-dimensional projection for different values of d. This figure shows that the phenomenon where a k-dimensional projection is dense for $k < d$ and really sparse for $k > d$ occurs for other values of d as well.

We repeat the experiments on the datasets NYC ($n = 429$, $d \approx 27$) and SSB ($n = 188$, $d \approx 29$). Figure 7.2 plots the expected size of a random k-dimensional projection derived analytically

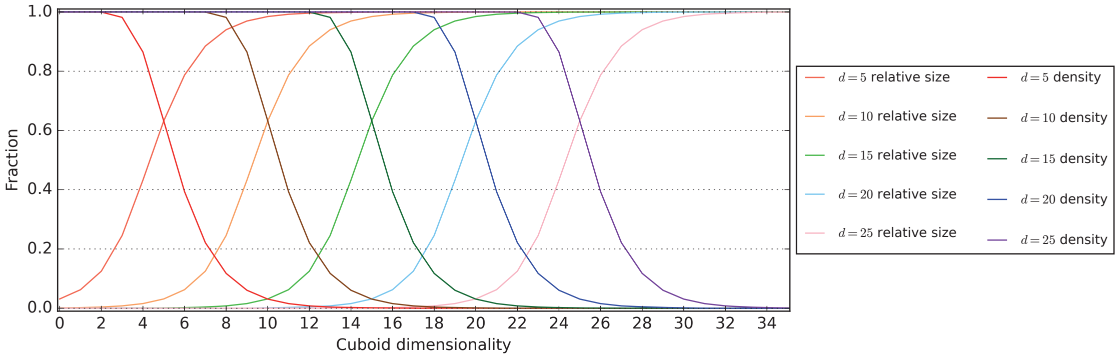


Figure 7.1: Expected relative size and density of a random k-dimensional projection of a base cuboid with support size 2^d

against the average size of 100 randomly selected cuboids from the dataset. We compare against the average sizes for the case when k dimensions are chosen uniformly at random and when k dimensions are selected among prefixes of binary dimensions encoding some columns. We observe that the size of random cuboids matches closely with the expected value, while the prefix cuboids are sparser than anticipated. Nevertheless, we will continue to use the analytically derived formula for computing the storage costs in further analysis. Emboldened by the experiment results, we can approximate the expression for the size of a k-dimensional cuboid as 2^k for $k < d$ and 2^d for $k \geq d$.

Given the simplified cost for storing a random k-dimensional cuboid, the expected number of cuboids that we can materialize subject to a budget factor of b is given by

$$N(k,n,d,b) = \begin{cases} \min\left(\binom{n}{k}, 2^{d+b-k}\right), & \text{if } k < \text{d} \\ \min\left(\binom{n}{k}, 2^{b}\right), & \text{otherwise} \end{cases} \tag{7.1}$$

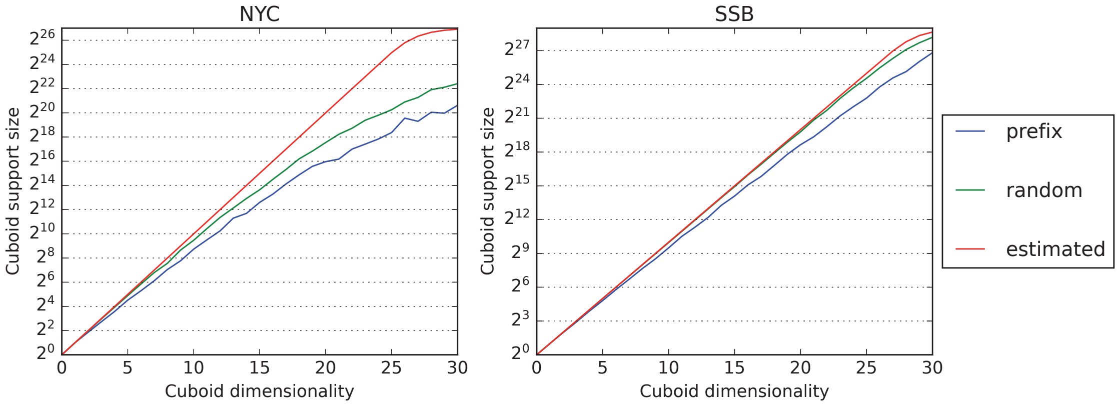


Figure 7.2: Expected support size vs. actual size for cuboids of different dimensionality whose dimensions are chosen following the prefix and random strategy

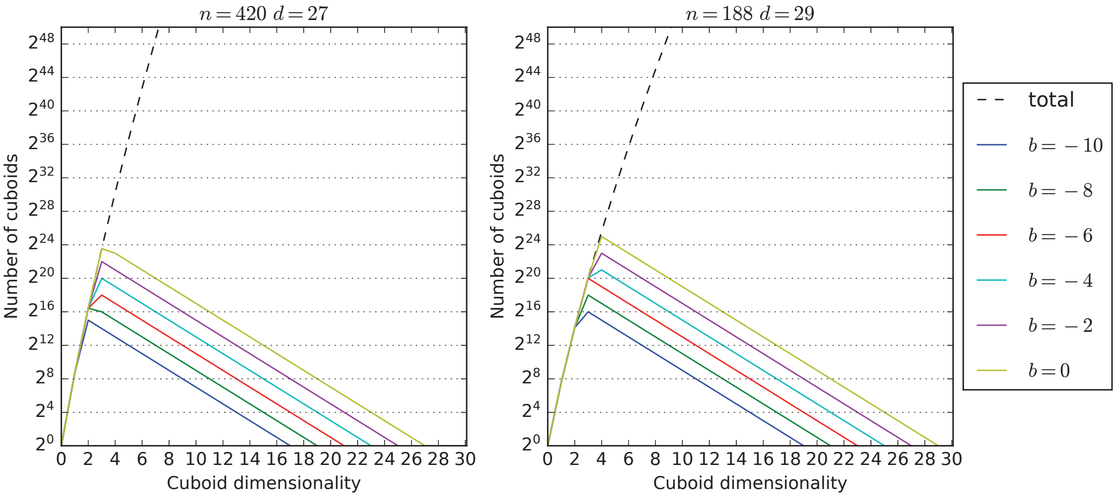


Figure 7.3: The number of materializable cuboids for each cuboid dimensionality for different budget factors for NYC and SSB datasets

Figure 7.3 shows the number of cuboids we can expect to materialize when the entire budget is allocated to materializing cuboids of a single dimensionality when parameters n and d are similar to those in NYC and SSB datasets. The total number of cuboids in the space from which one can pick cuboids for materialization is also shown in the figure. It is clear that we can materialize all cuboids only up to dimensionality around 4 while restricting the total storage cost to be comparable to that of the base cuboid ($b \approx 0$). For $k > 5$, only a tiny fraction of the cuboids can be materialized.

7.2 Score Functions

Now that we have examined the storage costs for cuboids of varying dimensionality and estimated the number of cuboids one can materialize given a storage budget, we analyze different strategies for picking which cuboids to materialize. We associate a materialization strategy $\mathscr{M}$ with the space of all possible sets of materialized cuboids M that this strategy can produce. We evaluate the utility of a given materialization strategy over a workload $\mathscr{Q}$ defined as the space of all possible queries Q. Each materialization strategy has different utility for different solvers. Ideally, the best materialization strategy for a solver picks cuboids that it would find most useful to answer queries in a given query workload $\mathscr{Q}$. The utility of a cuboid is very complex to analyze, so we assign simple score functions to any set of cuboids produced by some materialization strategy. For a given solver S, we assign solver-specific score $\text{Score}_S\{M, Q\}$ for every set of materialized cuboids M and query Q that reflects what S considers to be useful to answering Q. Then, we define the expected utility of a materialization strategy over the query workload $\mathscr{Q}$ as

$$\text{Utility}_S\{\mathscr{M}, \mathscr{Q}\} = \mathop{\mathbb{E}}_{Q \in \mathscr{Q}} \left[\mathop{\mathbb{E}}_{M \in \mathscr{M}} \left[\text{Score}_S\{M, Q\}\right]\right]$$

A potential candidate for the score function for solver S is the negative error of the result of query Q produced by S using the cuboids in M. The materialization strategy with the lowest error yields the maximum score for this score function. But the error is too complex to be analyzed directly, so we use scoring functions that loosely model the negative of the error.

To reduce the scope of our analysis, we only analyze materialization strategies that pick cuboids of some fixed dimensionality m and workloads containing queries of some fixed dimensionality q. We also apply approximations from the previous section. We approximate the number of cuboids N that we can materialize among cuboids of dimensionality m using Equation (7.1). We will analyze the strategies with $m > m_0$, where m_0 is the maximum dimensionality for which the storage budget allows the materialization of all possible cuboids of that dimensionality. For $m \leq m_0$, the materialization strategy is clear – select all cuboids for materialization. Based on Figure 7.3, we also assume that the number of cuboids that we choose for materialization is much smaller compared to the total number of possible cuboids that we do not need to differentiate between strategies that select cuboids with replacement from those that do it without replacement. In reality, they pick cuboids without replacement, but for the sake of analysis, we assume they pick cuboids with replacement.

Throughout this chapter, we will use the notation $[\![P]\!]$ to represent value 1 when some predicate P is true and 0 otherwise.

7.2.1 Score Function for Naive Solver

A cuboid C_I is relevant to the naive solver to answer a query Q only if $I \supseteq Q$. For a fixed dimensionality, several such cuboids may answer a given query, but having more than one such cuboid yields no additional benefit to answering this query. Therefore, we define the score function for the naive solver as the following expression which evaluates to 1 when some dimension set $I \in M$ is a superset of query Q, and 0 otherwise.

$$\text{Score}_{\text{naive}}\{M, Q\} = \max_{I \in M} \left[\!\left[I \supseteq Q \right]\!\right] \tag{7.2}$$

We will now estimate the expected utility of the materialization strategy $\mathcal{M}$ that picks cuboids of dimensionality m uniformly at random to answer queries of dimensionality q using the naive solver using this score function. Let Z be a random variable that denotes the number of cuboids in M that are supersets of Q.

$$Z := \left| \left\{ I \mid I \in M \text{ and } I \supseteq Q \right\} \right|$$

Then, the score function can be rewritten as

$$\text{Score}_{\text{naive}}\{M, Q\} = [\![(Z > 0)]\!]$$

The number of cuboids of dimensionality m that are supersets of query Q does not depend on Q itself, only its size q. This number equals $\binom{n-q}{m-q}$, the number of different ways of choosing $m-q$ dimensions after fixing q dimensions of the query from the remaining $n-q$ dimensions. Therefore, we can say that the random variable Z_{naive} follows a hypergeometric distribution with population size $\binom{n}{m}$, the number of events that lead to success equal to $\binom{n-q}{m-q}$ and number of trials equal to the number of cuboids we pick for materialization N. Based on our assumptions, we approximate it using a binomial distribution with success probability obtained by dividing the number of successful events by the size of the population.

$$\Pr(Z=k) = p^{k}\dot{}(1-p)^{k}, \qquad \text{where } p = \frac{\binom{n-q}{m-q}}{\binom{n}{m}}$$

The expected utility of the materialization strategy for the naive solver is given by

$$\begin{aligned}
\text{Utility}_{\text{naive}}\{\mathscr{M},\mathscr{Q}\} &= \mathbb{E}_{Q\in\mathscr{Q}}\left[\mathbb{E}_{M\in\mathscr{M}}\left[\,[\![(Z>0)]\!]\,\right]\right] \\
&= \sum_{Q\in\mathscr{Q}}\sum_{M\in\mathscr{M}}\left([\![(Z>0)]\!]\cdot\Pr(M)\right)\cdot\Pr(Q) \\
&= \sum_{Q\in\mathscr{Q}} \Pr(Z>0)\cdot\Pr(Q) \\
&= \sum_{Q\in\mathscr{Q}} \left(1-\Pr(Z=0)\right)\cdot\Pr(Q) \\
&= \sum_{Q\in\mathscr{Q}}\left(1-\left(1-\frac{\binom{n-q}{m-q}}{\binom{n}{m}}\right)^{N}\right)\cdot\Pr(Q) \\
&= \left(1-\left(1-\frac{\binom{n-q}{m-q}}{\binom{n}{m}}\right)^{N}\right)
\end{aligned}$$

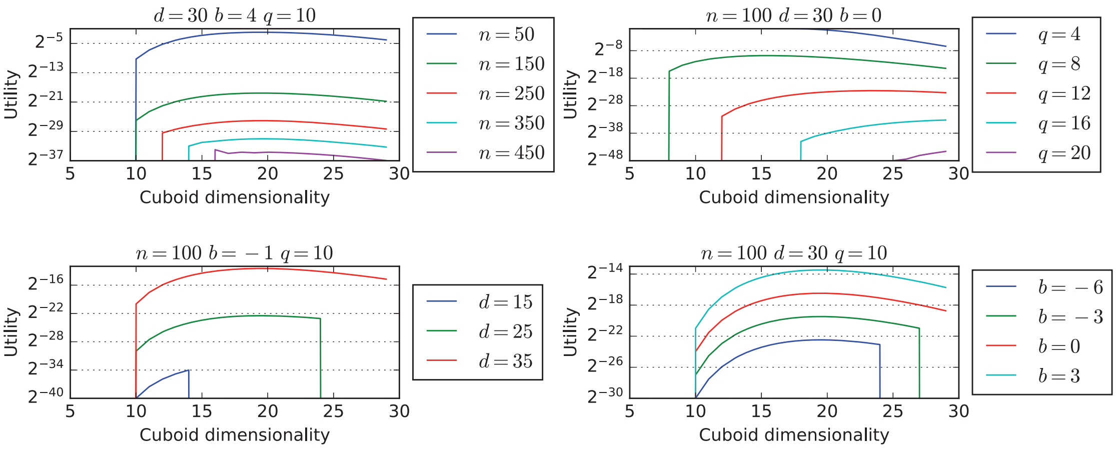


Figure 7.4: Expected utility of a materialization strategy for various cuboid dimensionality for the naive solver

Experiment 7.1 *Finding Cuboid Dimensionality for Maximizing Utility for Naive Solver*

Figure 7.4 shows the expected utility for the naive solver when the cuboid dimensionality m picked by the materialization strategy is varied. The utility for some specific dimensionality m depends on four parameters, the total dimensionality n, the support size of base cuboid 2^d, the budget factor 2^b, and the query dimensionality q. Each subfigure plots the utility varying one of these parameters keeping the other three fixed. It can be observed that for a particular query dimensionality q, the utility is maximum for $m = 2q$, suggesting that the best materialization strategy for picking cuboids with a fixed dimensionality is the one with dimensionality twice that of the query. We also observe that even the maximum utility is very low within the range of dimensionality we analyze, indicating that the naive solver has nearly no chance of answering queries when any such materialization strategy is applied. △

7.2.2 Score Function for Approximating from Projections

Next, we look into score functions for the solvers that approximate query cuboids from their projections, with a focus on the moment solver as the representative for this class of techniques. Like the case for the naive solver, we will design the score function $\text{Score}_{\text{moment}}\{M, Q\}$ to be negatively correlated to the error produced by the moment solver while approximating Q using the materialized cuboids in M.

