High-dimensional Data Cubes

Sachin Basil John
École Polytechnique Fédérale de Lausanne (EPFL)
Data Analysis Theory and Applications Laboratory
sachin.basiljohn@epfl.ch

Christoph Koch
École Polytechnique Fédérale de Lausanne (EPFL)
Data Analysis Theory and Applications Laboratory
christoph.koch@epfl.ch

ABSTRACT

This paper introduces an approach to supporting high-dimensional data cubes at interactive query speeds and moderate storage cost. The approach is based on binary(-domain) data cubes that are judiciously partially materialized; the missing information can be quickly reconstructed using statistical or linear programming techniques. This enables new applications such as exploratory data analysis for feature engineering and other fields of data science. Moreover, it removes the need to compromise when building a data cube – all columns that we might ever wish to use can be included as dimensions. Our approach also speeds up certain dice, roll-up, and drill-down operations on data cubes with hierarchical dimensions compared to traditional data cubes.

Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/epfldata/sudokube/blob/main/experiments/vldb2022_sudokube_reproducibility.zip.

1 INTRODUCTION

Data cubes [13] are a principled and successful way of employing pre-computation and view materialization [1] to support interactive-speed query processing [16, 23] and exploratory data analysis [18] on multidimensional data. Via their limited and often visual query languages (using operations such as roll-up, drill-down, and slice), they are accessible to users not trained to write complex code.

The number of dimensions supported by a data cube is often limited in practice due to the aggressive materialization of its cuboids [28]. However, if the dimensionality were not practically limited, we could have high-dimensional data cubes that would be beneficial in the following scenarios.

- **Natively high-dimensional data**: Data scientists routinely work with high-dimensional data. For tasks such as feature engineering, interactive tools for exploratory data analysis built on top of data cubes would be extremely useful [22].
- **Inlining dimension tables**: Unlike star/snowflake schemas [13] with additional dimension tables that need to be joined with the central fact table, one can use high-dimensional data cubes to perform aggregations over hierarchical dimensions without resorting to joins. Instead of what is traditionally represented by a single dimension, we can have every dimension attribute as separate dimensions at different granularity; for instance, one can add month, quarter, and year as dimensions instead of a single time dimension.
- **Aggregations on partial keys**: Breaking up what is traditionally a single dimension key into several dimensions allows aggregations on partial keys to be implemented as roll-up operations on a high-dimensional data cube.

This paper explores how such high-dimensional data cubes can be made a reality. We address the infeasibility of materializing the entire cuboid lattice of a high-dimensional data cube by working with principled schemes that *materialize only a subset of the cuboids* [3, 15, 16, 40] that traditionally form a data cube. While this goes against the original vision of data cubes (which aggressively do as much pre-computation as possible), the disadvantages of missing some cuboids are addressed by our technical contributions.

We show two algorithmic approaches by which available cuboids can be used to efficiently *approximate or reconstruct* missing ones that we chose not to materialize. In the first technique, we convert available cuboids into linear equation systems that constrain the query cuboid and which we solve by linear programming [12]. In general, the problem remains underconstrained, and this yields tight upper and lower bounds on the possible values in the query cuboid. In the second technique, we extract *moments* [35] that characterize the underlying data distribution from the available cuboids. The moments extracted from the projections of a query cuboid capture its lower-order statistics. We extrapolate the remaining moments from the available ones, allowing us to approximate the values in the query cuboid in expectation. This approximation works well when many lower-order moments are available and higher-order deviations are rare in the data. Both techniques, particularly the second one, can be done much faster than reading the smallest available cuboid that subsumes the query and projecting it down to the dimensions requested in the query.

We propose *binary data cubes* as an internal representation for our system. In these data cubes, the domain of each dimension consists of only two values. Real data does not comprise only such dimensions, but our model does not restrict generality. We support unrestricted classical dimensions with domains of m values using *cosmetic dimensions* obtained by grouping $\lceil \log_2 m \rceil$ binary dimensions. Dimension hierarchies are encoded within the binary dimensions of a cosmetic dimension by assigning a hierarchy among them, thereby eliminating the need for star and snowflake schemas.

Based on these principles, we built a data cube engine, Sudokube. Depending on the solving algorithm, it uses either *linear programming* or *moment extrapolation* to quickly answer queries over binary data cubes that may have hundreds or even thousands of dimensions. Furthermore, the system can approximate the query using an online [17] algorithm, starting with the smallest available cuboids, incrementally reading in larger and larger cuboids (in the worst case up to the smallest cuboid that subsumes the query), turning them into linear equations or moments, and continuously maintaining approximate query results that improve until an exact answer is achieved. This practical approach allows for interactive data analysis and exploration of very high-dimensional datasets.

We validate our claims using two datasets: (1) New York City Parking Violations Issued [33], a natively moderately high-dimensional dataset (43 columns / 429 binary dimensions) that cannot be currently explored and analyzed using existing full-materialization data cube technology, and (2) Star Schema Benchmark [34], a standard benchmark for decision support queries on a star schema. Our experiments show that Sudokube yields results with less than 1% average error in under a second for queries on both datasets. We also show how our approach fares with inlining dimension tables by turning columns into hierarchies of binary dimensions.

Example 1.1. On the New York City dataset, the query[1]

SELECT floor(issue_date_year/2), registration_state, SUM(1)
FROM NYC_Cube
GROUP BY floor(issue_date_year/2), registration_state

asks for the number of parking violations grouped by pairs of consecutive years and vehicle registration states. In Sudokube, this does not require a join with a dimension table or complex re-computation; the issue_date_year column is already internally represented by a sequence of binary dimensions, and this query can be expressed by grouping by all but the least significant of the column's binary dimensions. Generally speaking, by aligning them with binary prefix representations, we can represent wide ranges of dimension hierarchies, including non-numerical ones.

The paper is structured as follows. Section 2 discusses some challenges of high-dimensional data cubes. Section 3 introduces and formalizes our framework of binary data cubes and discusses how to work with them. Section 4 explores taking a set of cuboids of the data cube as a linear equation system that describes another cuboid (such as a query), and presents several properties that make this viewpoint useful; in particular, we show how easy it is to find a maximal linearly independent set of equations. Section 5 explores an alternative strategy where we introduce the concept of moments and explain how we extract moments from projections, extrapolate unknown moments and finally obtain an approximation of a cuboid from these extrapolated moments. Section 6 describes our system, Sudokube. Section 7 discusses related work. We experimentally evaluate the performance and the feasibility of our approach in Section 8. We conclude and discuss future work in Section 9.

2 THE INFEASIBILITY OF HIGH-DIMENSIONAL DATA CUBES

The number of dimensions d that can be supported in practice in state-of-the-art data cube technology is limited as the storage costs quickly become astronomical as d increases. In this section, we explain why this is the case. Consider some relational data stored in a *fact table* consisting of d *dimension columns* and one or more numerical *fact columns* whose values are to be aggregated, grouped by the dimension columns. For simplicity of notation, we will assume there to be just one fact column throughout the remainder of the paper, and we will focus on aggregation by summation. The dimension columns hold values, subsequently called *keys*, that are keys in so-called *dimension tables*. The dimension tables describe a further coarsening hierarchy for the dimensions. For instance, the fact table may register keys in a time dimension representing timestamps at the granularity of seconds, and the corresponding dimension table(s) map(s) these timestamps to hours, days, months, and years. Depending on the complexity of these additional tables and their join paths, one speaks of star or snowflake schemas [13].

[1]This is the first of the concrete queries experimented with in Figure 12.

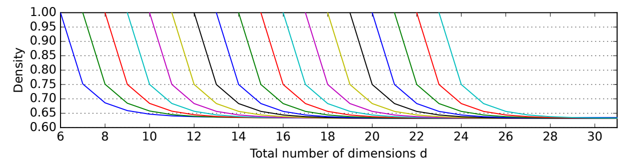


Figure 1: Simulation results for the density of a random d_0-dimensional projection of a d-dimensional cuboid. One curve for each d_0 value between 6 and 23.

Example 2.1. We now begin a running example that we will use throughout this paper. The fact table storing a company's sales data for four Swiss cities (Geneva G, Lausanne L, Zurich Z, and Bern B) and the four quarters of 2021 is shown in Figure 2a. Additional dimension tables for city and quarter could classify cities into French or German-speaking and map quarters to half-years, but are omitted for brevity.

Data cubes are built by defining views that aggregate the fact column of the fact table grouped by the dimension columns and all their subsets and materializing these views. We refer to these views as *cuboids*, following the intuitions of multidimensional array representations [1] where all the views are hypercubes. 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 d-dimensional data cube, for each $0 \leq k \leq d$, there are $\binom{d}{k}$ many k-dimensional projections. For instance, for $d = 3$, the base cuboid is three-dimensional, and its projections are three two-dimensional, three one-dimensional, and one zero-dimensional cuboid.

For the simplicity of analysis, assume the same number m of distinct keys in each dimension. A multidimensional array or *dense* representation of the base cuboid with m^d entries is out of the question even for moderate values of m. It would require excessive space, yet most of its entries would likely be zero. On the other hand, a *sparse* representation of the cuboid stores tuples for each non-zero array entry in relational format. While they efficiently handle the sparsity that comes with high-dimensional data, the storage requirements of the sparse (representations of the) projection cuboids depend significantly on data distributions.

Let n be the number of tuples in the sparse representation of the base cuboid and let $d_0 = \log_m n$. Consider a scenario where $n = m^{d_0}$ entries are assigned to (uniformly) random distinct places in a d-dimensional base cuboid. We ask how big the sparse representation of a random d_0-dimensional projection of such a base cuboid would be. We performed an extensive simulation with $m = 2$ and approximated the expected value for p/n, where p is the size of the sparse representation of the projection. Clearly for $d_0 = d$, this ratio is 1. It turns out that as $d - d_0$ increases, this value extremely quickly converges to a value in the vicinity of 0.63 (see Figure 1). That is, in the lattice of projections of the base cuboid, all the cuboids of at least d_0

Q	City	Sales
Q1	L	1
Q1	Z	1
Q1	B	1
Q2	Z	2
Q3	G	3
Q3	Z	2
Q4	G	4
Q4	L	2
Q4	Z	1

(a) Fact table

	3	2	1	0	Sales
1	0	0	0	1	1
2	0	0	1	0	1
3	0	0	1	1	1
4	0	1	1	0	2
5	1	0	0	0	3
6	1	0	1	0	2
7	1	1	0	0	4
8	1	1	0	1	2
9	1	1	1	0	1

(b) Sparse representation of the base cuboid of the binary data cube

Figure 2: Example sales data cube

dimensions have a size of at least $\approx 0.63 * n$. If, for instance, $d > 2d_0$, then there are more than 2^{d-1} such cuboids, and the overall data cube takes (far) more space than $2^{d-2}n$. 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 dimension columns) for it to fare much better than the random case. So fully materialized data cubes, whether sparse or dense, with large d really cannot be built.

