Portable In-Browser Data Cube Exploration

Kareem El Gebaly, Lukasz Golab, and Jimmy Lin
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
[kareem.elgebaly,lgolab,jimmylin]@uwaterloo.ca

ABSTRACT

Data cubes, which summarize data across multiple dimensions, have been a staple of On Line Analytical Processing (OLAP) for well over a decade. While users typically access data cubes through data warehouse systems or business intelligence tools, we demonstrate that data cubes can be explored effectively and efficiently *inside a browser*. We provide an overview of the two recent technologies that enable our portable data cube exploration approach: 1) Afterburner, an in-browser relational database management system, and 2) explanation tables, an information-theoretic technique for guided data cube exploration.

1 INTRODUCTION

Since their introduction [6], data cubes have become a staple of On Line Analytical Processing (OLAP) and decision support. Given a dataset with multiple *dimension attributes* and one or more *measure attributes*, data cubes compute aggregate functions of the measure attribute over all subsets of the dimension attributes. Users typically explore data cubes by selecting different subsets of dimension attributes and viewing the resulting aggregates: e.g., total sales by store, total sales by product type, total sales by day, total sales by store and product type, etc.

Data cubes may be very large; e.g., millions of distinct products, multiplied by hundreds of stores, multiplied by hundreds of days, etc. Typically, data warehouse systems and business intelligence tools allow users to "start small" and zoom in (i.e., drill down) to different dimensions; e.g., a user may start by viewing total sales and then view a breakdown of sales by store.

We ask the following question: *can data cube exploration be performed effectively and efficiently inside a browser?* There are several compelling reasons for doing this. As evidenced by tools such as Jupyter Notebook, which integrate code, output, and visualization, the browser is no longer a dumb rendering endpoint and has become the de-facto front end for data science applications. It is therefore reasonable to ask if the browser can also eliminate the need to maintain a local data management system or to obtain access to a remote database server, at least for some types of tasks and users. Furthermore, in keeping with the recent trend of *data democratization*, in-browser analytics can facilitate *data analysis democratization* as a cross-platform, easy-to-share (across users and organizations) data analysis framework for non-expert users such as journalists.

In this paper, we show that the answer to the above question is yes, at least for moderately-sized datasets that fit in the browser's memory. We demonstrate our portable in-browser data cube exploration tool and explain its design, which leverages two recent technologies:

(1) **Afterburner:** an in-browser relational database management system (RDBMS) recently demonstrated at the SIGMOD conference [4]. Afterburner is implemented in JavaScript and runs inside a browser with no external dependencies, taking advantage of column-oriented storage using typed arrays and query compilation into asm.js, a strictly-typed and easy to optimize subset of JavaScript. On modestly-sized datasets, the performance of Afterburner was shown to be similar to that of the columnar RDBMS MonetDB [4] on the well-known TPC-H benchmark.
(2) **Explanation tables:** an information-theoretic technique for explaining a measure attribute using combinations of dimension attributes [3], which, as we will demonstrate, provides a useful starting point for interactive data cube exploration.

The remainder of this paper is organized as follows: Section 2 gives an overview of data cubes and the information-theoretic cube exploration framework we use. In Section 3, we explain the implementation details of our data cube exploration tool built on top of Afterburner. Section 4 presents an outline of our demonstration, Section 5 discusses related work, and Section 6 concludes the paper with directions for future work.

2 DATA CUBE EXPLORATION

2.1 Illustrative Example

We illustrate data cubes and their exploration with a simple example. Suppose we have collected a dataset from smart refrigerators indicating which food items were eaten while they were still fresh, before their expiry dates, and which ones were expired and had to be thrown away. The dataset is shown in Table 1. Each row consists of a numeric `id` which serves as a key but is not relevant for this example, and the following three dimension attributes: an item `name`, the `season` when the item was stored, and the `location` of the refrigerator. Additionally, the binary measure attribute `expires` identifies items which were expired (in general, measure attributes can be numeric).