One of the metrics we know that affects the error of the moment solver is the number of projections of Q that are available from M. Let us define the set $S(M,Q)$ as

$$S(M,Q) = \{I \cap Q \mid I \in M\}$$

We will define several score functions based on $S(M,Q)$, examine their effectiveness in modeling the error, and estimate the expected utility for different materialization strategies $\mathscr{M}$ on different query workloads $\mathscr{Q}$. The *power*, *weighted*, *multiset* and *union* score functions are defined as

$$\text{Score}_{\text{power}}\{M,Q\} = \sum_{J \in S} 2^{|J|} \qquad \text{Score}_{\text{weighted}}\{M,Q\} = \sum_{J \in S} |J| \cdot 2^{|J|}$$

$$\text{Score}_{\text{multiset}}\{M,Q\} = \sum_{I \in M} 2^{|I \cap Q|} \qquad \text{Score}_{\text{union}}\{M,Q\} = \left| \bigcup_{J \in S} 2^J \right|$$

Before we analyze these scores functions to derive their expected values, let us define two random variables Y_J and Z_J for every $J \subseteq Q$ as follows

$$Z_J = \left|\left\{I \mid I \in M \text{ and } I \cap Q = J\right\}\right|$$
$$Y_J = \left|\left\{I \mid I \in M \text{ and } I \subseteq J\right\}\right|$$

All of these scores defined above can be expressed in terms of Y_J and Z_J as follows

$$\text{Score}_{\text{power}}\{M,Q\} = \sum_{J \subseteq Q} 2^{|J|} \cdot \llbracket (Z_J > 0) \rrbracket \qquad \text{Score}_{\text{weighted}}\{M,Q\} = \sum_{J \subseteq Q} |J| \cdot 2^{|J|} \cdot \llbracket (Z_J > 0) \rrbracket$$

$$\text{Score}_{\text{multiset}}\{M,Q\} = \sum_{J \subseteq Q} 2^{|J|} \cdot Z_J \qquad \text{Score}_{\text{union}}\{M,Q\} = \sum_{J \subseteq Q} \llbracket (Y_J > 0) \rrbracket$$

We will now define success events associated with random variables Z_J and Y_J and estimate their count. First, we shall estimate, for any given set J, the number of possible sets I such that $|I| = m$ and $I \cap Q = J$. Picking such a set I constitutes a success event for the random variable Z_J. Such sets can be constructed by picking $m - |J|$ elements to be added to J from a pool that avoids all elements of Q. Therefore, the number of such sets is given by $\binom{n-q}{m-|J|}$. The random variable Z_J follows a hypergeometric distribution with population size $\binom{n}{m}$, number of events that lead to success $\binom{n-q}{m-|J|}$ and number of trials equal to the number of cuboids picked for materialization N. We will approximate it using a binomial distribution as before.

$$\Pr(Z_J = k) = p^k \cdot (1-p)^{N-k}, \qquad \text{where } p = \frac{\binom{n-q}{m-|J|}}{\binom{n}{m}}$$

Similarly, the success even for the random Y_J for any given J is when some m-dimensional set I is a superset of J. The number of such supersets is given by $\binom{n-|J|}{m-|J|}$. The probability distribution for variable Y_J can be approximated using a binomial distribution as

$$\Pr(Y_J = k) = p^k \cdot (1-p)^{N-k}, \qquad \text{where } p = \frac{\binom{n-|J|}{m-|J|}}{\binom{n}{m}}$$

We will now derive the expected utility of any score function f that only depends on the size of J, the dimensionality m of cuboids being materialized by the strategy $\mathscr{M}$ and the dimensionality q of queries in the workload $\mathscr{Q}$ as follows

$$\begin{aligned}
\text{Utility}_{\text{f}}\{\mathscr{M},\mathscr{Q}\} &= \mathbb{E}_{Q\in\mathscr{Q}}\left[\mathbb{E}_{M\in\mathscr{M}}\left[\sum_{J\subseteq Q} f(|J|,m,q)\,\llbracket (Z_J>0)\rrbracket\right]\right] \\
&= \sum_{Q\in\mathscr{Q}}\sum_{M\in\mathscr{M}}\sum_{J\subseteq Q}\left(f(|J|,m,q)\cdot\llbracket (Z_J>0)\rrbracket\right)\cdot\Pr(M)\cdot\Pr(Q) \\
&= \sum_{Q\in\mathscr{Q}}\sum_{J\subseteq Q} f(|J|,m,q)\Pr(Z_J>0)\cdot\Pr(Q) \\
&= \sum_{Q\in\mathscr{Q}}\sum_{J\subseteq Q} f(|J|,m,q)(1-\Pr(Z_J=0))\cdot\Pr(Q) \\
&= \sum_{Q\in\mathscr{Q}}\sum_{J\subseteq Q} f(|J|,m,q)\left(1-\left(1-\frac{\binom{n-q}{m-|J|}}{\binom{n}{m}}\right)^N\right)\cdot\Pr(Q)
\end{aligned}$$

$$= \sum_{Q\in\mathscr{Q}} \sum_{j=0}^{q} \binom{q}{j} f(j,m,q) \left(1 - \left(1 - \frac{\binom{n-q}{m-j}}{\binom{n}{m}}\right)^{N}\right) \cdot \Pr(Q)$$

$$= \sum_{j=0}^{q} f(j,m,q) \binom{q}{j} \left(1 - \left(1 - \frac{\binom{n-q}{m-j}}{\binom{n}{m}}\right)^{N}\right)$$

Replacing f with the correct function for the power and weighted score gives their expected utility as

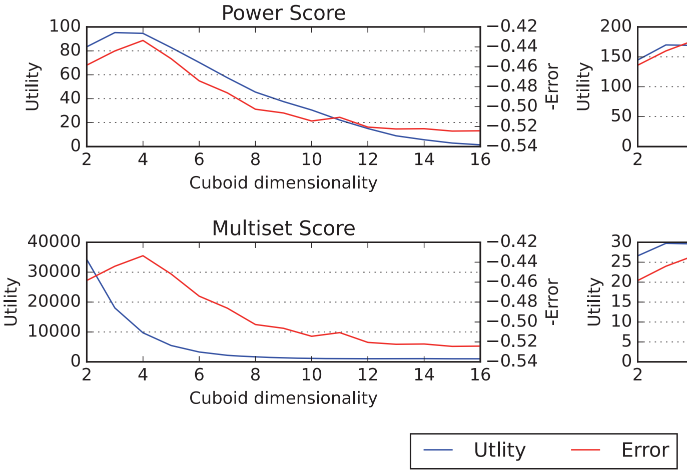
$$\begin{aligned}
\text{Utility}_{\text{weighted}}\{\mathscr{M},\mathscr{Q}\} &= \sum_{j=0}^{q} j \cdot 2^{j} \cdot \binom{q}{j} \cdot \left(1 - \left(1 - \frac{\binom{n-q}{m-j}}{\binom{n}{m}}\right)^{N}\right) \\
\text{Utility}_{\text{power}}\{\mathscr{M},\mathscr{Q}\} &= \sum_{j=0}^{q} 2^{j} \cdot \binom{q}{j} \cdot \left(1 - \left(1 - \frac{\binom{n-q}{m-j}}{\binom{n}{m}}\right)^{N}\right)
\end{aligned}$$

A similar analysis can be used to derive the expected utility according to the union score by replacing $\Pr(Z_J = 0)$ with $\Pr(Y_J = 0)$ and and the same according to multiset score by replacing $\Pr(Z_J = 0)$ by $\mathbb{E}(Z_J)$ as follows

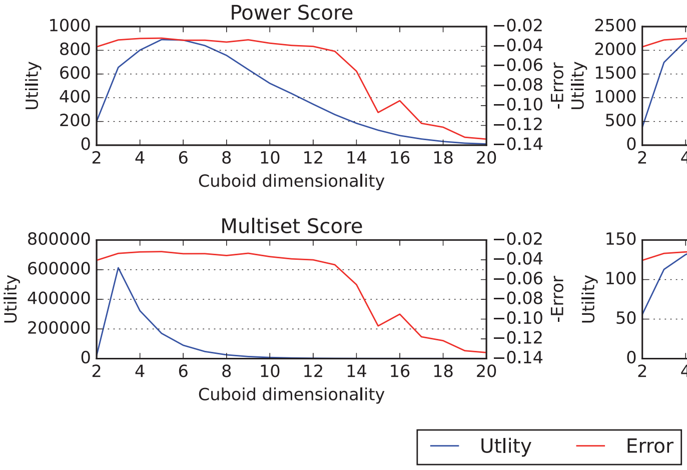
$$\begin{aligned}
\text{Utility}_{\text{union}}\{\mathscr{M},\mathscr{Q}\} &= \sum_{j=0}^{q} \binom{q}{j} \cdot \left(1 - \left(1 - \frac{\binom{n-j}{m-j}}{\binom{n}{m}}\right)^{N}\right) \\
\text{Utility}_{\text{multiset}}\{\mathscr{M},\mathscr{Q}\} &= \sum_{j=0}^{q} 2^{j} \cdot \binom{q}{j} \cdot N \cdot \frac{\binom{n-q}{m-j}}{\binom{n}{m}}
\end{aligned}$$

Experiment 7.2 *Comparing Error, Score, and Utility for Moment Solver*

In this experiment, we pick 100 queries comprising 10 dimensions chosen uniformly randomly from the NYC and SSB data sets. Because many cuboids have to be computed for small dimensions, we build data cubes on a sample of size 2^{20} from the SSB dataset. We run them on data cubes built on these datasets with various dimensionality of materialized cuboids with budget factor 2^{0} for the SSB dataset and 2^{-13} for the NYC dataset. Figure 7.5 plots the average error for these queries along with the average score and utility for each of the score functions against the materialized cuboid dimensionality. We observe that the multiset score overcounts the cuboids and has no direct correlation with the error, while the other three are reasonably similar to the error. △



(a) NYC dataset



(b) SSB dataset

Figure 7.5: Comparing error, score, and utility for moment solver for various dimensionality of materialized cuboids

Experiment 7.3 *Finding Cuboid Dimensionality for Maximizing Utility for Moment Solver*

For each score function, we plot its expected utility for different materialized cuboid dimensionality in an attempt to find the optimum dimensionality that maximizes the utility. Figure 7.6 shows the expected utility for various values of the total dimensionality of the base cuboid n. Figure 7.7 shows the expected utility for various budget factors 2^b. Figure 7.8 shows the expected utility for various support sizes of the base cuboid 2^d. Finally, Figure 7.9 shows the expected utility for various query dimensionality q. From the experiment results, we observe that the optimum dimensionality for materialization does not seem to depend on the query dimensionality and is likely to be the dimensionality m_0 where every possible cuboid of that dimensionality can be materialized while staying within budget. △