3 PARTIALLY MATERIALIZED HIGH-DIMENSIONAL BINARY DATA CUBES

Our system uses binary data cubes where the key in every dimension is either 0 or 1. This does not lead to any loss in functionality or expressive power — any dimension with m values can be encoded as a cosmetic dimension using $\lceil \log_2 m \rceil$ binary dimensions. Formally, a *d-dimensional binary cuboid* C is a map $(I \to \{0,1\}) \to K$ where I is a d-element set of binary dimensions and K is a field (such as $\mathbb{R}$), representing facts. In other words, C maps 2^d *cells* (in a d-dimensional array) to elements of K.

Example 3.1. We encode the fact table in Figure 2a into a 4-D base cuboid shown in Figure 2b as follows. The two least significant bits 1,0 (we use big-endian order) encode the cities with $G \mapsto 00, L \mapsto 01, Z \mapsto 10, B \mapsto 11$ and bits 3,2 encode the quarter with $\mathrm{Q}i \mapsto i-1$. The two pairs of binary dimensions are also hierarchical: dimension 1 selects among French (Geneva and Lausanne) and German-speaking cities (Zurich and Bern); dimension 3 selects between the first and second half-year of 2021.

3.1 Projection and Partial Materialization

Given a cuboid C with dimensions I, a *projection* $\Pi_J(C)$ of C is a cuboid with dimensions $J \subseteq I$ such that value of cell p is the sum of the values for all cells q where p is consistent with q on J, i.e.,

$$\Pi_J(C)(p) = \sum \left\{ C(q) \middle| \begin{array}{l} q : I \to \{0,1\}, \\ p(j) = q(j) \text{ for all } j \in J \end{array} \right\} \qquad (1)$$

Given C, $\Pi_J(C)$ is uniquely determined by J ("the dimensions to project to"), so we will also denote $\Pi_J(C)$ by C_J. So far, we have insisted on materializing all possible projections corresponding to the subset lattice of the dimensions of the data cube. We could instead choose to materialize only a subset of them; but to make the approach feasible, it would have to be a *very* small fraction [16].

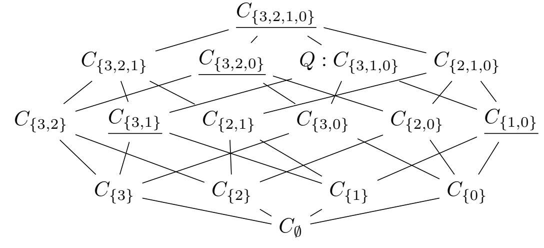


Figure 3: The lattice of cuboids of the 4-D sales data cube with materialized ones underlined

Example 3.2. The 4-D sales data cube has 16 cuboids, of which we choose to materialize three, $C_{\{3,1\}}$, $C_{\{1,0\}}$, and $C_{\{3,2,0\}}$, in addition to the base cuboid $C_{\{3,2,1,0\}}$ at data cube creation time. The lattice of cuboids is shown in Figure 3, with materialized cuboids underlined.

The number of possible cuboids in a high-dimensional data cube – the number of elements in the lattice – is astronomical. Given that only a very small fraction of these can be materialized, how could such data cubes support interactive querying? We assume that the number of dimensions requested in a query is small – producing a table to be displayed to a human user [28]. Let $Q \subsetneq I$ be one such query. The query result is then given by the cuboid C_Q, which may or may not be materialized. To answer the query Q, we can use the smallest subsuming cuboid C_S with $Q \subseteq S \subseteq I$ that has been materialized [3]. We call this the *naïve* algorithm. We will consider better and more interesting alternatives to this later in the paper. Assuming that the base cuboid is always among those materialized, there exists a materialized subsumer for every query, and every query can be answered. The challenge then becomes to decide, given a storage budget, which cuboids should be materialized so that a query workload can be answered quickly [15].

Example 3.3. Consider the query asking for sales grouped by city and half-year of 2021, whose result is cuboid $C_{\{3,1,0\}}$. This cuboid has not been materialized. The smallest materialized cuboid that subsumes it is $C_{\{3,2,1,0\}}$, the base cuboid. Under the naïve algorithm, we project it to dimensions $\{3,1,0\}$ to answer the query.

3.2 Advantages of Binary Data Cubes

Binary data cubes have the following strong points, especially when high-dimensional. Firstly, the mathematics, algorithms, and implementation are much cleaner. Some of the technical contributions in Sections 4 and 5 would necessarily suffer from rather awkward and bloated notation if we were to stick with unrestricted dimension domains, while the move to binary domains seems to expose numerous connections to relevant fields and solution approaches. Secondly, the finer granularity of dimensions in a binary data cube allows us to encode structure and hierarchy within the relationship among dimensions itself, eliminating the need for star or snowflake schemas plus joins. For example, compare a data cube with a single time dimension (of timestamps) with a binary cube in which time consists of separate groups of bit-sized dimensions for years, quarters (two bits), months (e.g., two bits for months per quarter), etc. If the query asks to break down sales by quarter, in the first case, we have no other choice than to fetch a cuboid with the time

dimension at its full granularity, join with the dimension hierarchy, and aggregate to obtain quarterly data; in the latter case, this cuboid may be available directly (see Example 3.1, where we introduced – very simple – hierarchical time and location dimensions).

4 SOLVING QUERIES USING LINEAR PROGRAMMING

The naïve algorithm described in the previous section can always answer any query $Q \subseteq I$, though not necessarily in interactive time. In the worst case, the query is answered by aggregating the base cuboid, which can take a long time depending on the size. This section explores an alternative approach to assembling the query cuboid C_Q from the cuboids it subsumes.

$C_{\{1,0\}}$		$0 \mapsto$ 0	$0 \mapsto$ 1
$1 \mapsto$	0	7	3
	1	6	1

$C_{\{3,1\}}$		$1 \mapsto$ 0	$1 \mapsto$ 1
$3 \mapsto$	0	1	4
	1	9	3

$C_{\{3,2,0\}}$, $3 \mapsto 0$		$0 \mapsto$ 0	$0 \mapsto$ 1
$2 \mapsto$	0	1	2
	1	2	0

$C_{\{3,2,0\}}$, $3 \mapsto 1$		$0 \mapsto$ 0	$0 \mapsto$ 1
$2 \mapsto$	0	5	0
	1	5	2

$C_{\{3,0\}}$		$0 \mapsto$ 0	$0 \mapsto$ 1
$3 \mapsto$	0	3	2
	1	10	2

Figure 4: Dense representations of the pre-materialized cuboids $C_{\{1,0\}}, C_{\{3,1\}}$, and $C_{\{3,2,0\}}$ of the sales data cube as well as of $C_{\{3,0\}}$ obtained by projecting $C_{\{3,2,0\}}$ at runtime.

Example 4.1. The materialized projections $C_{\{1,0\}}$, $C_{\{3,1\}}$, and $C_{\{3,2,0\}}$ of the sales data cube are shown in Figure 4. The cell $p = \{3 \mapsto 1, 1 \mapsto 0\}$ in $C_{\{3,1\}}$, for example, contains value 9 because the 5th, 7th, and 8th row of the base cuboid have mappings whose restrictions to dimensions $\{3, 1\}$ is $\{3 \mapsto 1, 1 \mapsto 0\}$, and the sum of the sales in these rows is 9. For the query $Q = \{3, 1, 0\}$, the query result C_Q subsumes the materialized cuboids $C_{\{1,0\}}$ and $C_{\{3,1\}}$. Furthermore, C_Q also subsumes the projection $C_{\{3,0\}}$ which can be obtained by projecting $C_{\{3,2,0\}}$ at runtime.

The result C_Q of a query Q contains $2^{|Q|}$ cells identified by a mapping $q : Q \to \{0, 1\}$. For simplicity, let the variable x_q denote the value $C_Q(q)$. Then, for some cuboid C_J such that $J \subseteq Q$, Equation 1 yields a system of $2^{|J|}$ linear equations which constrain the x_q values that make up C_Q. The equation e_p for cell p is given by

$$e_p : \qquad \sum \left\{ x_q \;\middle|\; \begin{array}{l} q : Q \to \{0,1\}, \\ p(j) = q(j) \text{ for all } j \in J \end{array} \right\} = C_J(p)$$

These equations define a vector space [19], a subspace of the $2^{|Q|}$-dimensional space spanned by the cells of C_Q. We will be interested in the dimensionality of its solution space (kernel); obviously, a zero-dimensional solution space means we can reconstruct C_Q precisely. When several cuboids are available to constrain C_Q, we can combine the linear equations of the individual cuboids. This system of equations is, in general, not linearly independent. To reduce solving time, we want to have a maximal set of linearly independent equations so as to capture all the constraints from the fetched cuboids using as few equations as possible. Before we explain how to obtain a maximal set of linearly independent equations, we introduce some notation. For some variable order $\prec$, let $\hat{x}_e$ denote the minimal variable that occurs in some equation e. Furthermore, Let $e_1 \equiv e_2$ be true for two equations e_1 and e_2 if $\hat{x}_{e_1} = \hat{x}_{e_2}$. Of course, $\equiv$ forms an equivalence relation.

		x_{000}	x_{001}	x_{010}	x_{011}	x_{100}	x_{101}	x_{110}	x_{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 5: 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. We pick the equations with circled ones to form a basis.

Example 4.2. Consider the query Q = $\{3, 1, 0\}$ and three of its projections $C_{\{1,0\}}, C_{\{3,1\}}$, and $C_{\{3,0\}}$. The system of linear equations defined by these projections is shown in Figure 5. For some bits $b_3, b_1, b_0 \in \{0, 1\}$, in order to concisely show cells like $p = \{\, 3 \mapsto b_3, 1 \mapsto b_1, 0 \mapsto b_0 \,\}$, we write them as ordered sequences $b_3b_1b_0$, making Q implicit. The 8 variables used are written as x_{000}, x_{001}, $\ldots, x_{111}$. Out of the 12 equations, at most 7 are linearly independent. Under the variable order $x_{000} \prec x_{001} \prec \cdots \prec x_{110} \prec x_{111}$, 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 $S_{111} = \emptyset$.

There is an easy way of finding a maximal subset of linearly independent equations – a basis [19]. We now state a theorem we use to construct a basis. Due to space constraints, we omit its proof here, but we include it in Appendix B.

Theorem 4.3. *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.*

Picking a set of linear equations in this way 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 there was no equation having 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.

Example 4.4. In Figure 5, we have marked the chosen witness of every equivalence class with a circle. According to Theorem 4.3, $\{e_{*00}, e_{*01}, e_{*10}, e_{*11}, e_{10*}, e_{11*}, e_{1*1}\}$ 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 $e_{*00}+e_{1*1}-e_{10*} = x_{000} + x_{111} = 0$. If we impose a non-negativity constraint on all the x values, this equation gives us $x_{000} = x_{111} = 0$, and so we obtain the query result $(x_{000}, x_{001}, \dots, x_{111}) = (0, 1, 3, 1, 7, 2, 3, 0)$.

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 upper and lower bounds of every variable x_q using linear programming [12]. Our experiments show that this approach quickly yields tight bounds on query results for low-dimensional queries but does not scale well with query dimensionality. For this reason, we do not go into further details of this approach and present a better one in the next section.

