Aggregation and Exploration of High-Dimensional Data Using the Sudokube Data Cube Engine

Sachin Basil John
sachin.basiljohn@epfl.ch
EPFL
Lausanne, Switzerland

Peter Lindner
peter.lindner@epfl.ch
EPFL
Lausanne, Switzerland

Zhekai Jiang*
zhekai.jiang@mail.mcgill.ca
McGill University
Montreal, Canada

Christoph Koch
christoph.koch@epfl.ch
EPFL
Lausanne, Switzerland

ABSTRACT

We present Sudokube, a novel system that supports interactive speed querying on high-dimensional data using partially materialized data cubes. Given a storage budget, it judiciously chooses what projections to precompute and materialize during cube construction time. Then, at query time, it uses whatever information is available from the materialized projections and extrapolates missing information to approximate query results. Thus, Sudokube avoids costly projections at query time while also avoiding the astronomical compute and storage requirements needed for fully materialized high-dimensional data cubes. In this paper, we show the capabilities of the Sudokube system and how it approximates query results using different techniques and materialization strategies.

KEYWORDS

data cubes, approximate query processing, online aggregation, online analytical processing, data exploration, data visualization

1 INTRODUCTION

Online analytical processing (OLAP) [2] forms an important class of querying methods for aggregated data, used extensively in business intelligence [3], exploratory data analysis [8, 10], and feature engineering [9]. Data cubes [5] are often used to accelerate such queries using precomputation of several projections of the data and materializing them in the form of cuboids. Additionally, they offer simple operations such as roll-up, drill-down, and slice, and users do not need to write complex code to analyze the data. However, they cannot handle high-dimensional data very well as they can only materialize some of the cuboids due to computation and storage limitations [1, 11]. Therefore, they typically compute queries through projection from the smallest subsuming cuboid [6], which can be slow for large cuboids.

In this paper, we present Sudokube[1] [1], a novel data cube system that allows fast querying of high-dimensional data and provides a versatile suite of functionalities to perform OLAP tasks. Unlike classical approaches that answer queries exactly by projecting the smallest subsuming cuboid, Sudokube uses every available projection of a query (as shown in Figure 1) to approximate its result at interactive speeds in an online [7] fashion. The results are updated as more projections are processed in the increasing order of dimensionality, and an exact answer is obtained after processing a cuboid that contains the entire query as its projection.

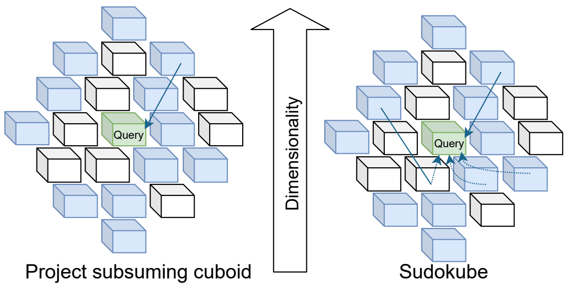

Figure 1: Approaches for querying partial data cubes

Supporting interactive-time querying for high-dimensional data gives Sudokube several advantages over classical approaches. Firstly, Sudokube can load all data without forcing users to build an Extract-Transform-Load (ETL) pipeline that distills the data down to a minimal number of dimensions. This allows off-the-shelf OLAP, enabling even non-technical users to use the data cube's interactive data exploration features to architect data cleaning and transformation scripts. Secondly, Sudokube offers data scientists a simple and interactive way to explore intrinsically high-dimensional data for feature engineering. Thirdly, Sudokube can load denormalized data directly, eliminating the need for users to design star or snowflake schemas [5] to separate dimension attributes from a central fact table. Sudokube stores each attribute of a hierarchical dimension as individual dimensions and avoids the expensive joins required for coarsening it at runtime. Finally, Sudokube enables users to break down dimensions into smaller components and query every bit of the internal binary encoding of keys in each component. This allows for powerful wildcard-based pattern matching on dimensions through simple roll-up queries.

In this paper, we showcase the Sudokube system in two scenarios that illustrate its utility and highlight the above-mentioned advantages. We also demonstrate the trade-offs associated with different materialization and approximation strategies.