Suppose we want to understand the reasons why some food items are consumed and some expire: is it the item type, the season, the location, or some combination of these? To do so, we compute a data cube over Table 1, as shown in Table 2. The two aggregate functions are count and average of the `expires` values, denoting how likely a subset of items is to expire. The first

IDEA'17, August 14th, 2017, Halifax, Nova Scotia, Canada

Table 1: Example dataset

id	item	season	location	expires?
1	Cheese	Winter	Kitchen	0
2	Cherries	Summer	Summer house	1
3	Chocolate	Summer	Summer house	0
4	Chocolate	Spring	Bedroom	0
5	Chocolate	Winter	Office	0
6	Chocolate	Summer	Basement	0
7	Chocolate	Fall	Winter house	0
8	Eggs	Fall	Kitchen	1
9	Eggs	Winter	Winter house	1
10	Juice	Spring	Office	0
11	Milk	Spring	Office	1
12	Milk	Summer	Winter house	1
13	Veggies	Spring	Summer house	1
14	Veggies	Winter	Winter house	1

Table 2: Fragments of a data cube over the example dataset

id	item	season	location	count	AVG(exp.)
1	*	*	*	14	0.5
2	Cheese	*	*	1	0
3	Cherries	*	*	1	1
9	*	Winter	*	4	0.5
10	*	Summer	*	4	0.5
13	*	*	Kitchen	2	0.5
14	*	*	Bedroom	1	0
19	Cheese	Winter	*	1	0
20	Chocolate	Summer	*	2	0
32	Cheese	*	Kitchen	1	0
33	Eggs	*	Kitchen	1	0
46	*	Winter	Kitchen	1	0
47	*	Spring	Office	2	0.5
57	Cheese	Winter	Kitchen	1	0
58	Chocolate	Spring	Bedroom	1	0

row of the data cube corresponds to the values of the two aggregates over the entire dataset, with stars denoting all possible values. Row ids 2 and 3 correspond to an SQL `GROUP BY` query over the `item` column; row ids 9 and 10 to `GROUP BY season`; row ids 13 and 14 to `GROUP BY location`; row ids 19 and 20 to `GROUP BY item, season`, and so on. The entire data cube consists of 70 rows and corresponds to a union of aggregation queries over all possible subsets of dimension attributes.

Two fundamental data cube operations are *drill down* and *roll up*, corresponding to adding or removing a dimension attribute, respectively. For example, starting from row id 1, we can drill down into the `item` dimension attribute and obtain more details about different items. Conversely, starting, say, from row id 19, we can roll up the `item` attribute, leaving only `season` (row ids 9 and 10).

2.2 Explanation Tables

Since data cubes may be very large, a useful data exploration technique is to identify interesting or informative parts of a data cube

Table 3: An explanation table over the example dataset

item	season	location	count	AVG(expires)
*	*	*	14	0.5
Chocolate	*	*	5	0
*	*	Winter house	4	0.75
*	*	Summer house	3	0.67

that users should examine. One such technique is the *explanation table* [3], which identifies rows from the data cube that provide the most information about the distribution of the measure attribute.

Table 3 shows an explanation table over our example dataset. Each row of an explanation table is a row from the data cube and includes the aggregates computed over the measure attributes; from now on, we refer to explanation table rows as *patterns*. The first pattern states that, on average, half of the items in the dataset have expired. The second pattern states that none of the five chocolates have expired and is included in the explanation table because it provides the most additional information about the distribution of the measure attribute. Intuitively, this is because chocolates are far less likely to expire than other products, and there are sufficiently many chocolates in the dataset. The third and fourth patterns, respectively, indicate that items located in the winter house and the summer house are more likely to expire. Again, they are included because they provide the most additional information about the distribution of the measure attribute.

An explanation table provides a useful starting point for data cube exploration: each of its patterns is *informative* and may be explored further by the user for additional insight. For example, the second pattern in Table 3 reveals that chocolates tend not to expire. The user may then drill down by `season` and/or `location` to see if chocolates purchased in different seasons or stored in different locations are more or less likely to expire. This is exactly how we leverage explanation tables in our data cube exploration tool.

2.3 Constructing Explanation Tables

We now give a brief overview of the algorithm for constructing informative explanation tables; see El Gebaly et al. [3] for full details.

The idea is to maintain *maximum-entropy* estimates for the values of the measure attribute and refine them as new patterns are added to the explanation table. In each iteration of the algorithm, we greedily add a pattern to the explanation table that gives the greatest information gain, more precisely, the greatest reduction in the *Kullback-Leibler Divergence* between the actual measure values and the estimated ones. We stop after k iterations, where k is a user-supplied parameter, yielding an explanation table with k patterns.