5 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 the cuboid, 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.

5.1 Moments of the Cuboid

We now describe the characterization of a d-dimensional cuboid with dimension set I in terms of its moments. Given a cuboid, we define the moment m_J for every subset $J \subseteq I$ as follows

$$\begin{aligned} m_J &:= \sum \left\{ x_q \mid q : I \to \{0,1\}, q(j) = 1 \text{ for all } j \in J \right\} \\ &= C_J(p), \text{ where } p(j) = 1 \text{ for all } j \in J \end{aligned}$$

In other words, for some $J \subseteq I$, the moment m_J of a cuboid C is the value of cell p in C_J that maps all dimensions in J to 1. A d-dimensional cuboid has 2^d moments, one for every subset of its dimension set. Also, note that for any set K such that $J \subseteq K \subseteq I$, m_J is one of the $2^{|K|}$ moments of C_K as well.

If the values of a cuboid are non-negative, which we will now assume throughout the rest of this paper, we can also have a probabilistic interpretation for these values. For every dimension $i \in Q$ of the cuboid, we have a corresponding Bernoulli random variable X_i. We interpret the values of a cuboid as unnormalized probabilities in the joint distribution of these random variables. The probabilities are obtained by rescaling the values using their total. Under this model, the projections of a cuboid are marginal distributions on subsets of random variables. Furthermore, we can express any moment m_J of the cuboid C_Q in terms of mixed moments [35] of random variables X_j for any $j \in J$ as

$$\frac{m_J}{m_\emptyset} = E\left[\prod_{j \in J} X_j\right] \tag{2}$$

Example 5.1. The joint probability distribution as well as the mixed moment of binary random variables X_3 and X_0 associated with the projection $C_{\{3,0\}}$ for $b_3, b_0 \in \{0,1\}$ are given by

$$P(X_3 = b_3, X_0 = b_0) = \frac{C_{\{3,0\}}(3 \mapsto b_3, 0 \mapsto b_0)}{\sum_p C_{\{3,0\}}(p)}, \text{ and}$$

$$E\left[X_3 \cdot X_0\right] = \sum_{b_3, b_0} \left(P(b_3, b_0) \cdot b_3 \cdot b_0\right) = \frac{m_{\{3,0\}}}{m_\emptyset} = \frac{2}{17}.$$

Under this probabilistic interpretation, our characterization of binary cuboids using their moments is essentially how multivariate Bernoulli distributions are characterized in [44] and [11]. Given some ordering of the dimensions, we order the subsets of dimensions and, thereby, the moments lexicographically. We define the moment vector $\mathbf{m}$ of a cuboid to have all of its moments arranged in this order. Recall that we also define a similar order for the values of the cuboid. We similarly define the vectorization $\mathbf{x}$ of the cuboid to contain all its values in that order. The following proposition [11, 44] describes the relationship between $\mathbf{m}$ and $\mathbf{x}$.

Proposition 5.2. *Given the moment vector* $\mathbf{m}$ *of a* d*-dimensional cuboid with values* $\mathbf{x}$ *and two matrices* $\mathbf{M}$ *and* $\mathbf{W}$ *as specified below, the following two statements are true where* $\otimes$ *denotes the Kronecker product (applied* d *times in the equations):*

$$(1)\ \mathbf{m} = \mathbf{M}^{\otimes d}\mathbf{x} \quad \text{and} \quad (2)\ \mathbf{x} = \mathbf{W}^{\otimes d}\mathbf{m},$$

where $\mathbf{M} = \left[\begin{smallmatrix} 1 & 1 \\ 0 & 1 \end{smallmatrix}\right]$ *and* $\mathbf{W} = \mathbf{M}^{-1} = \left[\begin{smallmatrix} 1 & -1 \\ 0 & 1 \end{smallmatrix}\right]$.

Example 5.3. The moments of the query cuboid C_Q for the query $Q = \{3,1,0\}$ with $\mathbf{x} = (x_{000} \dots x_{111})$ on the sales data cube are

$$\begin{bmatrix} m_\emptyset \\ m_{\{0\}} \\ m_{\{1\}} \\ m_{\{1,0\}} \\ m_{\{3\}} \\ m_{\{3,0\}} \\ m_{\{3,1\}} \\ m_{\{3,1,0\}} \end{bmatrix} = \begin{bmatrix} 1&1&1&1&1&1&1&1 \\ 0&1&0&1&0&1&0&1 \\ 0&0&1&1&0&0&1&1 \\ 0&0&0&1&0&0&0&1 \\ 0&0&0&0&1&1&1&1 \\ 0&0&0&0&0&1&0&1 \\ 0&0&0&0&0&0&1&1 \\ 0&0&0&0&0&0&0&1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 3 \\ 1 \\ 7 \\ 2 \\ 3 \\ 0 \end{bmatrix} = \begin{bmatrix} 17 \\ 4 \\ 7 \\ 1 \\ 12 \\ 2 \\ 3 \\ 0 \end{bmatrix}$$

5.2 Cuboid Approximation from Projections

Given a query Q, Sudokube approximates its result C_Q from the moments of the available projections. We make statistical assumptions about the underlying data distribution and extrapolate the missing moments of C_Q from the known ones. For example, we can assume that the values are uncorrelated beyond what is already known. This is not an unreasonable assumption, and while one can always find data that deviates significantly from this model, it becomes increasingly harder as more data is known.

Before we describe the formula for computing the missing moments, we define two additional terms and explain their relationships with the moments. The mixed central moment μ_J and the mixed moment product p_J for a set of dimensions J are defined as

$$\frac{\mu_J}{m_\emptyset} := E\left[\prod_{j \in J} \left(X_j - E\left[X_j\right]\right)\right] \text{ and } p_J := \prod_{j \in J} E\left[X_j\right] = \prod_{j \in J} \frac{m_{\{j\}}}{m_\emptyset}.$$

We now describe the moment m_J in terms of mixed central moments and moment products associated with all subsets of J. In the derivation of the formula shown below, we first split each X_j in the definition of the moment m_J into $(X_j - E[X_j])$ and $E[X_j]$ and then apply the multi-binomial expansion on the products. The next step is obtained from the linearity of expectation since every $E[X_\ell]$ is constant with respect to the expectation. Finally, we rewrite the expression in terms of μ_K and $p_{J\setminus K}$ using their definitions. We can also derive a similar formula to compute μ_J as an alternating sum of products of m_K and $p_{J\setminus K}$ for all sets $K \subseteq J$.

$$\begin{aligned} m_J &= m_\emptyset \cdot E\left[\prod_{j\in J}\left(\left(X_j - E\left[X_j\right]\right) + E\left[X_j\right]\right)\right] \\ &= m_\emptyset \cdot E\left[\sum_{K\subseteq J}\prod_{k\in K}\left(X_k - E\left[X_k\right]\right)\cdot\prod_{\ell\in J\setminus K}E\left[X_\ell\right]\right] \\ &= \sum_{K\subseteq J} m_\emptyset \cdot E\left[\prod_{k\in K}\left(X_k - E\left[X_k\right]\right)\right]\cdot\prod_{\ell\in J\setminus K}E\left[X_\ell\right] \\ &= \sum_{K\subseteq J}\mu_K \cdot p_{J\setminus K} \end{aligned} \tag{3}$$

Given some query Q and any set $J \subseteq Q$, let $\text{Known}(J)$ denote the set of all subsets $K \subseteq J$ for which the projection C_K is known. Clearly, $\text{Known}(J) \subseteq 2^J$ and since $J \subseteq Q$, $\text{Known}(J) \subseteq \text{Known}(Q)$ as well. Furthermore, since all projections can be computed from a cuboid, if $K \in \text{Known}(Q)$, then we also have $2^K \subseteq \text{Known}(Q)$.

We assume that all subsets of Q up to size 1 are in $\text{Known}(Q)$. The one-dimensional projections are few in number and small in size and they can all be cached in memory. With this assumption, p_J can always be computed for every $J \subseteq Q$. Additionally, for every $K \in \text{Known}(Q)$, we can compute m_K and μ_K from the cuboid C_K.

Next, we explain how we extrapolate the known moments to predict the missing moments of the query result C_Q. As described earlier in this section, we assume that the data is uncorrelated beyond what is known. We model this by setting all unknown mixed central moments μ_U to zero. The zero value of some mixed central moment indicates that the corresponding dimensions do not result in extreme deviations in values when changed together. While it is certainly possible for the data to have extreme higher-order deviations (e.g., when the most significant values are concentrated along the diagonals of the cuboid), this seems unlikely to happen in real-world data. Under this model, we extrapolate m_J as m'_J for any $J \subseteq Q$ as

$$m'_J := \sum_{K\in\text{Known}(J)} \mu_K \cdot p_{J\setminus K}. \tag{4}$$

The following proposition states the correctness of the extrapolation formula, which can be derived easily by setting all unknown mixed moments μ_U to zero in Equation 3.

Proposition 5.4. *Equation 4 describes the extrapolated moment m'_J for some set $J \subseteq Q$ satisfying the following conditions: (1) If $J \in \text{Known}(Q)$, then $m'_J = m_J$. (2) Otherwise, m'_J denotes the extrapolated moment under the uncorrelatedness assumption.*

Algorithm 1 shows an incremental algorithm for extrapolating the moments according to Equation 4. The initial state of the algorithm is when $\text{Known}(Q)$ contains all subsets up to size 1. Then the algorithm incrementally updates the unknown moments as the subsets of Q with sizes greater than one are added to $\text{Known}(Q)$ one after the other. We assume that before a set S is added to $\text{Known}(Q)$, all of its subsets were added in previous steps. The correctness of this algorithm is given by the following theorem, whose proof can be found in Appendix B.

Algorithm 1: Online algorithm to extrapolate unknown moments from known moments

initial input: Moments m_K for all $K \subseteq Q$ with $|K| \leq 1$
output: Extrapolated moments m'_J for all $J \subseteq Q$

```
[1] init:
[2]     foreach J ⊆ Q do
[3]         p_J ← ∏_{j∈J} m_{j} / m_∅
[4]         m'_J ← m_∅ · p_J
[5] upon receiving m_S:                 //Add S to Known(Q)
[6]     μ_S ← m_S − m'_S
[7]     foreach J ⊆ Q such that J ⊇ S do
[8]         m'_J ← m'_J + μ_S · p_{J\S}
```

Theorem 5.5. *Algorithm 1 computes the extrapolated moments m'_J of cuboid C_Q representing query Q according to Equation 4 as subsets S are added to $\text{Known}(Q)$ one after the other.*

The most expensive step of the extrapolation algorithm is updating the extrapolated moments of all supersets. However, this algorithm can be further optimized by using techniques similar to those in [6], but a full discussion is beyond the scope of this paper.