*Work done while at EPFL
[1]https://github.com/epfldata/sudokube

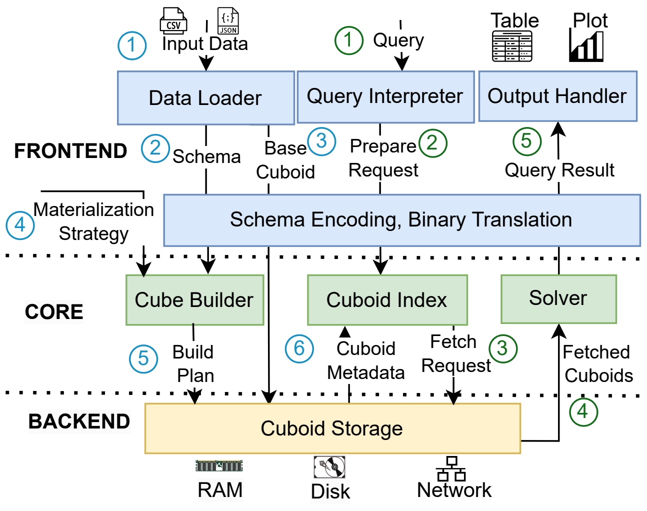

Figure 2: Sudokube workflow

2 SYSTEM ARCHITECTURE

Sudokube comprises three components – *frontend*, *core query execution engine* and *backend*. The frontend offers a basic user interface for data loading, querying, and displaying the results. It also provides schema support and handles the binary encoding of keys. The rest of the Sudokube system sees only the binary dimensions representing the individual bits of these keys. The core engine decides what cuboids to materialize during cube construction time and what cuboids to fetch and process during query time. It allows users to choose from several solvers to extrapolate query results from the fetched cuboids. Finally, the backend is responsible for storing, projecting and retrieving materialized cuboids.

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

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

2.1 Cube Specification and Querying

Sudokube supports all the fundamental data cube operations. Before a cube is constructed, Sudokube requires all *measures* of interest to be specified so they can be precomputed. Users can designate individual columns as measures or specify functions that produce

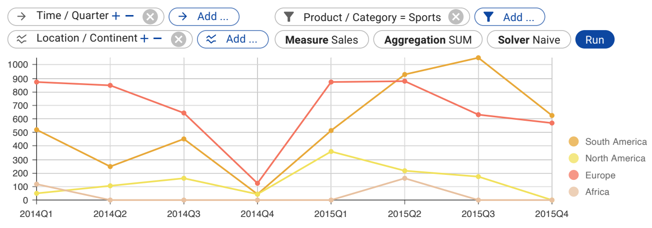
Time / Quarter + –
Add ...
Product / Category = Sports
Add ...
Location / Continent + –
Add ...
Measure Sales
Aggregation SUM
Solver Naive
Run

Figure 3: Sudokube UI for querying

measure values from multiple columns. After the cube has been built, users can query one or more of these measures. Sudokube allows users to specify a variety of aggregation operations such as sum, count, average, variance, correlation and linear regression coefficient. Users can *pivot* dimensions across both the horizontal and vertical axes. In addition to traditional dimensional hierarchies, Sudokube allows fine-grained virtual hierarchies for each dimension where the consecutive values are grouped together in sizes of powers of two. For example, users can specify a hierarchy for the time dimensions such as Year - Month - Day, and Sudokube additionally offers virtual dimensions such as Year/4 or Day/2 where four consecutive years or two consecutive days are grouped together, respectively. Users can then *drill down* on the result by either going down one level on the hierarchy for some dimension or adding a new one to some axis. Conversely, the user can *roll up* going up the hierarchy for some dimension or removing one. Finally, users can *slice* and *dice* on multiple dimensions to filter the keys in the result. We provide users with a visual interface (inspired by existing visualization tools such as [12]) as shown in Figure 3 for easily specifying such queries.

Sudokube contains a library that implements other operations through post-processing. These operations include window-based aggregations, user-defined grouping of values, and defining views that transform data cubes to add, remove, or modify dimensions in some way. High-level operations for data exploration that wrap several basic operations are also included. For example, users can load arbitrary semi-structured data into a data cube and analyze possible schema changes or functional dependencies.

2.2 Frontend

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

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

2.3 Backend

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

The current Sudokube backend is single-node and loads all the cuboids of a data cube into the main memory before it can be queried. However, a future version of Sudokube can have a distributed backend and a caching policy that determines what cuboids are kept in main memory while keeping the rest on disk. Performance optimizations including just-in-time compilation and CPU vectorization for projection operations are also planned.

2.4 Materialization Strategy

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

2.5 Query Approximation

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

During the fetch phase, the specified cuboids are projected by the backend and fetched as described by the plan. Finally, the fetched cuboids are fed into the *solver* during the solve phase. Depending on their needs, users can choose from several solvers offered by Sudokube. First, the *naive solver* gives the exact result for any query by projecting the smallest materialized cuboid that subsumes it. However, in practice, this subsuming cuboid is almost always the base cuboid for which projection may take a long time. Next, we have the *linear programming solver* that constructs linear equations on query result variables from the fetched cuboids and outputs lower and upper bounds for each variable. While these bounds are guaranteed to be correct, they can be quite lax, and their computation does not scale well to higher-dimensional queries. Finally, we have two *approximate* query solvers. The *moment solver* [1] extracts (stochastic) moments [14] that capture inter-dependencies between the query dimensions from the fetched cuboids using a process similar to the Fourier transform. It then extrapolates them by assuming uncorrelatedness for query dimensions with unknown interaction. Additional heuristics are employed to counteract cases where the assumption is infeasible. This solver yields query results very quickly but is less accurate, particularly for high-dimensional queries. The *graphical model solver* uses iterative proportional fitting [4, 13] to find the maximum entropy query result subject to the constraints imposed by the fetched cuboids. It starts with a uniform distribution of data and iteratively scales the data to fit the fetched cuboids until convergence. This yields more accurate results but takes more time than the moment solver.