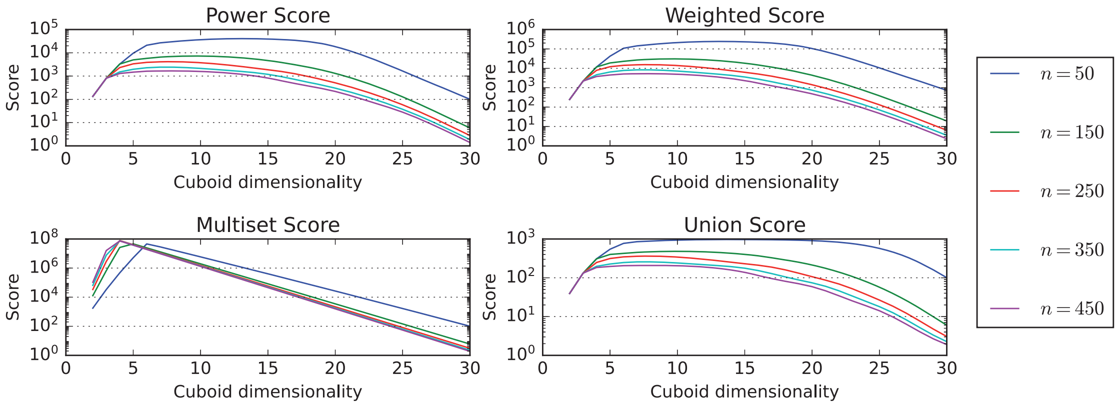


Figure 7.6: Expected utility for moment solver for various values of total dimensionality n

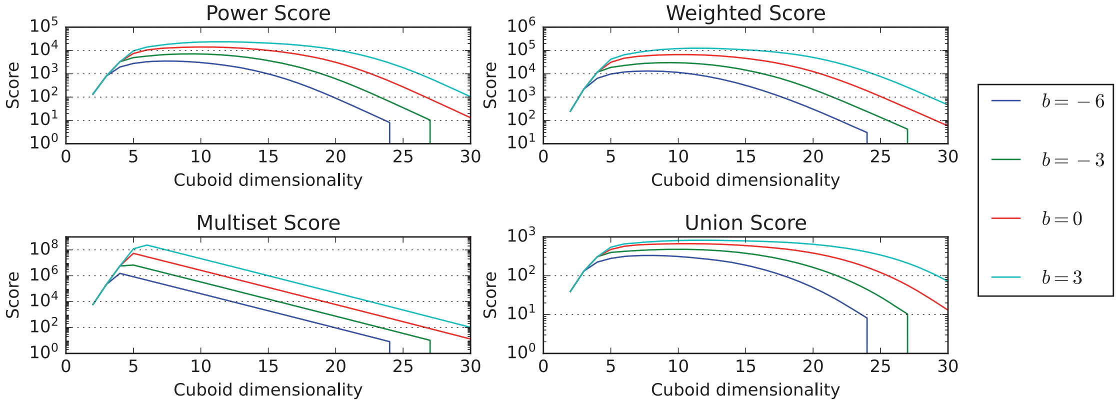


Figure 7.7: Expected utility for moment solver for various budget factors given by 2^b

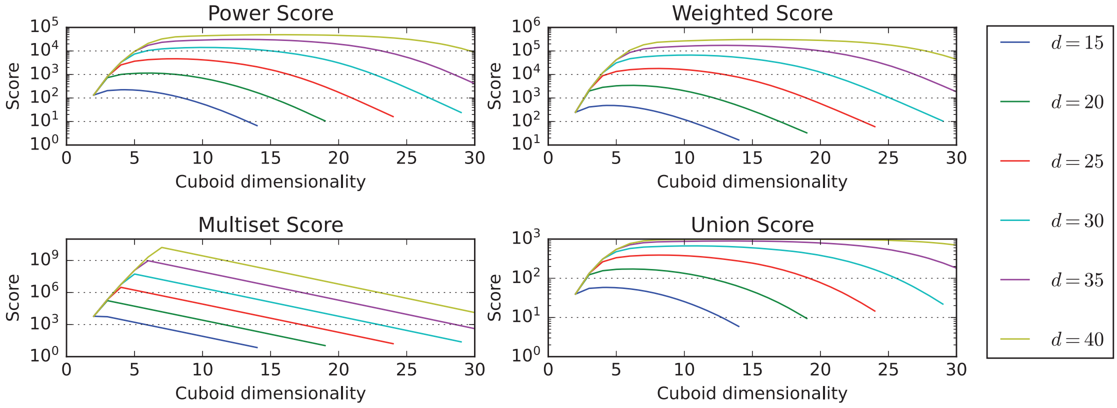


Figure 7.8: Expected utility for moment solver for various support sizes of the base cuboid given by 2^d

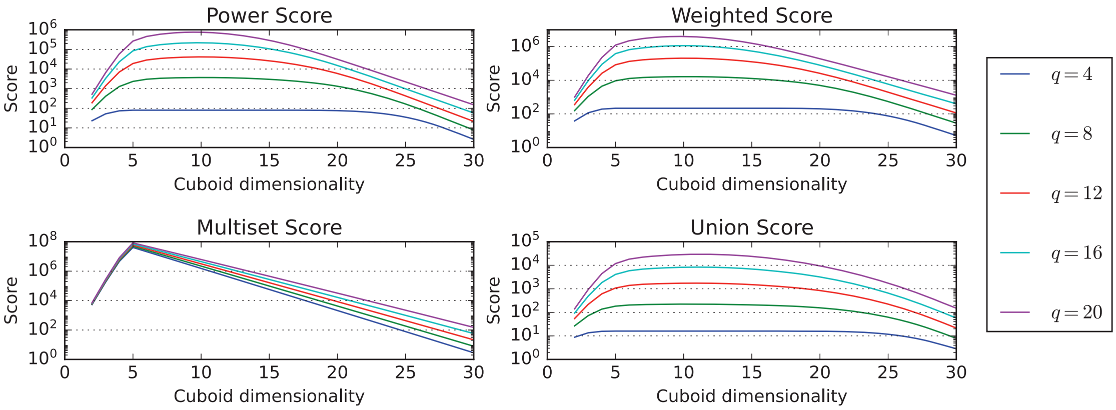


Figure 7.9: Expected utility for moment solver for various query dimensionality q

In this chapter, we studied the problem of finding the best materialization strategy for a given query workload for different solving techniques presented in Chapter 6. We derived an expression for approximating how many different cuboids we could materialize for various cuboid dimensionality given a storage budget.

We formulated a score metric to assess the performance of the naive solver considering a particular query and a set of materialized cuboids. Further, our analysis unveiled that when the materialization strategy selected cuboids of a singular dimensionality for a query workload comprising queries of fixed dimensionality, the optimal approach was to choose cuboids of dimensionality that was twice that of the query.

Extending our exploration, we conceived comparable scoring systems to evaluate the performance of the moment solver given a specific query and a set of materialized cuboids. Upon analyzing their expected utility for varying materialization strategies that target specific dimensionalities for cuboid materialization, we found that the optimal strategy appeared to be independent of the query dimensionality. The preference consistently leaned towards the dimensionality that allowed for the maximum number of cuboid materializations.

The scoring methods discussed in this chapter were designed to loosely model the inverse of the error, such that the materialization strategy scoring the highest would be anticipated to yield the lowest error for a particular query workload. Nonetheless, attaining the lowest error might not be a crucial factor if the solver requires prolonged query execution times when operating on data cubes constructed with that specific materialization strategy. Consequently, a more comprehensive scoring function that also encapsulates the execution time for queries is warranted.

8 Sudokube Backend

The Sudokube backend is responsible for storing the materialized cuboids as well as projecting those cuboids to other cuboids on demand. In this chapter, we explore various choices that go into designing a good backend for the system.

8.1 Cuboid Layout

The layout is an essential consideration when designing a backend for storing cuboids. It can affect the scalability of the system and the performance of projection operations. We discuss three different formats for storing cuboids in this section.

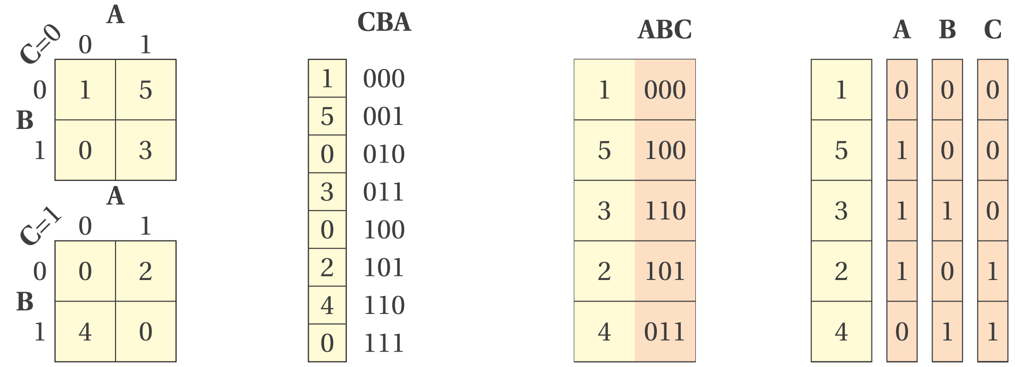
8.1.1 Dense Format

The dense format stores cuboids as an array of measure values, with the binary key for each cell encoded into the index of the corresponding element in the array. This format is highly space-efficient for low-dimensional cuboids because it does not store the keys separately, resulting in minimal storage requirements. Figure 8.1b shows how the cuboid shown in Figure 8.1a is stored in the dense format.

However, as the dimensionality of the cuboid increases, the storage space required by the dense format grows exponentially. This is because the format allocates space for each cell, regardless of its measure value. As a result, the dense format is unsuitable for high-dimensional cuboids in a system with a limited storage budget.

8.1.2 Sparse Format

The sparse format offers a more efficient method of storing high-dimensional cuboids compared to the dense format. In realistic scenarios, many cells in a cuboid have a measure value of zero, leading to significant space savings when using the sparse format to store only the cells with non-zero measure values. However, the sparse format requires that the keys be explicitly



(a) Cuboid $C_{\{A,B,C\}}$ (b) Dense (c) Sparse row (d) Sparse column

Figure 8.1: Cuboid $C_{\{A,B,C\}}$ stored in dense, sparse row and sparse column formats

stored along with the measure values, which could result in higher storage requirements for low-dimensional cuboids compared to the dense format. Despite this drawback, the space savings usually outweigh the additional cost of storing the keys for high-dimensional cuboids.

In Sudokube, two variants of the sparse format are used - the sparse row format and the sparse column format. The sparse row format stores keys and measure values in contiguous blocks of memory, one for every cell with a non-zero measure value. The sparse row format is particularly useful for storing cuboids that are in the process of being constructed. This format allows for the efficient insertion of new keys with non-zero measure values as well as updating the measure value for existing keys. Figure 8.1c shows how the cuboid shown in Figure 8.1a is stored in the sparse row format.

However, it's important to note that the sparse row format is not ideal for applying projections on the cuboid. During projection, the entire key needs to be loaded before the required bits are kept and the rest discarded. This can lead to performance issues due to poor locality, particularly for high-dimensional cuboids. Therefore, switching to an alternative format once the cuboid is fully built to optimize further projections is better.

In contrast, the sparse column format stores bits of keys from the same dimension as well as the measure values in contiguous blocks of memory. The format first stores all the non-zero measure values of the cuboid, followed by the first bit of the corresponding keys, then the second bit, and so on for every dimension in the cuboid. Unlike the sparse row format, during projection, only the bits that are required need to be loaded, making it an ideal format for cuboids being projected. However, after projecting the keys, the duplicate keys need to be aggregated, which is better done in either dense or sparse row formats. Figure 8.1d shows how the cuboid shown in Figure 8.1a is stored in the sparse column format.

The choice of the storage format for a cuboid depends on the specific use case and the properties of the data. In Sudokube, the base cuboid is initially stored in a sparse row format while it is being constructed from the raw data. Once the cuboid is fully built, it is converted into a sparse column format, which is more suitable for applying projections. During projection, the output cuboid is typically converted into a dense or sparse row format depending on the cuboid size for aggregating duplicate keys. If the resulting cuboid is sufficiently sparse, it is converted back to sparse column format for long-term storage. By carefully selecting the appropriate format for each use case, Sudokube can effectively store and manage large amounts of data while providing users with efficient access and processing capabilities.

8.2 Projection

One of the most common operations performed on a cuboid is projection, which involves reducing the number of dimensions in the keys and aggregating the measure values of duplicate keys. Projection can occur at two different times: during data cube construction and during query time. During data cube construction, materialized cuboids are projected to create other cuboids, starting from the base cuboid. During query time, some of the materialized cuboids are projected to only the dimensions that are relevant to the query. An optimized projection algorithm is crucial to ensure optimal performance during both data cube construction and query time. The algorithm should be designed to efficiently project the cuboid to the desired dimensions while minimizing the amount of data that needs to be accessed and processed.

Algorithm 18: High-level algorithm for projecting a cuboid

```
input  :Cuboid $C$ with dimensions $S$ to be projected to dimensions $T$
output :Cuboid $D$ that is a projection of $C$ to $T$
def ProjectCuboidMain($C$, $S$, $T$):
    $m \leftarrow$ GenerateMask($S$, $T$)
    if $C$ is dense then
        $D \leftarrow$ ProjectCuboid($C$, $m$, $hashing$) // dense format
    else if $|T| < 32$ then
        $D \leftarrow$ ProjectCuboid($C$, $m$, $hashing$) // dense format
        $ssize \leftarrow$ size of $D$ if stored in sparse format
        $dsize \leftarrow$ size of $D$ if stored in dense format
        convert $D$ to sparse format if $ssize < 0.5 * dsize$
    else
        $D \leftarrow$ ProjectCuboid($C$, $m$, $sorting$) // sparse format
    return $D$
```

The high-level overview of the projection algorithm used by Sudokube is described in Algorithm 18. The first step in the projection algorithm is preprocessing the dimensions of the input and output cuboids to generate projection masks. Projection masks are data structures that efficiently encode the positions of the dimensions of the output cuboid among the dimensions of the input cuboid. Various projection algorithms presented in this chapter have different implementations of the projection masks based on their needs. By precomputing these masks, these projection algorithms can significantly speed up the projection of individual cells of the cuboid that happens next. Depending on the storage format, Sudokube iterates over the entries of the input cuboid and projects each key to the desired dimensions using the precomputed masks. We will discuss the projection loop for each storage format later.