Example 5.6. Consider the query $Q = \{3, 1, 0\}$ on the sales data cube. Initially, $\text{Known}(Q) = \{\emptyset,\{0\},\{1\},\{3\}\}$. The initial values of the extrapolated moments $m'_{\{1,0\}}$ and $m'_{\{3,1,0\}}$ are given by

$$m'_{\{1,0\}} = \frac{m_{\{1\}}m_{\{0\}}}{m_\emptyset} = \frac{28}{17}, \text{ and } m'_{\{3,1,0\}} = \frac{m_{\{3\}}m_{\{1\}}m_{\{0\}}}{{m_\emptyset}^2} = \frac{336}{289}.$$

Once the cuboid $C_{\{1,0\}}$ is fetched, $m_{\{1,0\}}$ is computed and the set $\{1, 0\}$ is added to $\text{Known}(Q)$. The updated moments are

$$m'_{\{1,0\}} = 1, \text{ and } m'_{\{3,1,0\}} = \frac{336}{289} + \left(1 - \frac{28}{17}\right)\cdot\frac{12}{17} = \frac{12}{17}.$$

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

$$m'_{\{3,1,0\}} = \frac{336}{289} + \left(1 - \frac{28}{17}\right)\cdot\frac{12}{17} + \left(3 - \frac{84}{17}\right)\cdot\frac{4}{17} + \left(2 - \frac{48}{17}\right)\cdot\frac{7}{17} = \frac{-26}{289}.$$

Algorithm 2: In-place Fast Inverse Transform

input: Array A representing $\mathbf{m}$ of size $N = 2^d$
output: Same array A whose contents now represent $\mathbf{x}$

```
[1] s ← 1
[2] while s < N do
[3]     for i ← 0 until N step by 2 ∗ s do
[4]         for j ← i until i + s do
[5]             A[j + s] ← min(A[j], max(0, A[j + s]))
[6]             A[j] ← A[j] − A[j + s]
[7]     s ← s ∗ 2
```

From this point, we ignore the distinction between the real 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 $\mathbf{x} = \mathbf{W}^{\otimes d}\mathbf{m}$ from Proposition 5.2. However, computing the values of the cuboid naïvely using this equation is expensive. We describe a $O(|\mathbf{m}| \log |\mathbf{m}|)$ complexity algorithm for the computation of the cuboid values from the moments in Algorithm 2. Like the FFT algorithm [10], it uses the recursive property of the repeated Kronecker product to break a problem of size N into two $N/2$ sized problems. We perform some local perturbations during the inverse transform to improve the error. We need to do these because our assumption of zero value for the unknown mixed central moments is not always feasible. The values of the vector $\mathbf{m}$ must satisfy several constraints so that all values of the vector $\mathbf{x}$ are non-negative. These conditions enforce lower and upper bounds for the unknown moments based on lower-order ones, and our extrapolation technique does not always respect these bounds. By imposing local bounds on the extrapolated moments in Line 5, we minimize such occurrences and, thereby, the error.

Example 5.7. The following figure shows the computations in the fast inverse transform for the result C_Q of the query $Q = \{3, 1, 0\}$.

$m_\emptyset$	17	13	7	0	x_{000}
$m_{\{0\}}$	4	4	3	1	x_{001}
$m_{\{1\}}$	7	6	6	3	x_{010}
$m_{\{1,0\}}$	1	1	1	1	x_{011}
$m_{\{3\}}$	12	10	7	7	x_{100}
$m_{\{3,0\}}$	2	2	2	2	x_{101}
$m_{\{3,1\}}$	3	3	3	3	x_{110}
$m_{\{3,1,0\}}$	$\frac{-26}{289}$	0	0	0	x_{111}

6 SYSTEM ARCHITECTURE

The previous sections described different approaches to making interactive querying of high-dimensional data cubes feasible. We now describe the details of the Sudokube system where we implemented these approaches. As shown in Figure 6, Sudokube comprises three main components. First, the *backend* manages the materialized cuboids and performs projections of cuboids to fewer dimensions. Second, the *core query execution engine* decides which (possibly projected) cuboids to instruct the backend to provide. These are turned into equation systems or moments and then passed to one of the solvers to be processed in batch or online aggregation mode. Finally, the *frontend* provides schema support and gives the illusion of unrestricted dimension domains, making the internally used binary dimensions transparent to the user. It also offers basic data exploration support and a user interface and API. We now describe different aspects of the system in more detail.

6.1 Data Loading

The system design takes a fundamentalist stance regarding binary dimensions; the storage backend, the core query engine, and the solver only know binary-dimensional cuboids. Non-binary dimensions and dimension-table style information are considered a concern of the frontend. The system's frontend supports an abstraction called a *cosmetic dimension*. Such a dimension may have an arbitrary finite (but unbounded) domain D, thus an arbitrary number of alternative values; internally, such a dimension is represented by (at least) $k = \lceil \log_2 |D| \rceil$ binary dimensions. A key in such a cosmetic dimension is a k-bit number. Where needed, an explicit NULL value is made part of D.

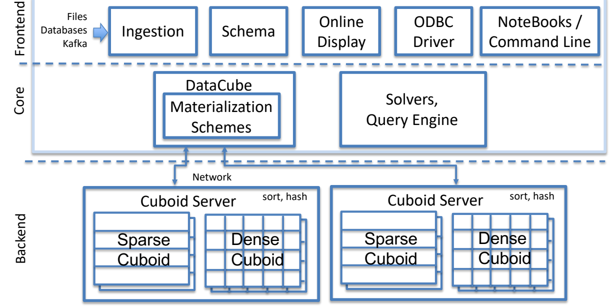


Figure 6: Sudokube architecture

The system poses no upper bound on the allowable number of (cosmetic or binary) dimensions. A schema, storing names and key-value maps for dimensions, is supported. For instance, one can group two bits and label them "quarter". The number of bits for a cosmetic dimension does not have to be determined beforehand; when loading data into a cosmetic dimension and the system runs out of keys, it adds a new binary dimension to the cosmetic dimension. Thus, the binary dimensions representing a cosmetic dimension are generally not adjacent in a cuboid's storage representation. As an optimization, when Sudokube loads data with fixed schema and domain from CSV or other formatted files, it pre-allocates all the binary dimensions adjacent to one another for faster data loading.

6.2 Cube Construction and Storage

After data loading, the frontend generates a d-dimensional base (sparse) binary cuboid as a sequence of pairs of a d-bit key and a fact value. It is then passed to the core engine, and a materialization strategy is applied. The materialization strategy determines which projections of the base cuboid are materialized and stored given a storage budget. The storage budget is provided as a fraction of the size of the base cuboid. For example, if we have a budget of 1.1, then 10 percent of the size of the base cuboid is available for materializing its projections. The user can select one of the many materialization strategies supported by Sudokube, such as picking a random subset of binary dimensions. Each strategy has its own space of all cuboids from which it picks a few cuboids of different dimensionalities to materialize. The core engine then instructs the backend to materialize and store the specified cuboids.

The Sudokube backend supports both sparse and dense representations of cuboids. A sparse representation of a k-dimensional cuboid is a sequence of pairs of k-bit keys and fact values, as is the case for the base cuboid. In contrast, a dense representation of the

same comprises 2^k-sized arrays of the fact values with the keys implicitly encoded as array indexes.

6.3 Query Processing

The core query engine receives a query as a subset of dimensions by which the facts are grouped and aggregated, and the result is returned as an array. We support further query operations through post-processing. Sudokube supports several solvers in the query engine that, when given a query, decide which materialized cuboids are to be fetched and what to do with the fetched cuboids. In this paper, we focus on the naïve solver (which computes queries by projecting from the smallest subsuming cuboid), the LP solver, and the moment solver following approaches described in Sections 3, 4, and 5, respectively. The latter two solvers can be run either in batch or online modes. The solver is invoked once after fetching all subsumed cuboids in the batch mode, and the base cuboid is never projected. Whereas in the online mode, the cuboids (including those projected from the base cuboid) are fetched in the increasing order of dimensionality. The solver is invoked at regular intervals, and intermediate results are provided via a callback function.

The core query engine processes a query in several phases. During the *Prepare* phase, depending on the solver, the engine creates a plan that specifies which cuboids to project and fetch and in what order. Given a query Q, we consider all materialized cuboids $C_1, \ldots, C_n$ with dimensions $J_1, \ldots, J_n$, respectively. Let $\Pi_{Q \cap J_i} C_i$ denote the projection of stored cuboid C_i down to the dimensions relevant to answering the query. Throughout the remainder of this paper, when we talk about fetching some materialized cuboid to answer some query, we always mean fetching its projection relevant to that query. The cost of projecting and fetching such a projection is proportional to the stored cuboid's size, but the data communicated from the backend to the solver is only as large as the projection. So we fetch relevant cuboids in the order of increasing cost, skipping cuboids that are subsumed by other cuboids whose cost is below a definable threshold. We use data structures such as set-tries [38] and optimizations such as encoding sets using integers for fast set operations. Thus at the end of *Prepare* phase, the engine has a sequence of cuboids to be fetched.

After the *Prepare* phase, the cuboids are fetched by the backend in the *Fetch* phase. This is the final phase for the naïve solver, and the fetched cuboid is returned as the query result. There is a further *Solve* phase for the other two solvers. If the query is run in batch mode, the *Solve* phase follows the *Fetch* phase, whereas, in online mode, they are interleaved. During the *Solve* phase, equations or moments from the cuboid are fed into the solver, which then produces an approximate result returned directly or through the callback function, depending on the query mode.

6.4 Current Prototype

In the current prototype of Sudokube, the backend is implemented on a single node. While all other system components are implemented in Scala, the backend is implemented in C++ and accessed via the Java Native Interface. This design keeps the bulk of the data storage and processing out of the JVM while retaining the ease of prototyping offered by Scala.

Multithreading is employed to fetch and project multiple cuboids in parallel in the backend, but cuboids are not sharded, and a cuboid is always just worked on by a single thread. In our experiments, since the naïve solver fetches a single cuboid on a single thread, in the interest of fairness, the other two solvers do not fetch cuboids in parallel even though they could. However, data cube construction does employ parallelism. The frontend and the core engine run in separate threads, facilitating online query mode. In principle, the query engine can execute different queries in parallel, even though this is not done in the experiments. The backend can load and save a data cube from and to the file system. However, the system held all the materialized cuboids in RAM in our experiments.

7 RELATED WORK

Several approaches [27, 43] were proposed to rewrite SQL queries so as to be answered using materialized views. However, for the narrow class of queries on a single relation without joins that we focus on in this paper, finding the views that answer queries is straightforward. Furthermore, Sudokube uses data from views that cannot answer the query exactly to extrapolate the result.

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 [8] and [32]. Several algorithms [1, 9, 36, 47] have been proposed to speed up the construction of the complete cube lattice, but they cannot be applied to high-dimensional data cubes. There has also been prior research on materializing only a subset of the cuboid lattice [3, 15, 16, 40] using different heuristics. All of this work can be incorporated as different materialization strategies in Sudokube.

Research on reducing the storage overhead of the data cube has led to Quotient [24], Dwarf [42], Condensed [46], and Cure [31] data cubes. While these frameworks successfully compute and store the full cube lattice at a fraction of its total unoptimized size, they still may require a storage space several orders of magnitude larger than the base cuboid, particularly in the case of high-dimensional data. This makes them impractical for large datasets.