Example 2.1. Consider a query that sums the measure values grouped by three binary dimensions. Each of the three 2-D projections of this query yields four linear constraints with sums of two measure values for fixed keys of the dimension pair. Alternatively, the three 2-D projections capture dependencies between any two dimensions, treated as random variables, in the form of three covariances. The linear programming solver works with the linear constraints in the standard way. The graphical model solver reconstructs the maximum entropy distribution subject to the constraints.

The moment solver makes use of the fact that the query result can be exactly reconstructed from its eight moments [14]. For this, it assumes that unknown moments are zero, such as the "generalized covariance" of all three dimensions in this example.

Furthermore, Sudokube supports both *online* and *batch* modes for querying. In batch mode, the base cuboid is never fetched, and the solve phase starts after the fetch phase ends. The final (approximate) query result is then returned to the frontend for decoding and displaying to the user. In online mode, however, the fetch and the solve phases are concurrent, and the query result is updated and displayed periodically using a callback function as more and more cuboids are fetched.

3 USE CASE SCENARIOS

We showcase the capabilities of Sudokube through two scenarios that illustrate its key features. The first scenario lets users explore the different materialization strategies and approximation techniques available in Sudokube for typical OLAP workloads. The second scenario demonstrates how Sudokube's support for high-dimensional data can facilitate tasks such as data cleaning that typically precede data cube construction.

Scenario 1: Basic OLAP Queries using Dimension Hierarchies. This scenario models a typical OLAP workload in Sudokube and demonstrates the entire workflow in Figure 2, from cuboid materialization to running exploratory queries and visualizing the results.

Users begin by selecting one of several datasets with hierarchical dimensions whose structure is known beforehand, such as sales data from an online retailer or flight record data. They can edit the data and add or remove columns as needed. Next, users can define a static schema to load the data into Sudokube and build a data cube by either selecting a materialization strategy or manually specifying which cuboids to materialize.

Once the data cube is built, users can interact with it using the visual interface shown in Figure 3. For example, with the sales dataset, users can generate queries that display total sales per country over time and later drill down to cities or roll up to continents. Users can specify the solver used for generating query results. Sudokube generates a plot, displaying the exact result, an approximation, or bounds, depending on the selected solver. Users can explore the differences in query execution time and result quality of different solvers on other larger data cubes that have been pre-built offline.

Scenario 2: Data Cleaning and Transformations on Streams. This scenario demonstrates how Sudokube can assist users in designing data cleaning and transformation scripts. It is based on large-scale event logging systems, such as those used in data center operation management, where a large number of software packages produce events and frequently get updated. Managing the schema of such event log streams can be challenging while a relational update stream is loaded into the data cube, particularly when columns are renamed without notice.

Consider a dataset where some columns were renamed after a certain number of tuples. Loading this dataset into Sudokube using a dynamic schema causes loaded tuples to have dimensions defined for both the old and the new column names, with NULL values for missing column. Even if the schema changes multiple times, Sudokube can handle all the additional dimensions easily.

With Sudokube's querying and data exploration facilities, users can interactively find timestamps where earlier tuples have only NULL values for the new columns, while newer tuples have only NULL values for old columns. These timestamps could suggest to the ETL designer that a schema change occurred, and that the ETL script should be updated. After discovering renamed columns, users can instruct Sudokube to create a view that combines them and use it to ask queries involving the combined dimensions. Without a tool like Sudokube, discovering and handling such schema changes would be challenging.

REFERENCES

[1] Sachin Basil John and Christoph Koch. 2022. High-dimensional Data Cubes. In *Proceedings of the VLDB Endowment*, Vol. 15. 3828–3840.
[2] Surajit Chaudhuri and Umeshwar Dayal. 1997. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Record* 26, 1 (1997), 65–74.
[3] Surajit Chaudhuri, Umeshwar Dayal, and Vivek Narasayya. 2011. An overview of business intelligence technology. 54, 8 (aug 2011), 88–98.
[4] W. Edwards Deming and Frederick F. Stephan. 1940. On a least squares adjustment of a sampled frequency table when the expected marginal totals are known. *The Annals of Mathematical Statistics* 11, 4 (1940), 427–444.
[5] 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.
[6] 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.
[7] 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.
[8] David C. Hoaglin, Frederick Mosteller, and John W. Tukey (Eds.). 2006. *Exploring Data Tables, Trends, and Shapes.* John Wiley & Sons.
[9] 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.
[10] 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.
[11] 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.
[12] Jose Juan Montes. 2016. CubesViewer - OLAP Visual Viewer and Explore Tool. Retrieved January 13, 2023 from http://www.cubesviewer.com
[13] Yee Whye Teh and Max Welling. 2003. On Improving the Efficiency of the Iterative Proportional Fitting Procedure. In *International Workshop on Artificial Intelligence and Statistics.* PMLR, 262–269. https://proceedings.mlr.press/r4/teh03a.html
[14] Jozef L Teugels. 1990. Some representations of the multivariate Bernoulli and binomial distributions. *Journal of Multivariate Analysis* 32, 2 (1990), 256–268.