Consider Table 3. Based on the first pattern (the all-stars pattern), the maximum-entropy estimate of the true `expires` values is to set each value (of each of the 14 rows in Table 1) to 0.5. Of course, the actual `expires` values are binary, but we allow the estimates to be real numbers between zero and one. This maximum-entropy estimate only uses the information implied by the first pattern of the explanation table, which is that `AVG(expires)=0.5`, without making any other assumptions.

As it turns out, the pattern (Chocolate,*,*) is then added to the explanation table because it provides the greatest information gain. As a result, we update the estimates of `expires` as follows. Since the second pattern implies that all chocolates have `expires`=0, we set the estimates for rows 3 through 7 to zero. Next, for consistency with the first pattern, which requires the average value of `expires` over the whole dataset to be 0.5, we must set the estimates for all non-chocolate rows to $\frac{7}{9}$ each. This gives us a maximum-entropy estimate that only considers the information contained in the first two patterns of the explanation table. As in El Gebaly et al. [3], we use *iterative scaling* to compute updated estimates whenever a new pattern is added to the explanation table.

To summarize, the greedy algorithm for constructing an explanation table works as follows. We iterate k times, once for each pattern. In each iteration, we 1) compute the information gain of a set of candidate patterns, 2) add to the explanation table the pattern with the greatest gain, and 3) update the maximum-entropy estimates. To compute the information gain in step 1), we build a data cube with the average of the *estimated* measure attribute as the aggregate function, and compare the estimates with the actual values. For efficiency, our implementation uses sampling to compute these data cubes, following El Gebaly et al. [3].

3 IN-BROWSER IMPLEMENTATION

At a high level, our data cube exploration tool issues and consumes the output of SQL queries executed by the Afterburner RDBMS. Both explanation table construction and subsequent drill-down into individual patterns are accomplished with a series of SQL group-by/aggregation queries. Our implementation handles interactive data cube exploration of moderately-sized datasets (around 10 million records) in sub-second time.

Afterburner (and our tool running on top of it) is implemented as a JavaScript library and runs completely stand-alone inside a browser. Datasets are loaded from the local file system or from a remote server. Once loaded, data are immutable and stored in memory in column-oriented format. As discussed below, Afterburner exploits two JavaScript features: typed arrays for memory-efficient storage and asm.js for fast compiled queries.

3.1 Columnar Storage Using Typed Arrays

Array objects in JavaScript can store elements of any type and are not arrays in a traditional sense (compared to say, C) since consecutive elements may not be contiguous; furthermore, the array itself can dynamically grow and shrink. This flexibility limits the optimizations that the JavaScript engine can perform both during compilation and at runtime. In contrast, typed arrays in JavaScript are comprised of buffers, which simply represent untyped binary data, and views, which impose a read context on the buffer.

Typed arrays allow the developer to create multiple views over the same buffer. Afterburner takes advantage of this feature to pack relational data into a columnar layout. In our implementation, each column is laid out end-to-end in the underlying buffer, which can be traversed with a view of the corresponding type. The table itself is a group of pointers to the offsets of the beginning of the data in each column. Intermediate data for query execution are also stored using typed arrays.

3.2 Query Compilation into Asm.js

In conjunction with typed arrays, Afterburner takes advantage of asm.js, a strictly-typed subset of JavaScript that is designed to be easily optimizable by an execution engine. Any JavaScript function can request validation of a block of code as valid asm.js via a special prologue directive, `use asm`, which happens when the source code is loaded. Validated asm.js code (typically referred to as an asm.js module) is amenable to ahead-of-time (AOT) compilation, in contrast to just-in-time (JIT) compilation in vanilla JavaScript. Executable code generated by AOT compilers can be quite efficient, through the removal of runtime type checks (since everything is statically typed), operation on unboxed (i.e., primitive) types, and the removal of garbage collection.

Afterburner translates SQL queries into the string representation of an asm.js module (i.e., the physical query plan), calls `eval` on the code, which triggers AOT compilation and links the module to the calling JavaScript code, and finally executes the module (i.e., executes the query plan). The typed array storing all the tables is passed into the module as a parameter, and the query results are returned by the module.