The issue of high-dimensional data cubes was addressed in [26, 28, 30, 41], which propose 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 or bitmap indices built on the cuboids of each fragment. Thus, these approaches join and aggregate tuples on the fly for any group-by dimension and cannot support interactive-time query results for large datasets. Commercial data warehouse solutions such as Vertica [25] and Amazon Redshift [14] that rely on indexes also have the same drawback.

Prior work has also been done on performing online aggregation [2, 17, 20], which provides approximate answers that improve over time through sampling. Other approaches that use sampling to approximate queries include [7, 21, 29]. However, these sampling techniques are susceptible to outliers and skew, especially in sparse, high-dimensional data. In contrast, we do not sample data but summarize the data distribution during cube construction time.

There is also other work on summarizing data using different models. [4] 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 [5], where they model dense regions of the base cuboid using log-linear models. A similar approach is suggested in [39], where a Gaussian kernel that explains the data distribution is obtained. Alternatively, [45] proposes approximating cuboids by applying a wavelet transformation on the logarithm of partial sums of values. These approaches are similar to those used by our moment solver, but adapting them to extremely sparse high-dimensional data cubes requires work.

8 EXPERIMENTS

We now evaluate the performance of Sudokube as a system. We mainly focus on four characteristics – storage overhead, interactivity, execution time, and error. Throughout this section, we use the word *error* to describe how the result of a query returned by the Sudokube system differs from the true result. The naïve solver always has zero error, and the error for the moment solver is calculated as $\frac{\sum_i |x_i - u_i|}{\sum_i u_i}$, where x_i and u_i refer to the predicted and true values of cell i of the query cuboid, respectively. The interactivity is measured by plotting error against time for queries run in the online mode. The execution times and errors are studied by running queries in the batch mode. We repeat our experiments with 100 randomly generated queries and average the results. The queries are generated by taking (possibly empty) prefixes of binary dimensions belonging to every cosmetic dimension. We choose binary prefixes as they correspond to roll-up queries on hierarchical cosmetic 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.

8.1 Dataset Description and Pre-processing

We use two datasets in our experiments — one real and one synthetic. The first dataset, which we shall refer to as NYC [33], contains real information regarding parking violations issued in New York between 2014 and 2021. This dataset comprises 43 columns describing different details of the vehicles, the issuer of the violation, its type, and location. There are nearly 93 million rows spread roughly evenly between the eight years in this dataset.

The second dataset we use is the Star Schema Benchmark (SSB) dataset [34, 37], which contains business-oriented synthetic data modeled using a star schema. The fact table lineorder describes several details regarding items in an order such as its quantity and price and additional information is stored in the customer, part, supplier and date dimension tables. We use the dataset with a scale factor of 100, resulting in 600 million rows of lineorder data.

We perform some minor pre-processing on both datasets before loading them into Sudokube. We first flatten the SSB dataset by joining the lineorder table with all the dimension tables using the respective keys. We then discard any dimension that cannot be used in a meaningful aggregation. For example, we do not load name, address, or customer_id dimensions. We describe the exact schemas in Appendix A. Most dimensions in both datasets are categorical, and we use dictionary encoding to replace them with integers. Numerical dimensions such as tax or revenue are encoded directly as fixed-width integers. We encode date or time columns by decomposing them into constituents such as year or hour, which are separately encoded as integers. This encoding strategy results in

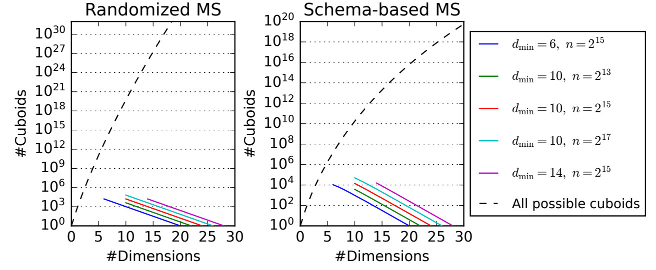


Figure 7: Distribution of materialized cuboids of different dimensionalities for the NYC dataset under different materialization parameters.

a total of 193 binary dimensions for the SSB dataset and 429 binary dimensions for the NYC dataset. Finally, we use the contribution of each line item towards the total order price as the fact for the SSB dataset and the number of rows for a given key as the fact for the NYC dataset. We allocate 8 bytes for storing each of these fact values. Thus, we have a base cuboid of size $(\lceil \frac{193}{8} \rceil + 8) \cdot 600 \cdot 10^6 \approx 19.8$ GB for the SSB dataset and a base cuboid of size $(\lceil \frac{429}{8} \rceil + 8) \cdot 93 \cdot 10^6 \approx 5.76$ GB for the NYC dataset.

8.2 Materialization Strategy

We run the experiments for two different materialization strategies, both of which are tuned by two parameters, the total number of materialized cuboids n and the minimum dimensionality of cuboids $d_{\min}$. The *Randomized Materialization Strategy* (RMS) picks binary dimensions randomly, and the *Schema-based Materialization Strategy* (SMS) picks prefixes of binary dimensions from every cosmetic dimension in the same way queries are generated. Using the query information results in a much smaller space of cuboids to materialize [3] in SMS as shown in Figure 7. In both materialization strategies, the maximum number of cuboids of some dimensionality i decreases exponentially as i increases, starting with $\frac{n}{2}$ for $i = d_{\min}$. Thus, the minimum dimensionality $d_{\min}$ is also the most common dimensionality of the materialized cuboids.

8.3 Storage

In this set of experiments, we explore the storage overhead for Sudokube. Table 1 shows the additional storage overhead as a fraction of the base cuboid size for different materialization strategies and parameters for different datasets. We observe that the overhead for SMS is smaller than that for RMS. This is because SMS picks prefixes of binary dimensions of every cosmetic dimension, resulting in very sparse cuboids. One possible explanation for the sparsity is that the data is more likely to be concentrated towards 0-cells than 1-cells for these prefixes for any cosmetic dimension as a result of our encoding. This phenomenon is amplified for cuboids containing the most significant binary dimensions from multiple cosmetic dimensions.

We observe that storage overhead increases linearly with the number of cuboids for both materialization strategies. While increasing $d_{\min}$ initially increases storage overhead exponentially, this growth slows down for higher dimensional cuboids due to increased sparsity. Generally, a given materialization strategy has

Table 1: Additional Storage Overhead

Dataset	Base Size	n	$d_{\min}$	RMS Overhead	SMS Overhead
NYC	5.76 GB	2^{13}	10	0.0445	0.0196
		2^{15}	6	0.016	0.0081
		2^{15}	10	0.1757	0.0831
		2^{15}	14	1.6368	0.5847
		2^{17}	10	0.7264	0.2932
SSB	19.8 GB	2^{15}	14	1.622	0.7002

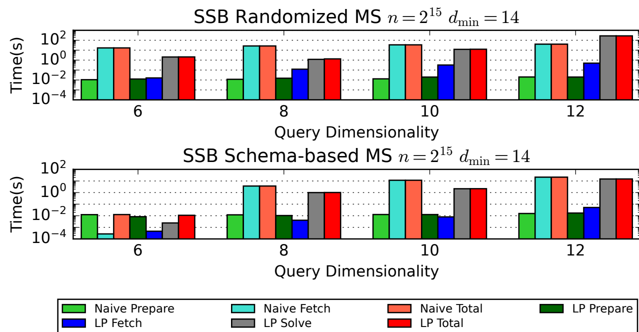


Figure 8: Execution time breakdown for the LP solver in batch mode for different query dimensionalities.

a lower overhead for the same parameters when the base cuboid is larger. This is evident while comparing the overheads of NYC and SSB data cubes for RMS with $d_{\min} = 14$ and $n = 2^{15}$. However, the overhead for SMS shows the opposite trend, likely arising from the much greater sparsity due to 429 dimensions in the NYC dataset compared to 193 in the SSB dataset outweighing effects due to smaller base cuboid size.

8.4 LP Solver: Query Dimensionality

We ran several experiments evaluating the performance of the LP solver. However, due to shortage of space, we only present the breakdown of execution time for running queries of different dimensionalities on the same data cube (SSB, $d_{\min} = 14$, $n = 2^{15}$) shown in Figure 8. The time taken for the *Prepare* phase is nearly constant in all cases. The solvers take more time to fetch cuboids as the query dimensionality increases as the cuboids being fetched become larger in dimensionality and size. We also see larger dimensionality cuboids being fetched in RMS compared to SMS. This is further exacerbated in the case of the naïve solver, which fetches the base cuboid nearly always in the case of RMS. In comparison, the naïve solver fetches cuboids of dimensionality slightly more than that of the query in the case of SMS. Finally, the time taken for computing the bounds for each value in the query result during the *Solve* phase increases significantly with query dimensionality. This experiment shows that using linear programming does not scale as much as necessary to beat the naïve solver beyond query dimensionality 12. One of the main reasons this approach is slow is because it needs numeric representations with a very high precision far greater than what the standard double-precision binary floating-point format supports. Without this high precision, the LP solver would often wrongly conclude that the linear programming problem is unsolvable even when it is not.

8.5 Moment Solver: Query Dimensionality

In this experiment, we evaluate queries of different dimensionalities using the moment solver on the data cube of the SSB dataset with $d_{\min} = 14$ and $n = 2^{15}$. Figure 9a shows the interactivity of the moment solver for various query dimensionalities. There is an increased delay for both materialization strategies until the first result appears as the query dimensionality increases due to increased *Fetch* and *Solve* time. In the case of RMS, the error drops down quickly and then flatlines while the base cuboid is projected. Whereas in the case of SMS, the solving time is higher, but the error drops to zero without projecting the base cuboid for most queries.

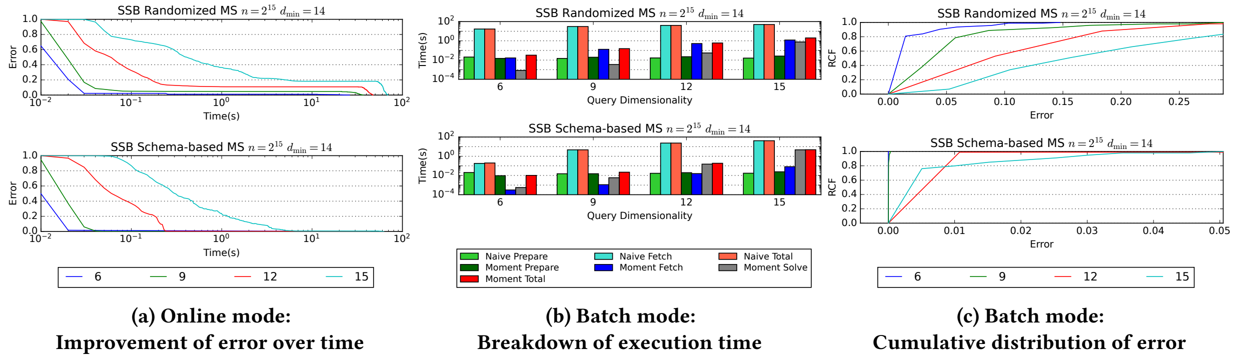
Figure 9b shows the breakdown of execution time (in batch mode) for both naïve and moment solvers. The duration of the *Prepare* phase is only slightly affected by query dimensionality for both solvers in either materialization strategy. In the case of RMS, the naïve solver almost always projects the base cuboid, and we observe a big *Fetch* time that is independent of the query dimensionality. On the other hand, in the case of SMS, there is some materialized cuboid with a slightly larger dimensionality than the query dimensionality that can fully answer the query, and we observe a shorter *Fetch* time for the naïve solver that increases with the query dimensionality. For the moment solver, both *Fetch* time and *Solve* time increase with query dimensionality. Furthermore, the moment solver fetches higher dimensional cuboids and produces fewer moments in the case of RMS compared to SMS. Therefore, we observe a higher *Fetch* time and shorter *Solve* time for RMS than SMS. In both materialization strategies, the moment solver returns approximate results faster than the naïve solver up to query dimensionality of 15, after which its *Solve* time exceeds the naïve solver *Fetch* time.