Once the keys are projected, the next step is to aggregate the measure values of the duplicate keys. Sudokube incorporates two approaches for aggregating duplicate keys, namely hashing and sorting. The hashing approach is used if the output cuboid has a low dimensionality. The dense format is used to store the output cuboid where the keys are hashed using their integer values and the measure values are aggregated in place. If the input cuboid is also dense, the output dense cuboid is returned without additional processing. However, if the input cuboid is in a sparse format, Sudokube calculates the number of cells with a non-zero measure value after the projection and computes the expected size of the resulting cuboid when stored in sparse format. If the size of the resulting sparse cuboid is sufficiently smaller than the size of the dense cuboid, Sudokube converts the cuboid to a sparse format and outputs that instead.

Algorithm 19: Aggregating duplicate keys in sparse row cuboids using sorting

```
input  : Sparse cuboid $D$ which contains duplicates
output : Sparse cuboid $D$ after merging duplicates
def AggregateBySorting($D$):
    sort $D$ according to its keys
    $(rlt, wlt) \leftarrow$ two iterators over $D$ one for reading and other for writing
    advance($rlt$)
    while $rlt$ has not finished do
        $(kr, vr) \leftarrow$ peek($rlt$)
        $(kw, vw) \leftarrow$ peek($wlt$)
        if $kr$ is same as $kw$ then
            $vw \leftarrow vw + vr$ // modify in place
        else
            advance($wlt$)
            if $wlt$ and $rlt$ are at different positions then
                copy values $(kr, vr)$ into current position of $wlt$
        advance($rlt$)
    truncate $D$ at position indicated by $wlt$
```

Aggregation using hashing as described above is not feasible if the output cuboid has a high dimensionality because the dense format is not suited for storing such cuboids. Furthermore, using a generic hash table is not optimal due to poor cache locality and weak hash functions for high-dimensional binary keys. Therefore, Sudokube uses the sparse row format to store the output cuboid and resorts to sorting for aggregating duplicate keys. In this approach, for every entry in the input cuboid, after the key is projected to the desired dimensions, it is appended to the output cuboid along with the measure value. Subsequently, the output cuboid is sorted based on the keys. Sorting the cuboid ensures that duplicate keys are adjacent to one another, making it easier to aggregate them together. The aggregated cuboid is then returned in the sparse format as the output cuboid. The algorithm for aggregating a projected sparse cuboid using sorting is described in Algorithm 19.

Next, we examine the main projection loop and the masks in detail for each storage format.

Algorithm 20: Algorithm for the projection loop for a sparse row or dense cuboid

```
input  : Input cuboid $C$, projection mask $m$, projection mode
output : Output cuboid $D$ obtained by projecting $C$ using $m$
def ProjectCuboid($C$, $m$, $mode$):
    if $mode$=$sorting$ then
        $D \leftarrow$ new empty cuboid in sparse row format
    else
        $D \leftarrow$ new empty cuboid in dense format
    foreach $(k, v) \in C$ do
        $j \leftarrow$ ProjectKey($k$, $m$)
        if $mode$=$sorting$ then
            append $(j, v)$ to $D$
        else
            $D[j] \leftarrow D[j] + v$
    if $mode$=$sorting$ then
        AggregateBySorting($D$)
    return $D$
```

8.2.1 Dense and Sparse Row

The main projection loop for a cuboid stored in a dense or sparse row follows the same procedure described in Algorithm 20. The procedure differs for the two formats only in how the cells of the cuboid are iterated. If the cuboid is dense, the main projection loop iterates over the measure values of every cell in the cuboid, along with the implicit keys encoded by their positions. On the other hand, if the cuboid is stored in the sparse row format, all non-zero measure values and their keys are iterated over. In either case, the algorithm projects the keys to the desired dimensions for each cell and then uses the hashing or the sorting approach to aggregate duplicate keys. In the case of the hashing approach, the measure value is updated in place within the output cuboid stored in dense format. Whereas in the case of the sorting approach, the projected key and the measure value is appended to the output cuboid. Then, the cuboid is sorted and the duplicate keys are aggregated.

Projecting Individual Keys

The most critical step in the projection algorithm for both dense and sparse row cuboids is projecting the individual keys to the desired dimensions. As described earlier, a projection mask is used to encode the relationship between the input and the output dimensions to speed up the process. One efficient way to encode this mask is to store the positions of the output dimensions among the input dimensions. By encoding the mask as a list of bit positions, the projection algorithm can run in linear time with respect to the number of output dimensions

Algorithm 21: Simple algorithm for projecting keys

```
input  : Input dimensions $S$; Output dimensions $T$
output : projection mask represented by bit positions $P$
def GenerateMask($S$, $T$):
    $P \leftarrow \emptyset$
    foreach $t \in T$ do
        $r \leftarrow$ rank of $t$ in $S$
        $P \leftarrow P \cup \{r\}$
    return $P$
input  : Input key $i$, list of sorted bit positions $P = \{p_0, \dots, p_n\}$
output : Projected key $j$
def ProjectKey($i$, $P$):
    $j \leftarrow 0$; $b \leftarrow 0$
    foreach $k \leftarrow 0$ to $n$ do
        $p \leftarrow p_k$
        if $p^{th}$ bit of $i$ is set then
            set $k^{th}$ bit of $j$
    return $j$
```

instead of the input dimensions. Algorithm 21 describes a straightforward algorithm for projecting the keys using such a mask. For every input key, the algorithm checks the bit positions specified by the mask and sets the corresponding bits in the output key.

The aforementioned projection algorithm for keys can be further optimized to reduce the number of operations. The operation of projecting keys is similar to the compression operation on bit vectors that moves the specified bits to one end. Instead of checking and setting individual bits, moving entire blocks of bits at once is possible. An efficient algorithm for the compress operation is described in [104] that can be adapted to project binary keys. The input keys are divided into 64-bit words, and the compression algorithm is applied separately to each word. The resulting words are then combined to form the output keys.

Algorithm 22 describes the procedures to generate the masks and do the projection when the input key contains at most 64 bits. The compress algorithm works as follows for a single 64-bit word. First, the number of positions to shift each bit right in the input key to form the output key is computed. Then, these positions are encoded into 6 additional masks $mv_0, \dots, mv_5$ that indicate the bits that have to be shifted by $2^0, \dots, 2^5$ positions, respectively. Finally, during the projection, the shifting of bits is performed in at most 6 steps, one per mask. This approach significantly reduces the number of operations required to project the keys compared to the simple approach.

Example 22. *For example, if the* 23^{rd} *bit of the input key becomes the* 6^{th} *bit of the output key, then it has to be shifted right by 17* $(= 2^4 + 2^0)$ *positions. The* 23^{rd} *bit is shifted once to the* 22^{nd} *bit in the first step, and then the* 22^{nd} *bit is shifted 16 positions to the* 6^{th} *bit. This two-step process is captured by setting the* 23^{rd} *bit of* mv_0 *and the* 22^{nd} *bit of* mv_4 *to 1.*

Algorithm 22: Algorithm to generate the projection masks to compress bit vectors

```
input  : Input dimensions $S$, output dimensions $T$
output : Projection mask $m$ and 6 additional masks $mv_0, \ldots, mv_5$ denoting positions of
         bits that need to be shifted by $2^0, \ldots, 2^5$
def GenerateMask($S$, $T$):
    $m \leftarrow |S|$-bit number where a bit is 1 only if the corresponding element is in $T$
    $mk \leftarrow \sim m \ll 1$ // bits that have 0 immediately to right
    for $i \leftarrow 0$ to 5 do
        $mp \leftarrow mk \oplus (mk \ll 1)$
        for $j \leftarrow 1$ to 5 do
            $mp \leftarrow mp \oplus (mp \ll 2^j)$
        $mv_i \leftarrow mp \,\&\, m$ // bits that need to be shifted by $2^i$
        $m \leftarrow (m \oplus mv_i) \,|\, (mv_i \gg 2^i)$ // shift the bits in the mask
        $mk \leftarrow (mk \,\&\, \sim mp)$
    return $(m, mv_0, \ldots, mv_5)$
def ProjectKey($i$, $(m, mv_0, \ldots, mv_5)$):
    $j \leftarrow 0$ ; $x \leftarrow i \,\&\, m$
    for $k \leftarrow 0$ to 5 do
        $t \leftarrow x \,\&\, mv_k$ // only bits that need to be shifted by $2^k$
        $x \leftarrow (x \oplus t) \,|\, (t \gg 2^k)$ // shift bits by $2^k$
    return $j$
```

If the input key contains more than 64 bits, then the algorithm can be modified to use an array of masks, one for every 64-bit word in the input key. In such a case, the number of output bits from each word is also calculated, which will later serve as offsets while merging the results from multiple words into a single output key.

8.2.2 Sparse Column

The main loop for projecting a cuboid stored in the sparse column format is described in Algorithm 23. The sparse column format stores bits from different keys corresponding to the same dimension in a continuous block of memory, making it impossible to iterate over whole keys directly. Therefore, this procedure iterates over groups of 64 rows together so that the bits for any dimension constitute a single 64-bit word. Once the rows have been grouped in this way, projecting the keys to the desired dimensions is a matter of selecting the right word from the bits representing those dimensions. This procedure uses the simple projection mask from Algorithm 21, represented as a list of bit positions to identify the dimensions to be kept.

If the number of output dimensions is greater than 64, the bit positions are split into groups of 64, and the procedure is repeated for each group separately. Once a 64-bit word has been obtained from at most 64 output dimensions, the next step is to aggregate duplicate keys. Before duplicate keys can be identified, the projected keys need to be converted into the row format. This is done by arranging the bits of 64 rows in a 64 × 64 matrix and transposing it.

Algorithm 23: Algorithm for the projection loop for a sparse column cuboid

```
input  : Input cuboid $C$, projection mask $m$ as a list of bit positions, aggregation mode
output : Output cuboid $D$
def ProjectCuboid($C$, $m$, $mode$):
    $numWords \leftarrow$ number of rows grouped into 64-bit words
    $A \leftarrow$ new array of size 64 containing 64-bit integers
    $colGroups \leftarrow$ split bit positions in $m$ into groups of 64
    if $mode$=$sorting$ then
        $D \leftarrow$ new empty cuboid in sparse row format
    else
        $D \leftarrow$ new empty cuboid in dense format
    for $rw \leftarrow 0$ to $numWords - 1$ do
        foreach $(cwIdx, colGroup) \in enumerate(colGroups)$ do
            set all entries in $A$ to 0
            for $(c0, col) \in enumerate(colGroup)$ do
                copy $rw^{th}$ word of $col^{th}$ dimension into slot $A[63 - c0]$
            Transpose64($A$)
            if $mode$=$sorting$ then
                for $r0 \leftarrow 0$ to 63 do
                    $j \leftarrow A[63 - r0]$
                    $r \leftarrow (rw \ll 6) + r0$
                    copy $j$ into the $cwIdx^{th}$ word of the key of $r^{th}$ row of $D$
        for $r0 \leftarrow 0$ to 63 do
            $r \leftarrow (rw \ll 6) + r0$
            $v \leftarrow r^{th}$ measure value
            if $mode$=$sorting$ then
                copy $v$ into the measure value of $r^{th}$ row of $D$
            else
                $j \leftarrow A[63 - r0]$
                $D[j] \leftarrow D[j] + v$
    if $mode$=$sorting$ then
        AggregateBySorting($D$)
    return $D$
```

To transpose the 64 × 64 bit-matrix, Sudokube uses Algorithm 24 derived from [104]. It takes an array of words where each word represents 64 rows of some dimension and transforms it such that each word now represents 64 bits of some key. If the number of output dimensions is small enough, Sudokube uses the hashing approach for aggregation. For every word in the array, the 64-bit key is hashed using its integer value to identify its position in the dense output cuboid. The associated value for that row is looked up from the input cuboid and is aggregated with the existing value at the identified index within the output cuboid.

Algorithm 24: Algorithm to transpose 64 × 64 bit matrix in place

```
input :Input matrix $A$ as an array of 64-bit unsigned integers
def Transpose64($A$):
    $m \leftarrow 2^{32} - 1$ ; $j \leftarrow 32$
    while $j \neq 0$ do
        $k \leftarrow 0$
        while $k < 64$ do
            $t \leftarrow (A[k] \oplus (A[k+j] \gg j)) \,\&\, m$
            $A[k] \leftarrow A[k] \oplus t$
            $A[k+j] \leftarrow A[k+j] \oplus (t \ll j)$
            $k \leftarrow (k+j+1) \,\&\, {\sim}j$
        $j \leftarrow j \gg 1$; $m \leftarrow m \oplus (m \ll j)$
```

On the other hand, if the number of output dimensions is high, Sudokube cannot use the hashing approach and must instead use the sorting approach. For each row in the input cuboid, the projected key is copied in groups of 64 bits to the row at the same position in the output cuboid. Then the process is repeated for the measure values. Finally, the output cuboid is sorted and aggregated using the procedure described in Algorithm 19.