For both materialization strategies, the execution time for queries in the online mode can be much worse than the same for batch mode for two reasons. Firstly, the base cuboid is allowed to be projected in the online mode but not in the batch mode causing an increased *Fetch* time. Secondly, the extrapolation algorithm is invoked repeatedly in the online mode, causing the *Solve* time to increase by a factor. This is particularly pronounced for high-dimensional queries where the *Solve* time is the dominant cost.

Figure 9c shows the relative cumulative frequency (RCF) of errors for the moment solver for different query dimensionalities. As the query dimensionality increases, we observe higher values for the error as the number of unknown moments increases. Around 90% of queries have an error less than 0.1 for query dimensionality 9 in the case of RMS, whereas in the case of SMS, 90% of queries have an error less than 0.02 for query dimensionality as high as 15. Overall, SMS results in more cuboids relevant to the query being materialized, leading to lower error.

8.6 Moment Solver: Materialization Parameters

We now study the impact of materialization parameters on the performance of the Sudokube system. We fix the query dimensionality to 10 and run queries on data cubes with the same base cuboid but

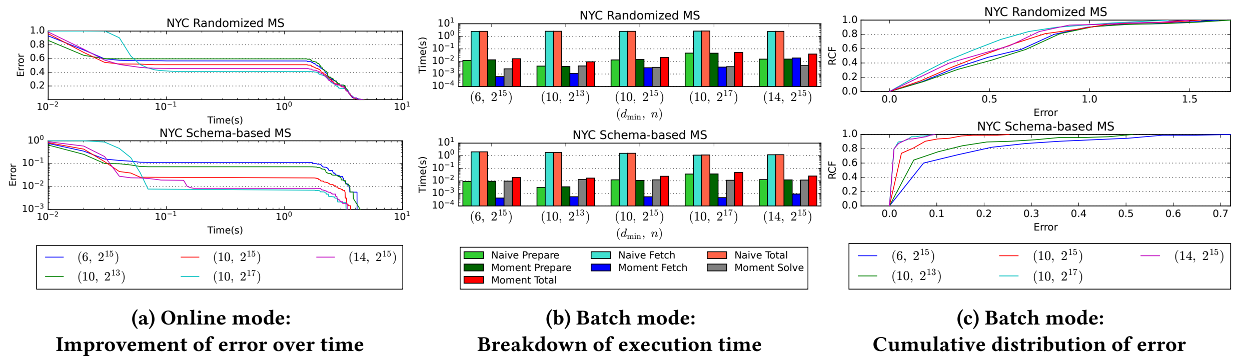


(a) Online mode: Improvement of error over time

(b) Batch mode: Breakdown of execution time

(c) Batch mode: Cumulative distribution of error

Figure 9: Moment solver experiments for different query dimensionalities

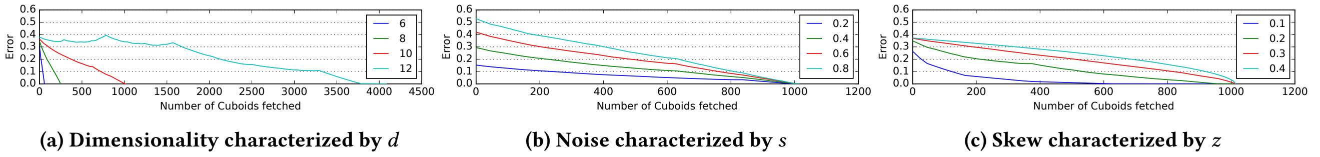


(a) Online mode: Improvement of error over time

(b) Batch mode: Breakdown of execution time

(c) Batch mode: Cumulative distribution of error

Figure 10: Moment solver experiments for different materialization parameters $(d_{\min}, n)$.



(a) Dimensionality characterized by d

(b) Noise characterized by s

(c) Skew characterized by z

Figure 11: Impact of various data distribution parameters of the microbenchmark on the moment solver error in online mode.

different values for $d_{\min}$ and n. Figure 10a shows the interactivity of the moment solver for different materialization parameters of the NYC data cube. The number of cuboids has a more significant impact on the time until the first result than the minimum cuboid dimensionality due to increased time for the *Prepare* phase. For both materialization strategies, the error drops quickly in the first few hundred milliseconds, and then it flatlines for the next 3 to 4 seconds while the base cuboid is projected. The lowest error before that happens is given by the data cubes with the maximum value of $d_{\min}$ and n, which can also be seen in Figure 10c. Increasing n results in a better error, increasing $d_{\min}$ improves it even more.

Figure 10b shows the impact of these parameters on different phases of the query execution. The *Prepare* time increases linearly in all cases with the number of cuboids n since it computes the intersection of dimensions between the materialized cuboids and the query. However, it is not affected by $d_{\min}$ as our fast set operations are not significantly impacted by the size of the sets. The naïve solver nearly always projects the base cuboid, and its *Fetch* time is nearly unaffected by materialization parameters in this scenario. On the other hand, the moment solver fetches smaller cuboids, and its *Fetch* time increases exponentially with $d_{\min}$. Increasing either parameter increases *Solve* time indirectly as more moments are available to be processed. While comparing the two materialization strategies, we again observe that RMS has a higher *Fetch* time but a lower *Solve* time than SMS for our moment solver as the former results in fetching larger cuboids to produce fewer moments.

While comparing the distributions of errors for queries run in the batch mode shown in Figure 10c, it is evident that the distributions differ only slightly for RMS as opposed to SMS. This is because the space of all possible cuboids is so large (see Figure 7) that the parameters used in this experiment do not make much of a difference. Nevertheless, in both cases, $d_{\min}$ has a much more significant impact on error than n. It does not come as a surprise

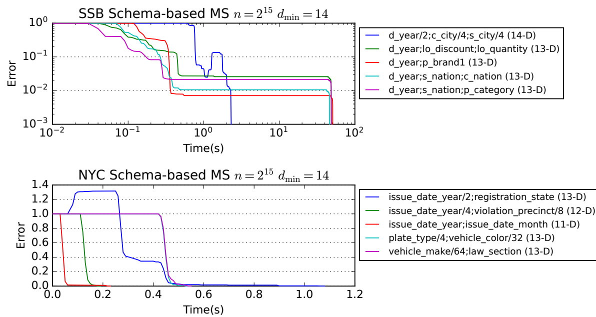


Figure 12: Performance of the moment solver in online mode on natural queries. Each line denotes an aggregation query grouped by the labeled dimensions. k consecutive values of dimension X are grouped together when labeled X/k.

since increasing $d_{\min}$ increases the number of available moments exponentially, whereas increasing n increases it only linearly.

8.7 Moment Solver: Natural Query Accuracy

In this experiment, we demonstrate the performance of our moment solver for five hand-chosen queries that are natural in the context of the two datasets. We only consider queries (without slicing) with dimensionality less than 15 in this experiment. We identify queries by the group-by dimensions and sometimes coarsen the granularity by taking only prefixes of cosmetic dimensions. For example, grouping by issue_date_year/2 indicates that we aggregate over periods of two consecutive years. We extend this notion also to cover non-numeric columns with similar behavior; grouping by plate_type/4 aggregates the fact values of four consecutive plate types into one entry. Figure 12 shows how the error improves with time in the online mode for the indicated queries on SSB and NYC data cubes. For most of these queries, Sudokube yields a result with less than 5% error within a second.

8.8 Microbenchmarks

We also run microbenchmarks to study how the error decreases as more cuboids are fetched under various conditions. For these experiments, we generate synthetic data for some d-dimensional cuboid with dimensions $I = \{0, \dots, d-1\}$ as described next, materialize the entire lattice comprising 2^d cuboids and then query the base cuboid. The fact associated with every cell in the cuboid is sampled from a log-normal distribution whose parameters depend on the position of the cell. For some external parameters z and s, the mean value μ and the standard deviation σ of the distribution for the cell with mapping $p : I \longrightarrow \{0, 1\}$ is given by

$$\mu = z^{\text{ones}(p)} \cdot (1-z)^{\text{zeroes}(p)}, \qquad \sigma = s \cdot \mu.$$

where the functions ones(p) and zeroes(p), respectively, count the number of dimensions mapped to one and zero in p. We multiply the value sampled from this log-normal distribution by 10^5 and then truncate the fraction.

First, we vary the dimensionality d, fixing $z = 0.25$ and $s = 0.5$ and the results are shown in Figure 11a. As the query dimensionality increases, the moment solver requires exponentially more cuboids to answer the query, and the error drops nearly linearly as more of these cuboids are fetched. Next, we study the impact of the parameter s. This parameter determines the variance of the distribution from which we sample the facts. The higher this parameter, the noisier the data is. Figure 11b shows the results for different s fixing $d = 10$ and $z = 0.25$. As expected, our estimate becomes worse as the data becomes noisier. Finally, we study the impact of skew in the data. We introduce skew in the data by lowering the value of z, which causes the fact values to concentrate in the 0-cells compared to the 1-cells in any dimension. Figure 11c shows the result for different values of z keeping $d = 10$ and $s = 0.5$. We observe that a lower value of z results in an increase in the number of zero values and moments. A zero moment, in particular, is very beneficial to the solver as it can immediately infer several higher-order moments also to be zero, thus lowering the error significantly when discovered. As a result, the solver only needs fewer cuboids to reach the same error level compared to the case when z is higher.

9 CONCLUSION AND FUTURE WORK

This paper introduces Sudokube, an OLAP system that supports interactive-time query results on high-dimensional data cubes. The high dimensionality of a data cube makes it impractical to materialize all but a small fraction of its cuboids. Sudokube employs several techniques that approximate the query cuboid from its projections to support interactive-time querying. Our experiments show that our moment solver produces good approximations with less than 1% average error in under a second for queries of dimensionalities as high as 12 when the naïve solver takes nearly two orders of magnitude longer time despite having all cuboids fetched from RAM. Should the cuboids be fetched from disk instead, the relative speed-up of the moment solver over the naïve one would be higher.