We will now evaluate the performance of five projection algorithms in a variety of experiments. *SparseRow Simple* and *Dense Simple* uses Algorithm 21 to project individual keys as part of projecting cuboids stored in the sparse row and dense formats using Algorithm 20. *SparseRow Optimized* and *Dense Optimized* uses Algorithm 22 to project keys using the same algorithm for projection. Finally, *SparseCol Optimized* uses Algorithm 23 to project cuboids stored in sparse column format. We omit evaluating the algorithms on cuboids stored in dense formats for high-dimensional cuboids for obvious reasons. We will also set the output dimensionality low enough so that hash-based projections are used in all cases for a fair comparison.

Experiment 8.1 *Varying Dimensionality of Input Cuboid for Projection Algorithms*

We study the impact of the dimensionality of the input cuboid on the running time of each of the five algorithms. We run the experiment for two scenarios, one where the input cuboid is low-dimensional and the other where it is high-dimensional. We generate cuboids with support size 2^{12} for the low-dimensional scenario and 2^{20} for the high-dimensional scenario. We fix the output dimensionality to 10 for both cases. We repeat the experiment 100 times each for different input dimensionality for randomly generated data and output dimensions.

Figure 8.2 shows the average projection time for each algorithm for various input dimensionality. We observe that for the low-dimensional scenario, initially, the dense format yields the fastest projection when the support size matches the number of cells. However, the time to project the cuboids in dense format increases exponentially with the input size, while it remains the same for the cuboids in sparse format. The optimized algorithms are better than the simple algorithm, and there is not much difference in time between projecting cuboids in the sparse row and sparse column formats. However, in the high-dimensional scenario, as the masks are generated for each group of 64 bits in the input cuboid, the time to project the sparse row format increases linearly with the number of such groups. On the other hand, transpose is done in groups of 64 bits of the output cuboid, and the projection time for sparse column format does not change. △

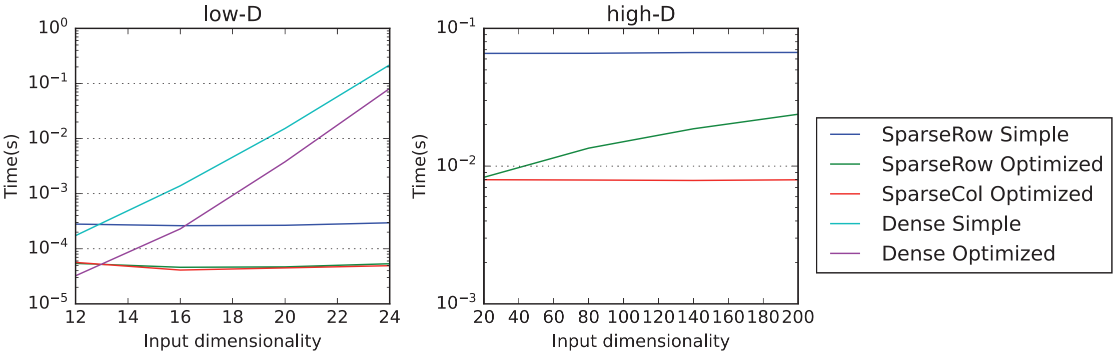


Figure 8.2: Projection times for each algorithm for various dimensionality of input cuboids

Experiment 8.2 *Varying Dimensionality of Output Cuboid for Projection Algorithms*

We now study the impact of the output cuboid dimensionality on the running times of the projection algorithms. We run the experiment for low-dimensional and high-dimensional input cuboids separately. We set the dimensionality of the input cuboid to be 200 for the high-dimensional case and 20 for the low-dimensional case. We set the support size in both cases to 2^{20}. We repeat the experiment 100 each for randomly generated data and output dimensions while varying the output dimensionality.

Figure 8.3 shows the average projection time for each algorithm for various output dimensionality. The projection time increases nearly linearly with the dimensionality of the output cuboids, both in low-dimensional as well as high-dimensional scenarios. △

Experiment 8.3 *Varying Support Size of Input Cuboid for Projection Algorithms*

Finally, we study the impact of the sparsity of the input cuboid on the running times. We run separate experiments for low and high-dimensional input cuboids. We set the dimensionality of the input cuboid to be 200 for the high-dimensional case and 20 for the low-dimensional case. We fix the output dimensionality to 10 in both cases. We repeat the experiment 100 times each for randomly generated data and output dimensions while varying the support size of the input cuboids and average the results.

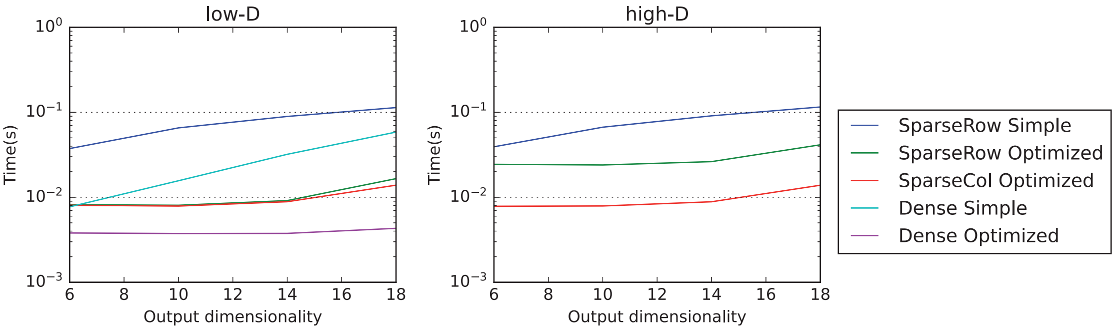


Figure 8.3: Projection times for each algorithm for various dimensionality of output cuboids

Figure 8.4 shows the average projection time for each algorithm for various sizes of the input cuboid support. Clearly, the projection times for the cuboids in dense formats are unaffected by the support sizes, as all cells have to be processed regardless of the value they contain. In both high-dimensional and low-dimensional scenarios, the projection times for cuboids in the sparse format increase linearly with the support size. We also observe again that the sparse formats are better when the data is sparse and the dense format is better when it is not. △

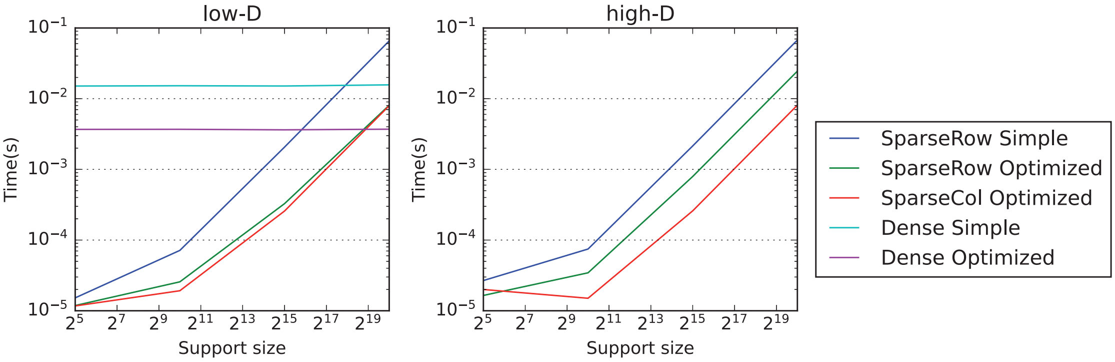


Figure 8.4: Projection times for each algorithm for various support sizes of input cuboids

8.3 Alternative Storage Layout

In this chapter, we have explored various formats for storing cuboids and algorithms for performing projection operations on them. Typically, a data cube is stored as a collection of cuboids that includes the base cuboid and several projections. This approach is most suitable for the traditional method of answering data cube queries by projecting the smallest subsuming cuboid.

However, Sudokube's alternate approach, which involves approximating queries from available projections, calls for a re-evaluation of the data cube storage strategy. Instead of using a single cuboid, Sudokube uses multiple cuboids to answer queries. However, multiple cuboids from the same data cube store redundant information, resulting in several issues.

Firstly, storing redundant information wastes storage space from a finite storage budget that could have been used to store additional information. Additionally, processing redundant information leads to slower query processing time. During query processing time, cuboids are processed to extract information from them that is suitable for the particular solver. However, since multiple cuboids contain redundant information, time spent extracting information from cuboids that will later be discarded is wasted.

To address these issues, alternative data cube storage approaches that minimize redundant information storage and reduce processing time should be considered. Such approaches enable Sudokube to provide faster and more accurate responses to data cube queries. For instance, a different approach involves a data cube storing several chosen moments along with the base cuboid instead of projections of the base cuboid.

8.3.1 Moment Store

A *moment store* backend stores selected moments of a data cube instead of storing cuboids. It uses a set-trie data structure[89] to store moments indexed by sets. A set-trie is a variant of the trie data structure modified for indexing subsets of a totally ordered set $(\mathscr{D}, \prec)$ instead of strings. Each node represents a subset of the domain $\mathscr{D}$ in a set-trie. The root node represents the empty set and the children of each node extend the set it represents by a single element. The children are sorted according to the element they store in the order specified by $\prec$.

We apply some restrictions and modify the data structure proposed in [89] to best suit our case. Firstly, we store a moment value associated with the set in each node instead of a single bit that indicates the presence or absence of the set. Secondly, we assume that before some set S is inserted into the trie along with some value v, all subsets of S have been previously inserted. Finally, we implement the trie using the first child-next sibling representation in an array. Our set-trie is simply an array of nodes, with each node storing the index of its first child and the next sibling. We use index value -1 to denote the absence of a child or a sibling.

Algorithm 25 describes the procedure to insert a set S and an associated value v into a set-trie. The set is broken up into its constituent elements and each element is used to traverse the trie in the order specified by $\prec$. Because of the restriction that all subsets of S must be inserted before S, only the last node in the path that represents the set S can be missing. This node is created and its value is set to v.

Given a query Q, the moment store returns $2^{|Q|}$ moments, with the stored moments retrieved efficiently from the set-trie and the other moments set to 0. Algorithm 26 describes an efficient algorithm to retrieve the values associated with all stored subsets of a given set from a set-trie. It takes as input a query Q and a zero-initialized array R of size $2^{|Q|}$ to store the results, the index q of the element from Q being processed, and the integer encoding r of the set represented by the path from the root to the current node. r has $|Q|$ bits, each of which indicates the presence of one element of Q in the set. Both q and r are initially 0, and node starts with the root of

Algorithm 25: Algorithm for inserting a set to a set-trie

```
input :set $S$ to be inserted into trie $T$ with value $v$
def TrieInsert($T$, $S$, $v$):
    if $T$ is empty then
        $node \leftarrow$ new node in $T$
        Initialize $node$ with $key \leftarrow$ -1, $value \leftarrow v$ , $firstChild \leftarrow$ -1, $nextSibling \leftarrow$ -1
    else
        $node \leftarrow$ node at index 0 in $T$
        foreach $k \in S$ do
            $node \leftarrow$ GetOrAddNode($T$, $node$, $k$)
        $node.value \leftarrow v$

def GetOrAddNode($T$, $node$, $k$):
    $prevChild \leftarrow$ null; $curChildID \leftarrow node.firstChild$
    $curChild \leftarrow$ node at index $curChildID$ in $T$
    while $curChildID \neq -1$ and $curChild.key < k$ do
        $prevChild \leftarrow curChild$; $curChildID \leftarrow curChild.nextSibling$
        $curChild \leftarrow$ node at index $curChildID$ in $T$
    if $curChildID \neq -1$ and $curChild.key = key$ then
        return $curChild$ // return existing child
    else
        ($newChild$, $newID$) $\leftarrow$ new node at the end of $T$
        Initialize $newChild$ with $key \leftarrow k$, $firstChild \leftarrow$ -1, $nextSibling \leftarrow curChildID$
        if $prevChild$ is null then $node.firstChild \leftarrow newID$
        else $prevChild.nextSibling \leftarrow newID$
        return $newChild$
```

set-trie. The procedure is recursive, and the first step is to store the current node's value in the result array at the position for the set represented by this node. Then every child node is processed one after the other. If a child node stores a key that is less than the current element of Q being processed, then the entire subtree of those nodes is ignored because they cannot contain any subset of Q. If a child node is found storing a key that is equal to the current element from Q, then the procedure is recursively called on that node to process its subtree. Before invoking the recursive procedure, q is advanced to process the rest of the query, and r is updated to include the q^{th} element of the query in the set. Finally, if the key of the child node is greater than the current element of the query, the element being processed is advanced until one of the other two cases becomes true or the end of the query is reached. After the procedure terminates, the array R contains all the moments from the set-trie at the right positions.

The array of moments produced by the moment store can directly be fed into the moment solver without any additional transformations to identify and remove redundant information.