The Sudokube system, as discussed in this paper, is far from complete. The current prototype supports slicing and dicing operations only through post-processing on the complete query result, and we are working on making the core engine support them efficiently. Investigations on alternative materialization strategies are also ongoing. Currently, the moment solver extrapolates unknown moments by setting the corresponding mixed central moments to zero. As future work, we plan to explore other extrapolation techniques involving cumulants, copulae, and graphical models such as Markov Random Fields. We also plan to scale our backend to supporting cuboids that would not fit in RAM, first by utilizing secondary storage on a single node and then moving on to a scale-out architecture. Additional optimizations on backend operations involving sharding of cuboids and multithreaded and distributed aggregation of these cuboid shards are being considered.

ACKNOWLEDGMENTS

We wish to thank Peter Lindner for his detailed feedback on the draft and insightful discussions on this topic.

REFERENCES

[1] Sameet Agarwal, Rakesh Agrawal, Prasad Deshpande, Ashish Gupta, Jeffrey F. Naughton, Raghu Ramakrishnan, and Sunita Sarawagi. 1996. On the Computation of Multidimensional Aggregates. In *Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB '96)*. 506–521.

[2] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. 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)*. 29–42.

[3] Elena Baralis, Stefano Paraboschi, and Ernest Teniente. 1997. Materialized Views Selection in a Multidimensional Database. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB '97)*. 156–165.

[4] Daniel Barbará and Mark Sullivan. 1997. Quasi-Cubes: Exploiting Approximations in Multidimensional Databases. *SIGMOD Record* 26, 3 (1997), 12–17.

[5] Daniel Barbará and Xintao Wu. 2000. Using Loglinear Models to Compress Datacubes. In *Proceedings of the 1st International Conference on Web-Age Information Management (WAIM '00)*. 311–323.

[6] Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto. 2007. Fourier Meets Möbius: Fast Subset Convolution. In *Proceedings of the 39th Annual ACM Symposium on Theory of Computing (STOC '07)*. 67–74.

[7] Surajit Chaudhuri, Gautam Das, and Vivek Narasayya. 2007. Optimized Stratified Sampling for Approximate Query Processing. *ACM Transactions on Database Systems* 32, 2 (jun 2007), 9.

[8] Surajit Chaudhuri and Umeshwar Dayal. 1997. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Record* 26, 1 (1997), 65–74.

[9] Zhimin Chen and Vivek R. Narasayya. 2005. Efficient Computation of Multiple Group By Queries. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD '05)*. 263–274.

[10] James W. Cooley and John W. Tukey. 1965. An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. Comp.* 19, 90 (1965), 297–301.

[11] Roberto Fontana and Patrizia Semeraro. 2018. Representation of multivariate Bernoulli distributions with a given set of specified moments. *Journal of Multivariate Analysis* 168 (2018), 290–303.

[12] Saul I. Gass. 2003. *Linear Programming: Methods and Applications*. Courier Corporation.

[13] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. 1997. Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub Totals. *Data Mining and Knowledge Discovery* 1, 1 (1997), 29–53.

[14] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon Redshift and the Case for Simpler Data Warehouses. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. 1917–1923.

[15] Himanshu Gupta and Inderpal Singh Mumick. 2005. Selection of Views to Materialize in a Data Warehouse. *IEEE Transactions on Knowledge and Data Engineering* 17, 1 (2005), 24–43.

[16] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. 1996. Implementing Data Cubes Efficiently. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD '96)*. 205–216.

[17] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. 1997. Online Aggregation. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data (SIGMOD '97)*. 171–182.

[18] David C. Hoaglin, Frederick Mosteller, and John W. Tukey (Eds.). 2006. *Exploring Data Tables, Trends, and Shapes*. John Wiley & Sons.

[19] Kenneth Hoffman. 1971. *Linear Algebra*. Englewood Cliffs, NJ, Prentice-Hall.

[20] Chris Jermaine, Subramanian Arumugam, Abhijit Pol, and Alin Dobra. 2008. Scalable approximate query processing with the DBO engine. *ACM Transactions on Database Systems* 33, 4 (2008), 23:1–23:54.

[21] Ruoming Jin, Leonid Glimcher, Chris Jermaine, and Gagan Agrawal. 2006. New Sampling-Based Estimators for OLAP Queries. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE '06)*. 18.

[22] Minsuk Kahng, Dezhi Fang, and Duen Horng (Polo) Chau. 2016. Visual exploration of machine learning results using data cube analysis. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics (HILDA '16)*. 1–6.

[23] Niranjan Kamat, Prasanth Jayachandran, Karthik Tunga, and Arnab Nandi. 2014. Distributed and interactive cube exploration. In *IEEE 30th International Conference on Data Engineering (ICDE '14)*. 472–483.

[24] Laks V. S. Lakshmanan, Jian Pei, and Jiawei Han. 2002. 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)*. 778–789.

[25] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. 2012. The Vertica Analytic Database: C-Store 7 Years Later. *Proceedings of the VLDB Endowment 2012* 5, 12 (aug 2012), 1790–1801.

[26] Fangling Leng, Yubin Bao, Ge Yu, Daling Wang, and Yuntao Liu. 2006. 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)*. 557–568.

[27] Alon Y. Levy, Alberto O. Mendelzon, and Yehoshua Sagiv. 1995. Answering Queries Using Views. In *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '95)*. 95–104.

[28] Xiaolei Li, Jiawei Han, and Hector Gonzalez. 2004. High-Dimensional OLAP: A Minimal Cubing Approach. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB '04)*. 528–539.

[29] Xiaolei Li, Jiawei Han, Zhijun Yin, Jae-Gil Lee, and Yizhou Sun. 2008. Sampling cube: a framework for statistical olap over sampling data. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*. 779–790.

[30] Eric Lo, Ben Kao, Wai-Shing Ho, Sau Dan Lee, Chun Kit Chui, and David W. Cheung. 2008. OLAP on sequence data. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIMOD '08)*. 649–660.

[31] Konstantinos Morfonios and Yannis E. Ioannidis. 2006. CURE for Cubes: Cubing Using a ROLAP Engine. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB '06)*. 379–390.

[32] Konstantinos Morfonios, Stratis Konakas, Yannis E. Ioannidis, and Nikolaos Kotsis. 2007. ROLAP implementations of the data cube. *Comput. Surveys* 39, 4 (2007), 12.

[33] New York City Department of Finance. 2021. *Parking Violations Issued - Fiscal Year 2021*. Retrieved August 4, 2022 from https://data.cityofnewyork.us/City-Government/Parking-Violations-Issued-Fiscal-Year-2021/kvfd-bves

[34] Patrick E. O'Neil, Elizabeth J. O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The Star Schema Benchmark and Augmented Fact Table Indexing. In *Performance Evaluation and Benchmarking (TPTPC '09)*. 237–252.

[35] Athanasios Papoulis and S Unnikrishna Pillai. 2002. *Probability, Random Variables and Stochastic Processes* (4 ed.). McGraw-Hill Professional.

[36] Kenneth A. Ross and Divesh Srivastava. 1997. Fast Computation of Sparse Datacubes. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB '97)*. 116–125.

[37] Eyal Rozenberg. 2020. Star Schema Benchmark data set generator (ssb-dbgen). Retrieved August 4, 2022 from https://github.com/eyalroz/ssb-dbgen

[38] Iztok Savnik. 2013. Index Data Structure for Fast Subset and Superset Queries. In *Availability, Reliability, and Security in Information Systems and HCI*. Springer, 134–148.

[39] Jayavel Shanmugasundaram, Usama M. Fayyad, and Paul S. Bradley. 1999. 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)*. 223–232.

[40] Amit Shukla, Prasad Deshpande, and Jeffrey F. Naughton. 1998. Materialized View Selection for Multidimensional Datasets. In *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB '98)*. 488–499.

[41] Rodrigo Rocha Silva, Celso Massaki Hirata, and Joubert de Castro Lima. 2020. Big high-dimension data cube designs for hybrid memory systems. *Knowledge and Information Systems* 62, 12 (2020), 4717–4746.

[42] Yannis Sismanis, Antonios Deligiannakis, Nick Roussopoulos, and Yannis Kotidis. 2002. Dwarf: shrinking the PetaCube. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD '02)*. 464–475.

[43] Divesh Srivastava, Shaul Dar, H. V. Jagadish, and Alon Y. Levy. 1996. Answering Queries with Aggregation Using Views. In *Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB '96)*. 318–329.

[44] Jozef L Teugels. 1990. Some representations of the multivariate Bernoulli and binomial distributions. *Journal of Multivariate Analysis* 32, 2 (1990), 256–268.

[45] Jeffrey Scott Vitter, Min Wang, and Balakrishna R. Iyer. 1998. Data Cube Approximation and Histograms via Wavelets. In *Proceedings of the 1998 ACM CIKM International Conference on Information and Knowledge Management (CIKM '98)*. 96–104.

[46] Wei Wang, Hongjun Lu, Jianlin Feng, and Jeffrey Xu Yu. 2002. Condensed Cube: An Efficient Approach to Reducing Data Cube Size. In *Proceedings of the 18th International Conference on Data Engineering (ICDE '02)*. 155–165.

[47] Yihong Zhao, Prasad Deshpande, and Jeffrey F. Naughton. 1997. An Array-Based Algorithm for Simultaneous Multidimensional Aggregates. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data (SIGMOD '97)*. 159–170.

A DATASET SCHEMA

Tables 2 and 3 show the cosmetic dimensions as well as the number of binary dimensions assigned to each of them by Sudokube for SSB and NYC datasets, respectively.

B PROOFS

In this section, we give the proofs of theorems from the main sections of the paper. Before we prove the correctness of the basis construction from Section 4, we need to introduce some additional notations. For a given cuboid C_J and a slice denoted by a mapping $p : L \longrightarrow \{0, 1\}$ for some set $L \subseteq J$, we define the equations for the slice p of the cuboid C_J as

$$C_J|_p := \{e_q \in C_J \mid q : J \longrightarrow \{0, 1\}, p(\ell) = q(\ell) \text{ for all } \ell \in L\}.$$

Table 2: Schema for SSB dataset

Cosmetic Dimension	#bits
order_date	14
ord_priority	3
ship_priority	0
quantity	6
extended_price	24
discount	4
revenue	24
sup_cost	17
tax	4
commit_date	14

Lemma B.1. *Given any two sets $L_1, L_2 \subseteq Q$ and a slice $p : L_1 \cap L_2 \to \{0,1\}$ present in both C_{L_1} and C_{L_2}, we have*

$$\sum C_{L_1}|_p = C_{L_1 \cap L_2}(p) = \sum C_{L_2}|_p.$$

Proof. From its definition, we know that the set $C_{L_1}|_p$ contains equations e_q for every $q : L_1 \to \{0,1\}$ that is consistent with p on $L_1 \cap L_2$. The equations e_q themselves contain the sum of variables x_r for $r : Q \to \{0,1\}$ that are consistent with q on L_1. Combining them, we have

$$\sum_q C_{L_1}|_p(q) = \sum \left\{ x_r \,\middle|\, \begin{array}{l} r : Q \to \{0,1\}, \\ r(\ell) = p(\ell) \text{ for all } \ell \in L_1 \cap L_2 \end{array} \right\} = C_{L_1 \cap L_2}(p)$$

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

Example B.2. 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 $C_{\{3,0\}}$:

$$\begin{array}{ll} C_{\{1,0\}}|_{\{0\mapsto b\}} = \{e_{*0b}, e_{*1b}\} & C_{\{1,0\}}|_{\{1\mapsto b\}} = \{e_{*b0}, e_{*b1}\} \\ C_{\{3,1\}}|_{\{1\mapsto b\}} = \{e_{0b*}, e_{1b*}\} & C_{\{3,1\}}|_{\{3\mapsto b\}} = \{e_{b0*}, e_{b1*}\} \\ C_{\{3,0\}}|_{\{0\mapsto b\}} = \{e_{0*b}, e_{1*b}\} & C_{\{3,0\}}|_{\{3\mapsto b\}} = \{e_{b*0}, e_{b*1}\} \end{array}$$

Then, from Lemma B.1, we have the following linear dependencies for $b \in \{0,1\}$.

$$\begin{array}{ll} e_{0b*} = e_{*b0} + e_{*b1} - e_{1b*} & \text{from } C_{\{1,0\}}|_{\{1\mapsto b\}} \text{ and } C_{\{3,1\}}|_{\{1\mapsto b\}} \\ e_{0*b} = e_{*0b} + e_{*1b} - e_{1*b} & \text{from } C_{\{1,0\}}|_{\{0\mapsto b\}} \text{ and } C_{\{3,0\}}|_{\{0\mapsto b\}} \\ e_{b*0} = e_{b0*} + e_{b1*} - e_{b*1} & \text{from } C_{\{3,1\}}|_{\{3\mapsto b\}} \text{ and } C_{\{3,0\}}|_{\{3\mapsto b\}} \end{array}$$

Next, we establish two auxiliary results. We say that a variable x *dominates* an equation e if it is same as or appears before (according to order $\prec$) the minimal variable $\hat{x}_e$ of the equation. A variable x dominates a set of equations if it dominates each of its members. From now on, we will allow a slight abuse of notation and use the name of the projection C also to denote its set of equations.

Lemma B.3. *Given an equation $e_q \in C_J$ and a slice $p : L \to \{0,1\}$ for some $L \subseteq J$ such that $q(\ell) = p(\ell)$ for every $\ell \in L$ and $q(\ell) = 0$ otherwise, then $\hat{x}_{e_q}$ dominates $C_J|_p$.*

Proof. $C_J|_p$ is defined as the equations $e_r \in C_J$ such that r is consistent with p. q extends p with $q(j) = 0$ on all $j \in J \setminus L$. Obviously, $\hat{x}_{e_q}$ dominates all the e_r. □

Lemma B.4. *Given two equations $e_{q_1} \in C_{J_1}$ and $e_{q_2} \in C_{J_2}$ such that $\hat{x}_{e_{q_1}} = \hat{x}_{e_{q_2}}$, then $q_1(j) = 0$ for all $j \in J_1 \setminus J_2$.*

Proof. Let $x_r := \hat{x}_{e_{q_1}}$ (so $r : I \to \{0,1\}$ is consistent with q_1). Assume, for a proof by contradiction, that $q_1(j) = 1$ for some $j \in J_1 \backslash J_2$, so $r(j) = 1$, too. Let r' be defined as $r'(j) := 0$ and $r'(i) := r(i)$ for all $i \in (I \setminus \{j\})$. Then, since q_2 is undefined on j, in addition to x_r, the variable $x_{r'}$ occurs in e_{q_2} as well. But then, since $x_{r'} \prec x_r$, we would have $\hat{x}_{e_{q_2}} \prec \hat{x}_{e_{q_1}}$, violating our assumptions. So $q_1(j) = 0$ for all $j \in J_1 \backslash J_2$. □

Example B.5. Consider again our running example, and equations $e_{*01} \in C_{\{1,0\}}$ and $e_{0*1} \in C_{\{3,0\}}$ with $\hat{x}_{e_{*01}} = \hat{x}_{e_{0*1}} = x_{001}$. Then, for the slice $p = \{0 \mapsto 1\}$, we have $C_{\{1,0\}}|_p = \{e_{*01}, e_{*11}\}$, and $C_{\{3,0\}}|_p = \{e_{0*1}, e_{1*1}\}$. Variable x_{001} dominates both $C_{\{1,0\}}|_p$ and $C_{\{3,0\}}|_p$, as required by the combination of Lemmas B.3 and B.4.

Proof of Theorem 4.3. Let E be an arbitrary set of linear equations constructed according to Theorem 4.3. This set is clearly linearly independent. Each row vector e has the leftmost nonzero element $\hat{x}_e$, 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 5 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 $B := C_{J_1}$. During step i, for each equation $e \in C_{J_i}$, processed in the order of increasing $\hat{x}_e$, 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 $\hat{x}_e$. 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.

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 e be of the form e_q and let $p : J_h \cap J_{i+1} \to \{0,1\}$ be the restriction of q to $J_h \cap J_{i+1}$. Then, $d \in C_{J_h}|_p$ and $e \in C_{J_{i+1}}|_p$. By Lemmas B.4 and B.3, all the equations in $C_{J_{i+1}}|_p \setminus \{e\}$ are dominated by $\hat{x}_e$ and thus have been previously processed by the algorithm. By Lemma B.1, e is linearly dependent with $C_{J_h}|_p$ and $C_{J_{i+1}}|_p \setminus \{e\}$, and thus, by the induction hypothesis, with B. □

Table 3: Schema for NYC dataset

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

Example B.6. If we execute the algorithm of the proof of Theorem 4.3 on the sales data cube, the first equation we encounter that is not added to B is e_{01*} ($q = \{3 \mapsto 0, 1 \mapsto 1\}$) 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, $d = e_{*11}$, $\hat{x}_d = x_{011}$, and $J_h = \{1, 0\}$. We also have $J_h \cap J_{i+1} = \{1\}$, and we construct $p = \{1 \mapsto 1\}$ from q. Therefore, $C_{J_h}|_p = \{e_{*10}, e_{*11}\}$ and $C_{J_{i+1}}|_p = \{e_{01*}, e_{11*}\}$. Indeed, $\hat{x}_{e_{01*}} = x_{010}$ dominates $C_{J_{i+1}}|_p$. $C_{J_h}|_p \cup (C_{J_{i+1}}|_p \setminus \{e\})$ is a subset of B and therefore linearly dependent with it.

Next, we show the correctness of the relationship between moments and values, the extrapolation formula, and the extrapolation algorithm from Section 5.

Proof of Proposition 5.2. Statement (2) immediately follows from Statement (1) because of the properties of Kronecker product given that $\mathbf{W} = \mathbf{M}^{-1}$. We only need to prove Statement (1), which we do by induction on the size of the cuboid d. Let the columns be labeled $0 \dots d-1$.

Base Case : For $d = 1$ from the definition of moments, we know that $m_\emptyset = x_0 + x_1$ and $m_{\{0\}} = x_1$. Thus, $\mathbf{m} = \mathbf{M}\mathbf{x}$.

Inductive Case : We assume that the statements are true for $d = k$ and prove them for $d = k+1$. The $(k+1)$-dimensional cuboid can be sliced along the last dimension into two k dimensional cuboids. Let $\mathbf{m}^{(i)}$ and $\mathbf{x}^{(i)}$ denote the moments and values of k-dimensional cuboid corresponding to the value of the last dimension for $i \in \{0, 1\}$. Furthermore, let $\mathbf{m}_i$ and $\mathbf{x}_i$ denote the first and second halves of the moments and values of the $(k+1)$-dimensional cuboid for $i = 0$ and $i = 1$ respectively. Then, we have $\mathbf{x}_i = \mathbf{x}^{(i)}$ from how the ordering is defined for values within the vector $\mathbf{x}$. Similarly, we have $\mathbf{m}_0 = \mathbf{m}^{(0)} + \mathbf{m}^{(1)}$ and $\mathbf{m}_1 = \mathbf{m}^{(1)}$. Using the inductive property, we have $\mathbf{m}^{(i)} = \mathbf{M}^{\otimes k}\mathbf{x}^{(i)}$. Combining them all together, we have

$$\begin{aligned} \mathbf{m} &= \begin{bmatrix} \mathbf{m}_0 \\ \mathbf{m}_1 \end{bmatrix} = \begin{bmatrix} \mathbf{M}^{\otimes k} & \mathbf{M}^{\otimes k} \\ \mathbf{0} & \mathbf{M}^{\otimes k} \end{bmatrix} \begin{bmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \end{bmatrix} \\ &= \left(\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \otimes \mathbf{M}^{\otimes k} \right) \begin{bmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \end{bmatrix} = \mathbf{M}^{\otimes (k+1)} \mathbf{x}. \end{aligned}$$ □

Proof of Proposition 5.4. The powerset 2^J of J can be partitioned into known subsets $K \in \text{Known}(J)$ and unknown subsets $U \in 2^J \setminus \text{Known}(J)$. As per our assumption of uncorrelatedness, setting $\mu_U = 0$ for every unknown subset U in Equation 3 gives Equation 4. Clearly, when $J \in \text{Known}(Q)$, then $\text{Known}(J) = 2^J$ and $U = \emptyset$, giving $m'_J = m_J$. □

Proof of Theorem 5.5. We use induction on the number of steps n to prove the correctness of Algorithm 1.

Base Case : n = 0. Initially, only moments m_J with $|J| \leq 1$ are known. From its definition, $\mu_\emptyset = m_\emptyset$ and $\mu_J = 0$ for any $J \subseteq Q$ with $|J| = 1$. Plugging these values into Equation 4 gives $m'_J = m_\emptyset * p_J$ for every $J \subseteq Q$.

Inductive Case: We assume that the algorithm is correct upto n steps and prove that it remains correct after one more step. Let the new subset being added to Known(Q) be S. Consider some arbitrary set $J \subseteq Q$. Since Known(J) contains only subsets of J from Known(Q), adding S to Known(Q) results in a change to Known(J) for only those J that are supersets of S. Therefore, only the extrapolated moments m'_J with $J \supseteq S$ change from the previous step. Furthermore, from Equation 4 the incremental change to these m'_J upon adding S to Known(J) is precisely $\mu_S * p_{J \setminus S}$. Thus, the only thing left to be proved is the correctness of the value of μ_S computed in Line 6. Since we add moments of all subsets before adding moment of a set, at step n, Known(S) = $2^S \setminus \{S\}$. Thus, we have the following two equations

$$m'_S = \sum_{K \subset S} \mu_K \cdot p_{S \setminus K} \quad , \quad \mu_S = m_S - \sum_{K \subset S} \mu_K \cdot p_{S \setminus K}.$$

The first equation comes from the inductive argument, and the second comes from separating the term corresponding to $K = S$ from Equation 3 for m_S and rearranging terms. Combining these two equations proves the correctness of μ_S in Line 6. □