Next, we investigate the effect of using this alternative backend on the approximation of query results in Sudokube. Ideally, an effective materialization strategy should guide the selection

Algorithm 26: Algorithm to retrieve values associated with every subset of a given set from a set-trie

```
input :Trie $T$, Query $Q$, array $R$ for storing results, position in query $q$, integer $r$
       representing path from root to current node $node$
def TrieQuery($T$, $Q$, $R$, $q$, $r$, $node$):
    $R[r] \leftarrow node.value$
    $childID \leftarrow node.firstChild$
    while $childID \neq -1$ and $q < |Q|$ do
        $childNode \leftarrow$ node at index $childID$ in $T$
        $k \leftarrow q^{th}$ element of $Q$
        if $childNode.key = k$ then
            TrieQuery($T$, $Q$, $R$, $q+1$, $r+2^q$, $childNode$)
            $childID \leftarrow childNode.nextSibling$
        else if $childNode.key < k$ then
            $childID \leftarrow childNode.nextSibling$
        else
            $q \leftarrow q+1$
```

of moments for storage. However, in the current implementation, we construct the moment store by extracting moments from existing data cubes and inserting them into a set-trie. We limit the moments to dimensionality up to 6 and discard the rest. This decision was motivated by the assumption that lower-dimensional moments are more likely to be useful for query results and more likely to exhibit larger values than their higher-dimensional equivalents.

Figure 8.5 lists the storage space for the largest data cubes we built on NYC and SSB datasets using Random and Prefix strategies, along with the storage cost of the moment store built using the aforementioned strategy. Given the imposed cut-off for moment dimensionality, the moment store contains strictly less information than the cuboids. Nevertheless, there might be instances where the moment store requires more storage than the cuboids, as demonstrated by the NYC Prefix case.

The primary reason behind this phenomenon lies in the data sparsity within the Prefix cubes. When a substantial number of cells possess a measure value of zero, storing cuboids in the sparse format leads to substantial space conservation. However, moments do not benefit from such sparsity and consequently may occupy more storage space compared to sparse cuboids.

In the following experiments, we will compare the execution time and the accuracy of the results when using the moment store against those obtained when using the cuboids directly.

Experiment 8.4 *Varying Query Dimensionality for Moment Solver in Batch Mode with Different Backends*

We pick 100 queries each of various dimensionality and run them on the improved moment solver in batch mode for NYC Random, NYC Prefix, SSB Random, and SSB Prefix data cubes

Dataset	Strategy	N	$d_{\min}$	Cuboid Storage (GB)	Trie Storage (GB)
NYC	Random	2^{15}	18	57.198	34.359
NYC	Prefix	2^{15}	18	17.757	18.265
SSB	Random	2^{15}	14	31.345	10.428
SSB	Prefix	2^{15}	14	17.259	3.005

Figure 8.5: Comparison of storage between data cubes and moment store

and the corresponding moment stores built from them. For this experiment, we choose the data cubes and moment stores built with the parameters listed in Figure 8.5.

Figure 8.6 shows the average times spent by the moment solver in various phases of the query execution. We observe that the combined prepare and fetch of the moment store following Algorithm 26 is much faster than both the prepare as well as fetch for the cuboid store. Furthermore, the solve times are lower when using the moment store compared to the cuboid store, even though the same solving algorithm is used in both cases. The reduction in the solving time can be explained by not having to transform cuboids to moments after fetching them.

Figure 8.7 shows the average error for the approximate answer returned by the moment solver when using the moment and cuboid stores as backends. We observe that there is a significant difference in errors in the case of Prefix cubes, but the errors are similar in the case of Random cubes. We know from the results of our previous experiments in Figure 6.10 that the expected size of the intersection of dimensions of the query and the fetched cuboids is roughly between 3 and 4 for Random cubes, but it is higher for Prefix cubes. Thus, there is nearly no loss of information for the query when we prune the moments above order 6 in the case of Random cubes, but some information is lost in the case of Prefix cubes. Regardless of the loss of information due to pruned moments, a sufficient number of moments are stored to still yield approximations with errors less than 0.02 for SSB Prefix cubes and 0.1 for NYC Prefix cubes. △

In this chapter, we studied different storage layouts and projection algorithms for cuboids and evaluated their performance in various scenarios. The best storage format for projection is the sparse column format when the support size is small compared to the number of cells in the cuboid and the dense format when it is reversed. We also saw how alternate storage representations, such as the moment store, can outperform even these optimized cuboid representations in terms of execution time. However, further research is needed in choosing the moments to be stored to minimize the error.

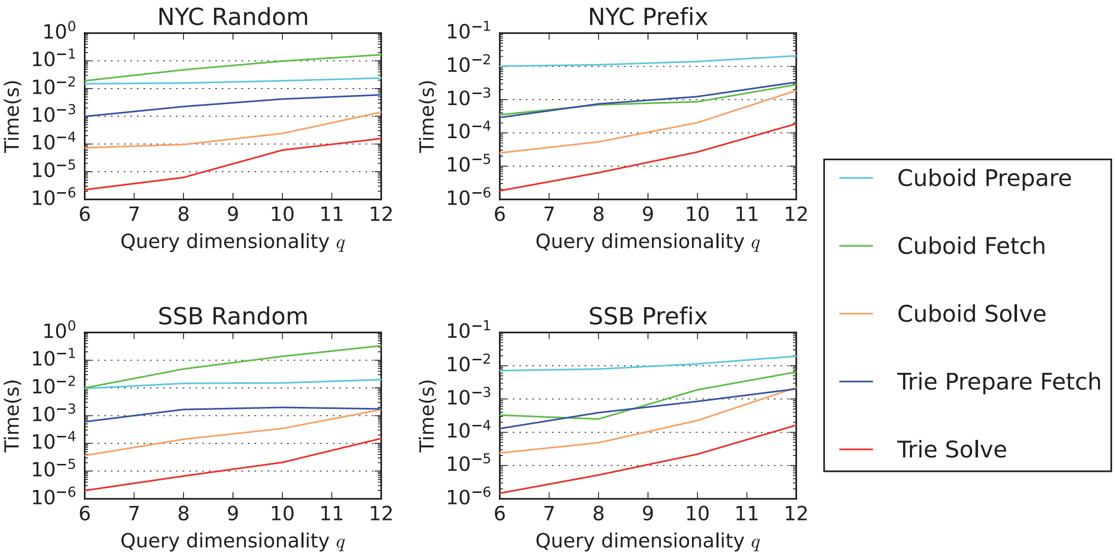


Figure 8.6: Average execution time for the improved moment solver in batch mode for various query dimensionality when using each backend

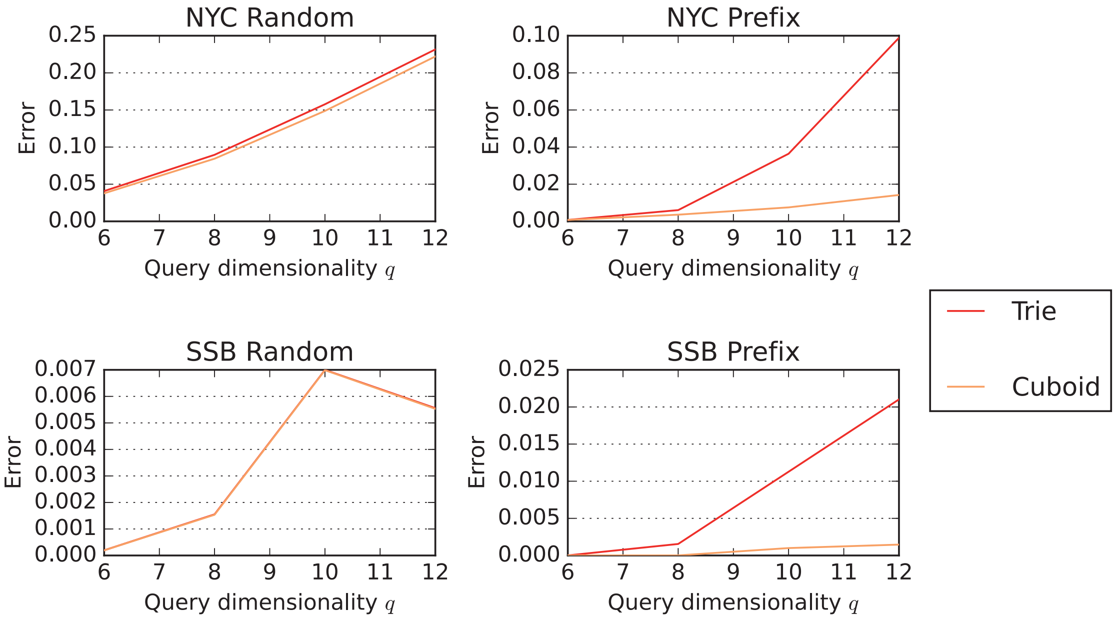


Figure 8.7: Average error for the improved moment solver in batch mode for various query dimensionality when using each backend

9 Conclusion and Future Work

This thesis explores how interactive-time queries can be made possible on large high-dimensional datasets. We explored various use cases where having high-dimensional data cubes would be very useful and studied the challenges faced by existing approaches both in terms of storage costs and query running time. We discussed how these challenges are mitigated in our system, Sudokube, and ventured into the system's architecture, the concepts behind its data representation, querying capabilities, and impressive ability to quickly approximate a query result from its projections.

We studied the limitations of current approaches in terms of both storage costs and running time of queries and discussed steps taken by Sudokube to allay these limitations. In particular, Sudokube avoids aggregating all the tuples at query time by precomputing some aggregations in data cubes and offloading the cost from query time to data cube construction time. Excessive storage costs were avoided by precomputing only some cuboids from the data cube and using efficient techniques to extrapolate missing cuboids from the precomputed ones.

Throughout this thesis, we saw how Sudokube effectively encodes data with its binary dimensions approach, simplifying the theory and implementation for storing, projecting, and extrapolating cuboids. The exposed binary dimensions allow users to run richer aggregate queries by defining groups based not just on the values of some columns but patterns of values as well. Additionally, we highlighted Sudokube's adaptability and flexibility in supporting basic query operations across static and dynamic schemas and its ability to handle hierarchical dimensions. Furthermore, we discussed the advanced query features supported in Sudokube through post-processing.

We analyzed the utility of various materialization strategies for answering queries using different solvers. Among the strategies that pick cuboids of a single dimensionality for materialization, we observed that choosing the maximum dimensionality where we can afford to materialize all the possible cuboids is likely the best choice to reduce error. Nevertheless, it remains uncertain whether this approach is the most suitable for optimizing a comprehensive metric that encompasses prepare, fetch, and solve times in addition to the error.

This thesis has shed light on the potential of Sudokube as a promising tool in the world of data analytics. Nonetheless, like any other tool, it is not without areas for possible enhancement and future development. For instance, the Sudokube core engine reconstructs entire cuboids and applies filters only during post-processing, which is wasteful. Also, expanding the range of solving techniques and supporting more query features natively in the core engine would allow Sudokube to answer a bigger class of queries with interactive speeds.

We observed in the experimental results that the moment extrapolation techniques lead to very low-quality approximations of the query result when applied directly. Perturbing the extrapolated moments to satisfy some heuristic bounds is observed to have significantly boosted the accuracy of the query results. Studying this heuristic in more detail and examining why it works could lead to discovering methods that improve the error even further.

An extension of our utility analysis to accommodate materialization strategies that select cuboids with varying dimensionality, for mixed query workloads with different levels of dimensionality, would provide insights into choosing the optimal set of cuboids to materialize in a broader range of scenarios.

An additional avenue for future research involves investigating the potential for enhancing the accuracy of the approximated query results by integrating online sampling with our current solving techniques.

On the system side, there is a significant benefit in scaling the backend to be distributed across multiple machines and leveraging multithreading to parallelize the solving algorithms. This would enable Sudokube to achieve fast response times even for larger datasets and bigger queries. Another compelling area for further research would be runtime code generation for efficient projection algorithms.

Looking forward, it would be worthwhile to explore these enhancements and other potential improvements to further refine and extend Sudokube's capabilities. It would also be interesting to test the system's performance with different types of data and queries, particularly in real-world, industry-specific applications. This could provide valuable insights into how Sudokube can be fine-tuned for optimum performance in diverse analytical scenarios.

In conclusion, Sudokube represents a significant step forward in analytical processing systems. Its robust querying capabilities, combined with its ability to efficiently handle massive datasets, position it as a highly promising tool in the field of data analytics. As we move into an era of ever-growing data, the importance of efficient, flexible, and powerful analytical tools such as Sudokube cannot be overstated.

Bibliography

[1] 4ti2 team, *4ti2—A software package for algebraic, geometric and combinatorial problems on linear spaces*. [Online]. Available: https://4ti2.github.io.

[2] S. Acharya, P. B. Gibbons, and V. Poosala, "Congressional samples for approximate answering of group-by queries", in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, ACM, May 16, 2000, pp. 487–498.

[3] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy, "The aqua approximate query answering system", in *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, 1999, pp. 574–576.

[4] P. Afshani and J. M. Phillips, "Independent range sampling, revisited again", 2019. arXiv: 1903.08014.

[5] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. 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 (EuroSys '13)*, 2013, pp. 29–42.

[6] S. Agarwal, R. Agrawal, P. Deshpande, *et al.*, "On the Computation of Multidimensional Aggregates", in *Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB '96)*, 1996, pp. 506–521.

[7] B. Babcock, S. Chaudhuri, and G. Das, "Dynamic sample selection for approximate query processing", in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, 2003, pp. 539–550.

[8] J. H. Badsberg and F. M. Malvestuto, "An Implementation of the Iterative Proportional Fitting Procedure by Propagation Trees", *Computational Statistics & Data Analysis*, vol. 37, no. 3, pp. 297–322, Sep. 2001.

[9] R. R. Bahadur, "A representation of the joint distribution of responses to n dichotomous items", 1961, pp. 158–168.

[10] E. Baralis, S. Paraboschi, and E. Teniente, "Materialized Views Selection in a Multidimensional Database", in *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB '97)*, 1997, pp. 156–165.

[11] D. Barbará and M. Sullivan, "Quasi-Cubes: Exploiting Approximations in Multidimensional Databases", *SIGMOD Record*, vol. 26, no. 3, pp. 12–17, 1997.

[12] D. Barbará and X. Wu, “Using Loglinear Models to Compress Datacubes”, in *Proceedings of the 1st International Conference on Web-Age Information Management (WAIM ’00)*, 2000, pp. 311–323.

[13] S. Basil John and C. Koch, “High-dimensional Data Cubes”, in *Proceedings of the VLDB Endowment*, vol. 15, 2022, pp. 3828–3840.

[14] S. Basil John, P. Lindner, Z. Jiang, and C. Koch, “Aggregation and Exploration of High-Dimensional Data Using the Sudokube Data Cube Engine”, in *Companion of the 2023 International Conference on Management of Data (SIGMOD-Companion ’23)*, Seattle, WA, USA: ACM, 2023.

[15] S. Basil John, P. Lindner, Z. Jiang, and C. Koch, “Fast Approximate Reconstruction of Joint Distributions from Low-Dimensional Projections”.

[16] K. S. Beyer and R. Ramakrishnan, “Bottom-Up Computation of Sparse and Iceberg CUBEs”, in *Proceedings of 1999 ACM SIGMOD International Conference on Management of Data (SIGMOD ’99)*, 1999, pp. 359–370.

[17] Y. M. Bishop, S. E. Fienberg, and P. W. Holland, *Discrete Multivariate Analysis: Theory and Practice*. Springer Science & Business Media, 2007.

[18] S. Chaudhuri, G. Das, M. Datar, R. Motwani, and V. Narasayya, “Overcoming limitations of sampling for aggregation queries”, in *Proceedings 17th International Conference on Data Engineering*, IEEE Comput. Soc, 2001, pp. 534–542.

[19] S. Chaudhuri, G. Das, and V. Narasayya, “Optimized Stratified Sampling for Approximate Query Processing”, *ACM Transactions on Database Systems*, vol. 32, no. 2, p. 9, Jun. 2007.

[20] S. Chaudhuri and U. Dayal, “An Overview of Data Warehousing and OLAP Technology”, *SIGMOD Record*, vol. 26, no. 1, pp. 65–74, 1997.

[21] S. Chaudhuri, R. Motwani, and V. Narasayya, “On random sampling over joins”, *ACM SIGMOD Record*, vol. 28, no. 2, pp. 263–274, 1999.

[22] B.-C. Chen, L. Chen, Y. Lin, and R. Ramakrishnan, “Prediction cubes”, in *Proceedings of the 31st International Conference on Very Large Data Bases*, 2005, pp. 982–993.

[23] Y. Chen, F. Dehne, T. Eavis, and A. Rau-Chaplin, “Building large ROLAP data cubes in parallel”, in *Proceedings. International Database Engineering and Applications Symposium, 2004. IDEAS ’04.*, Jul. 2004, pp. 367–377.

[24] J. W. Cooley and J. W. Tukey, “An Algorithm for the Machine Calculation of Complex Fourier Series”, *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965.

[25] I. Csiszár, “I-Divergence Geometry of Probability Distributions and Minimization Problems”, *The Annals of Probability*, vol. 3, no. 1, pp. 146–158, 1975.

[26] J. N. Darroch and D. Ratcliff, “Generalized Iterative Scaling for Log-Linear Models”, *The Annals of Mathematical Statistics*, vol. 43, no. 5, pp. 1470–1480, 1972.

[27] J. A. De Loera, R. Hemmecke, and M. Köppe, *Algebraic and Geometric Ideas in the Theory of Discrete Optimization*. SIAM, 2012.

[28] W. E. Deming and F. F. Stephan, "On a least squares adjustment of a sampled frequency table when the expected marginal totals are known", *The Annals of Mathematical Statistics*, vol. 11, no. 4, pp. 427–444, 1940.

[29] J. Dittrich, L. Blunschi, and M. A. V. Salles, "Dwarfs in the rearview mirror: how big are they really?", *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1586–1597, Aug. 1, 2008.

[30] M. Domingues, R. Rocha Silva, and J. Bernardino, "3iCubing: An Interval Inverted Index Approach to Data Cubes", *IEEE Access*, vol. 10, pp. 8449–8461, 2022.

[31] N. Y. C. D. of Finance. "Parking Violations Issued - Fiscal Year 2021". (2021), [Online]. Available: https://data.cityofnewyork.us/City-Government/Parking-Violations-Issued-Fiscal-Year-2021/kvfd-bves.

[32] B. J. Fino and V. R. Algazi, "A unified treatment of discrete fast unitary transforms", *SIAM Journal on Computing*, vol. 6, no. 4, pp. 700–717, 1977.

[33] B. J. Fino and V. R. Algazi, "Unified matrix treatment of the fast Walsh-Hadamard transform", *IEEE Transactions on Computers*, vol. 25, no. 11, pp. 1142–1146, 1976.

[34] R. Fontana and P. Semeraro, "Representation of multivariate Bernoulli distributions with a given set of specified moments", *Journal of Multivariate Analysis*, vol. 168, pp. 290–303, 2018.

[35] M. Fréchet, "Généralisation du théorème des probabilités totales", *Fundamenta Mathematicae*, vol. 25, no. 1, pp. 379–387, 1935.

[36] V. Ganti, M.-L. Lee, and R. Ramakrishnan, "Icicles: Self-tuning samples for approximate query answering", in *VLDB*, vol. 176, 2000.

[37] S. I. Gass, *Linear Programming: Methods and Applications*. Courier Corporation, 2003.

[38] B. Gavish and S. C. Graves, "The Travelling Salesman Problem and Related Problems", Massachusetts Institute of Technology, Operations Research Center, Working Paper, Jul. 1978. [Online]. Available: https://dspace.mit.edu/handle/1721.1/5363.

[39] J. Gray, S. Chaudhuri, A. Bosworth, *et al.*, "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals", *Data mining and knowledge discovery*, vol. 1, no. 1, pp. 29–53, 1997.

[40] A. Gupta, D. Agarwal, D. Tan, *et al.*, "Amazon Redshift and the Case for Simpler Data Warehouses", in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*, 2015, pp. 1917–1923.

[41] H. Gupta and I. S. Mumick, "Selection of Views to Materialize in a Data Warehouse", *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 1, pp. 24–43, 2005.

[42] P. J. Haas and J. M. Hellerstein, "Ripple Joins for Online Aggregation", *ACM SIGMOD Record*, vol. 28, no. 2, pp. 287–298, 1999.

[43] A. Y. Halevy, “Answering queries using views: A survey”, *The VLDB Journal*, vol. 10, no. 4, pp. 270–294, Dec. 1, 2001.

[44] J. Han, J. Pei, G. Dong, and K. Wang, “Efficient Computation of Iceberg Cubes with Complex Measures”, in *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD ’01)*, 2001, pp. 1–12.

[45] V. Harinarayan, A. Rajaraman, and J. D. Ullman, “Implementing Data Cubes Efficiently”, in *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD ’96)*, 1996, pp. 205–216.

[46] M. Held and R. M. Karp, “The Traveling-Salesman Problem and Minimum Spanning Trees”, *Operations Research*, vol. 18, no. 6, pp. 1138–1162, 1970. JSTOR: 169411.

[47] M. Held and R. M. Karp, “The traveling-salesman problem and minimum spanning trees: Part II”, *Mathematical Programming*, vol. 1, no. 1, pp. 6–25, Dec. 1, 1971.

[48] J. M. Hellerstein, P. J. Haas, and H. J. Wang, “Online Aggregation”, in *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data (SIGMOD ’97)*, ser. SIGMOD ’97, ACM, Jun. 1, 1997, pp. 171–182.

[49] K. Hoffman, *Linear Algebra*. Englewood Cliffs, NJ, Prentice-Hall, 1971.

[50] X. Hu, M. Qiao, and Y. Tao, “Independent range sampling”, in *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 2014, pp. 246–255.

[51] S. Ibrahim and X. Fu, “Recovering Joint Probability of Discrete Random Variables From Pairwise Marginals”, *IEEE Transactions on Signal Processing*, vol. 69, pp. 4116–4131, 2021.

[52] C. T. Ireland and S. Kullback, “Contingency Tables with Given Marginals”, *Biometrika*, vol. 55, no. 1, pp. 179–188, 1968.

[53] T. E. Jaynes, “Information Theory and Statistical Mechanics”, *The Physical Review*, vol. 106, no. 4, pp. 620–630, 1957.

[54] C. Jermaine, S. Arumugam, A. Pol, and A. Dobra, “Scalable approximate query processing with the DBO engine”, *ACM Transactions on Database Systems*, vol. 33, no. 4, 23:1–23:54, 2008.

[55] C. Jermaine, A. Dobra, S. Arumugam, S. Joshi, and A. Pol, “A disk-based join with probabilistic guarantees”, in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, 2005, pp. 563–574.

[56] C. Jermaine, A. Dobra, S. Arumugam, S. Joshi, and A. Pol, “The sort-merge-shrink join”, *ACM Transactions on Database Systems (TODS)*, vol. 31, no. 4, pp. 1382–1416, 2006.

[57] R. Jiroušek, “Solution of the Marginal Problem and Decomposable Distributions”, *Kybernetika*, vol. 27, no. 5, pp. 403–412, 1991.

[58] R. Jiroušek and S. Přeučil, “On the Effective Implementation of the Iterative Proportional Fitting Procedure”, *Computational Statistics & Data Analysis*, vol. 19, no. 2, pp. 177–189, Feb. 1995.

[59] M. Kahng, D. Fang, and D. H. (Chau, “Visual exploration of machine learning results using data cube analysis”, in *Proceedings of the Workshop on Human-In-the-Loop Data Analytics (HILDA ’16)*, 2016, pp. 1–6.

[60] N. Kamat, P. Jayachandran, K. Tunga, and A. Nandi, “Distributed and interactive cube exploration”, in *IEEE 30th International Conference on Data Engineering (ICDE ’14)*, 2014, pp. 472–483.

[61] N. Kargas, N. D. Sidiropoulos, and X. Fu, “Tensors, Learning, and “Kolmogorov Extension” for Finite-alphabet Random Vectors”, *IEEE Transactions on Signal Processing*, vol. 66, no. 18, pp. 4854–4868, Sep. 2018.

[62] R. Kimball and J. Caserta, *The Data Warehouse ETL Toolkit: Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data*. John Wiley & Sons, Apr. 2011.

[63] D. Koller and N. Friedman, *Probabilistic Graphical Models: Principles and Techniques* (Adaptive Computation and Machine Learning). Cambridge, MA: MIT Press, 2009.

[64] J. B. Kruskal, “On the shortest spanning subtree of a graph and the traveling salesman problem”, *Proceedings of the American Mathematical Society*, vol. 7, no. 1, pp. 48–50, 1956.

[65] H. Ku and S. Kullback, “Approximating discrete probability distributions”, *IEEE Transactions on Information Theory*, vol. 15, no. 4, pp. 444–447, Jul. 1969.

[66] L. V. S. Lakshmanan, J. Pei, and J. Han, “Quotient Cube: How to Summarize the Semantics of a Data Cube”, in *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB ’02)*, 2002, pp. 778–789.

[67] A. Lamb, M. Fuller, R. Varadarajan, *et al.*, “The Vertica Analytic Database: C-Store 7 Years Later”, *Proceedings of the VLDB Endowment 2012*, vol. 5, no. 12, pp. 1790–1801, Aug. 2012.

[68] D. C. Lay, “Linear Algebra and its Applications 4th edition”,

[69] F. Leng, Y. Bao, G. Yu, D. Wang, and Y. Liu, “An Efficient Indexing Technique for Computing High Dimensional Data Cubes”, in *Proceedings of the 7th International Conference on Advances in Web-Age Information Management (WAIM ’06)*, 2006, pp. 557–568.

[70] A. Y. Levy, A. O. Mendelzon, and Y. Sagiv, “Answering Queries Using Views”, in *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS ’95)*, 1995, pp. 95–104.

[71] F. Li, B. Wu, K. Yi, and Z. Zhao, “Wander join: Online aggregation via random walks”, in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 615–629.

[72] X. Li, J. Han, and H. Gonzalez, “High-Dimensional OLAP: A Minimal Cubing Approach”, in *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB ’04)*, 2004, pp. 528–539.

[73] N. Lomax and P. Norman, “Estimating Population Attribute Values in a Table: “Get Me Started in” Iterative Proportional Fitting”, *The Professional Geographer*, vol. 68, no. 3, pp. 451–461, Jul. 2016.

[74] G. Luo, C. J. Ellmann, P. J. Haas, and J. F. Naughton, “A scalable hash ripple join algorithm”, in *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, 2002, pp. 252–262.

[75] F. Malvestuto, “Computing the maximum-entropy extension of given discrete probability distributions”, *Computational Statistics & Data Analysis*, vol. 8, no. 3, pp. 299–311, 1989.

[76] R. McKenna, D. Sheldon, and G. Miklau, “Graphical-model based estimation and inference for differential privacy”, in *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, ser. Proceedings of Machine Learning Research, vol. 97, PMLR, 2019, pp. 4435–4444.

[77] J. J. Montes. “CubesViewer - Data exploration and visualization”. (), [Online]. Available: http://www.cubesviewer.com/ (visited on 05/08/2023).

[78] K. Morfonios and Y. E. Ioannidis, “CURE for Cubes: Cubing Using a ROLAP Engine”, in *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB ’06)*, 2006, pp. 379–390.

[79] K. Morfonios, S. Konakas, Y. Ioannidis, and N. Kotsis, “ROLAP implementations of the data cube”, *ACM Computing Surveys*, vol. 39, no. 4, p. 12, Nov. 2, 2007.

[80] R. O’Donnell, *Analysis of Boolean Functions*. Cambridge University Press, 2014.

[81] R. O’Donnell, “Analysis of boolean functions”, 2021. arXiv: 2105.10386.

[82] P. O’Neil, E. O’Neil, X. Chen, and S. Revilak, “The Star Schema Benchmark and Augmented Fact Table Indexing”, in *Performance Evaluation and Benchmarking*, R. Nambiar and M. Poess, Eds., ser. Lecture Notes in Computer Science, Springer, 2009, pp. 237–252.

[83] F. Olken, “Random sampling from databases”, Ph.D. dissertation, University of California at Berkley, 1993.

[84] N. Pansare, V. Borkar, C. Jermaine, and T. Condie, “Online aggregation for large mapreduce jobs”, *Proceedings of the VLDB Endowment*, vol. 4, no. 11, pp. 1135–1145, 2011.

[85] C. Qin and F. Rusu, “Parallel online aggregation in action”, in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, 2013, pp. 1–4.

[86] C. Qin and F. Rusu, “PF-OLA: a high-performance framework for parallel online aggregation”, *Distributed and Parallel Databases*, vol. 32, no. 3, pp. 337–375, Sep. 1, 2014.

[87] K. A. Ross and D. Srivastava, “Fast Computation of Sparse Datacubes”, in *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB ’97)*, 1997, pp. 116–125.

[88] E. Rozenberg, *Star Schema Benchmark data set generator (ssb-dbgen)*, 2020. [Online]. Available: https://github.com/eyalroz/ssb-dbgen.

[89] I. Savnik, “Index Data Structure for Fast Subset and Superset Queries”, in *Availability, Reliability, and Security in Information Systems and HCI*, Springer, 2013, pp. 134–148.

[90] J. Shanmugasundaram, U. M. Fayyad, and P. S. Bradley, “Compressed Data Cubes for OLAP Aggregate Query Approximation on Continuous Dimensions”, in *Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD ’99)*, 1999, pp. 223–232.

[91] A. Shukla, P. Deshpande, and J. F. Naughton, “Materialized View Selection for Multidimensional Datasets”, in *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB ’98)*, 1998, pp. 488–499.

[92] L. Sidirourgos, M. L. Kersten, and P. A. Boncz, “SciBORQ: Scientific data management with Bounds On Runtime and Quality.”, in *CIDR*, vol. 11, 2011, pp. 296–301.

[93] R. R. Silva, C. M. Hirata, and J. d. C. Lima, “Big high-dimension data cube designs for hybrid memory systems”, *Knowledge and Information Systems*, vol. 62, no. 12, pp. 4717–4746, 2020.

[94] R. R. Silva, J. d. C. Lima, and C. M. Hirata, “qCube: Efficient integration of range query operators over a high dimension data cube”, *Journal of Information and Data Management*, vol. 4, no. 3, pp. 469–469, Sep. 13, 2013.

[95] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis, “Dwarf: shrinking the PetaCube”, in *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD ’02)*, 2002, pp. 464–475.

[96] Y. Sismanis and N. Roussopoulos, “The complexity of fully materialized coalesced cubes.”, *VLDB*, vol. 4, pp. 540–551, 2004.

[97] D. Srivastava, S. Dar, H. V. Jagadish, and A. Y. Levy, “Answering Queries with Aggregation Using Views”, in *Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB ’96)*, 1996, pp. 318–329.

[98] Y. W. Teh and M. Welling, “On Improving the Efficiency of the Iterative Proportional Fitting Procedure”, in *International Workshop on Artificial Intelligence and Statistics*, PMLR, Jan. 2003, pp. 262–269.

[99] J. L. Teugels, “Some representations of the multivariate Bernoulli and binomial distributions”, *Journal of Multivariate Analysis*, vol. 32, no. 2, pp. 256–268, 1990.

[100] J. S. Vitter, M. Wang, and B. R. Iyer, “Data Cube Approximation and Histograms via Wavelets”, in *Proceedings of the 1998 ACM CIKM International Conference on Information and Knowledge Management (CIKM ’98)*, 1998, pp. 96–104.

[101] M. J. Wainwright and M. I. Jordan, “Graphical Models, Exponential Families, and Variational Inference”, *Foundations and Trends® in Machine Learning*, vol. 1, no. 1–2, pp. 1–305, 2007.

[102] L. Wang, R. Christensen, F. Li, and K. Yi, “Spatial online sampling and aggregation”, *Proceedings of the VLDB Endowment*, vol. 9, no. 3, pp. 84–95, Nov. 1, 2015.

[103] W. Wang, H. Lu, J. Feng, and J. X. Yu, “Condensed Cube: An Efficient Approach to Reducing Data Cube Size”, in *Proceedings of the 18th International Conference on Data Engineering (ICDE ’02)*, 2002, pp. 155–165.

[104] H. S. Warren, *Hacker’s Delight*. Pearson Education, 2013.

[105] S. Wu, S. Jiang, B. C. Ooi, and K.-L. Tan, “Distributed online aggregations”, *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 443–454, 2009.

[106] S. Wu, B. C. Ooi, and K.-L. Tan, “Continuous sampling for online aggregation over multiple queries”, in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, 2010, pp. 651–662.

[107] D. Xin, J. Han, X. Li, and B. W. Wah, “Star-Cubing: Computing Iceberg Cubes by Top-Down and Bottom-Up Integration”, in *Proceedings of 29th International Conference on Very Large Data Bases (VLDB ’03)*, 2003, pp. 476–487.

[108] K. Zeng, S. Agarwal, A. Dave, M. Armbrust, and I. Stoica, “G-ola: Generalized on-line aggregation for interactive analysis on big data”, in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 913–918.

[109] H. Zhang and F. Ding, “On the Kronecker products and their applications”, *Journal of Applied Mathematics*, vol. 2013, 2013.

[110] Y. Zhao, P. Deshpande, and J. F. Naughton, “An Array-Based Algorithm for Simultaneous Multidimensional Aggregates”, in *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data (SIGMOD ’97)*, 1997, pp. 159–170.

SACHIN BASIL JOHN

Office BC 214, EPFL/IC/IINFCOM/DATA
sachin.basiljohn@epfl.ch
+41 76 640 86 15
linkedin.com/in/sachinbjohn

EXPERIENCE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE, SWITZERLAND

09/2017 – present	Doctoral Student, Data Analysis Theory and Application Lab
09/2016 – 08/2017	Scientist, Data Analysis Theory and Application Lab
09/2015 – 02/2016	Student Research Assistant, Data Analysis Theory and Application Lab

RESEARCH AND TECHNOLOGY CENTER, SIEMENS AG, MUNICH, GERMANY

07/2015 – 09/2015	Software Intern

UMIC RESEARCH CENTRE, RWTH AACHEN UNIVERSITY, GERMANY

05/2013 – 07/2013	Research Intern

TATA INSTITUTE OF FUNDAMENTAL RESEARCH, MUMBAI, INDIA

05/2012 – 07/2012	Research Intern

EDUCATION

2017 – present	ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE, SWITZERLAND Ph.D. in Computer Science	**GPA 5.74/ 6**
2014 – 2016	ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE, SWITZERLAND M.S. in Computer Science	**GPA 5.70/ 6**
2010 – 2014	INDIAN INSTITUTE OF TECHNOLOGY, PATNA B.Tech. in Computer Science	**GPA 9.79/10**

PUBLICATIONS

- **Sachin Basil John**, Zhekai Jiang, Peter Lindner and Christoph Koch. 2023. **Aggregation and Exploration of High-Dimensional Data Using the Sudokube Data Cube Engine.** In *Companion of the 2023 International Conference on Management of Data* (SIGMOD 2023).
  DOI: https://doi.org/10.1145/3555041.3589729
- **Sachin Basil John** and Christoph Koch. 2022. **High-dimensional Data Cubes**. In *Proceedings of 48th International Conference on Very Large Databases* (VLDB 2022).
  DOI: https://doi.org/10.14778/3565838.3565839
- Mohammad Dashti, **Sachin Basil John**, Amir Shaikhha and Christoph Koch. 2017. **Transaction Repair for Multi-Version Concurrency Control**. In *Proceedings of the 2017 ACM International Conference on Management of Data* (SIGMOD 2017).
  DOI: https://doi.org/10.1145/3035918.3035919
- Mohammad Dashti, **Sachin Basil John**, Thierry Coppey, Amir Shaikhha, Vojin Jovanovic and Christoph Koch. **Compiling Database Application Programs**. preprint arXiv:1807.09887 (2018).
  DOI: https://doi.org/10.48550/arXiv.1807.09887

HONORS AND AWARDS

- ACM SIGMOD 2018 Comprehensive Reproducibility Award for paper titled "Transaction Repair for Multi-Version Concurrency Control" published in SIGMOD 2017.
- EPFL Graduate Fellowship (2017) in recognition of academic excellence.
- President of India Gold medal for securing highest GPA among all undergraduate courses in the 2010-2014 batch of Indian Institute of Technology, Patna.
- Best B.Tech. project in Computer Science department in 2010-2014 batch of Indian Institute of Technology, Patna.
- All India Rank 42 in Graduate Aptitude Test in Engineering (GATE) 2014 in Computer Science Paper.
- DAAD scholarship for research project in May -July 2013.
- Scholarship from the Indian Academy of Science for summer research project at Tata Institute of Fundamental Research, Mumbai in 2012.
- Director's congratulation letter for securing perfect Semester Point Index (10/10) in three semesters at Indian Institute of Technology, Patna.
- All India Rank of 4173 in IITJEE 2010 among 4,50,000 students.
- Rank 29 in the Kerala State Engineering Entrance Examination 2010.
- Best Outgoing Student in Grade 12.

PROJECTS

Sudokube, Data Analysis Theory and Applications Lab, EPFL
Under supervision of Prof. Christoph Koch.
System to support high-dimensional data cubes at interactive query speeds and moderate storage cost through judicious partial materialization of binary data cubes and quick reconstruction of missing information using statistical or linear programming techniques.

Θ-DB, Data Analysis Theory and Applications Lab, EPFL
Under supervision of Prof. Christoph Koch.
A database system that implements θ-joins with inequality predicates efficiently by identifying rewrite rules for pushing aggregations past the join, resulting in the pre-computation of partial aggregates based on the join predicate before performing the join operation.

Beta, Data Analysis Theory and Applications Lab, EPFL
Under supervision of Prof. Christoph Koch and Dr. Mohammad Dashti.
Transaction Repair for Multi-Version Concurrency Control - a new optimistic concurrency control algorithm that reuses the computations performed before a conflict to increase transaction throughput.
Compiling Transaction Programs - applying domain specific optimizations during compilation to increase performance of database application programs.

Efficient Distributed Causal Memory, Distributed Computing Lab , EPFL
Under supervision of Prof. Rachid Guerraoui
Develop an algorithm for better throughput of causally consistent systems.

Parallel Software Development, Summer Internship, Siemens AG, Munich.
Parallelize tasks on an embedded multicore system using a parallel library developed at Siemens.

Android Ecosystem Analysis, Dependable Systems lab, EPFL
Gather statistics on how apps use the system API provided by Android by decompiling and analyzing around 88,000 apps.

Transactional Key-Value Store over YARN, Big Data Project, EPFL
Design a platform to evaluate concurrency control algorithms on a classical distributed NoSQL system, a key-value store.

Design of Improved Algorithms for STMs, IIT Patna
Under supervision of Prof. Sathya Peri
Analyze the two categories of concurrency algorithms (Single Conflict Abort, Serialization Graph Testing) used in Software Transactional Memory (STMs) and design an algorithm that combines the good properties of the both while still keeping it efficient.

Design, Fabrication and Programming of Unmanned Ground Vehicle, Mobile Robotics, IIT Patna
Build an unmanned ground vehicle with differential drive that can navigate on its own to a given destination using GPS when a map of the environment is already known.

Co-Processor Design for Automatic Speech Recognition, UMIC Lab, RWTH Aachen
Under supervision of Prof. Anupam Chattopadhyay
Extend an existing architectural design written in LISA to suit speech recognition in Sphinx system and implement feature extraction efficiently using layered Course Grained Reconfigurable Architecture (CGRA) design.

Static Analysis for Malware Detection, Tata Institute of Fundamental Research, Mumbai
Under supervision of Prof. R.K. Shyamasundar
Semantic signature extraction using static analysis of source code replacing ineffective traditional syntactic signatures to model malware behavior and enhance detection capabilities.