3.3 Query Operators and Materialization

Supported SQL operators include selection/filters, aggregates, group by (using hashing) and joins (also using hashing). Notably, Afterburner avoids materialization of intermediate results as much as possible in order to fit inside a web browser memory. For example, we only store record identifiers in hash tables instead of copying the values in order to minimize the memory footprint of the hash-based operators such as joins and group bys.

For data cube exploration, we need to materialize fragments of the data cube. To reduce the memory footprint, we use dictionary encoding for dimension attribute values.

4 DEMONSTRATION SCENARIOS

In our demonstration, participants will create explanation tables to help guide their data cube exploration. We will prepare several real-world datasets for the demonstration, including airline upgrades (where the goal will be to understand what makes a passenger more likely to have their seat upgraded to first class) and U.S. census data (where the goal will be to understand what makes a person more likely to earn an annual income over $50,000). Most of our demonstration datasets are downloaded from the UCI archive.[1]

Figure 1 shows a screenshot of an explanation table with $k = 5$ patterns over the census dataset. The dimension attributes include workclass, education level, marital status, occupation, relationship to the head of the household, race, sex, and country. `COUNT(*)` refers to the number of rows in the dataset covered by a pattern and `AVG(p)` is the average value of the binary indicator attribute whose value is one if the income exceeds $50,000. The column labeled `distribution` visualizes the proportion of the (binary) measure attribute values that are one (in green) vs. zero (in red).

Clicking on the `explore` link at the end of each pattern allows users to drill into the rows captured by the pattern. For example, a user may want to learn more about the third pattern, which indicates

[1] http://archive.ics.uci.edu/ml/datasets/

Explanation Table

workclass	education	status	occupation	relationship	race	sex	country	COUNT(*)	AVG(p)	distribution	Explore
*	*	*	*	*	*	*	*	32561	0.24		explore
*	*	Never-married	*	*	*	*	*	10683	0.05		explore
*	Bachelors	Married-civ-spouse	*	*	*	*	*	2768	0.67		explore
*	*	Divorced	*	*	*	*	*	4443	0.10		explore
*	*	*	Other-service	*	*	*	*	3295	0.04		explore

Figure 1: A five-pattern explanation table over the U.S. census dataset.

Control Exploration

workclass	education	status	occupation	relationship	race	sex	country
ALL	Bachelors	Married-civ-spouse	*	*	*	*	*

Figure 2: Drilling into a pattern.

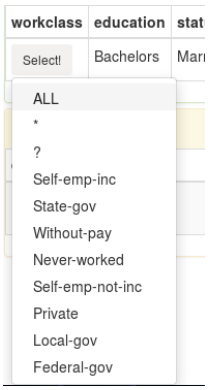

Figure 3: Manually drilling into a pattern.

that married people with a Bachelor's degree are more likely to earn a high salary. Figure 2 shows a screenshot of the corresponding exploration panel. The user can select one or more attributes to drill into. In the figure, `workclass=ALL` will compute separate aggregates for married people with a Bachelor's degree and for each workclass. Rather than selecting `ALL`, the user can also specify selected values of interest of additional attributes such as workclass. For example, Figure 3 shows a dropdown menu with all the values of workclass in the dataset.

Figure 4 shows the results of drilling into the third pattern, as specified in Figure 2. The user can now explore the distribution of the measure attribute for each workclass of married people with a Bachelor's degree.

Exploration

workclass	education	status	count(*)	avg(p)	distribution	explore
Self-emp-not-inc	Bachelors	Married-civ-spouse	265	0.51		explore
Private	Bachelors	Married-civ-spouse	1747	0.70		explore
Local-gov	Bachelors	Married-civ-spouse	220	0.62		explore
Federal-gov	Bachelors	Married-civ-spouse	105	0.72		explore
?	Bachelors	Married-civ-spouse	94	0.44		explore
Self-emp-inc	Bachelors	Married-civ-spouse	210	0.76		explore
State-gov	Bachelors	Married-civ-spouse	127	0.65		explore

Figure 4: A drill-down of the different patterns.

5 RELATED WORK

This paper is related to two bodies of work: data cube exploration and query plan compilation.

As we explained earlier, we use explanation tables for guided data cube exploration. The idea of information-theoretic data summarization initially appeared in Sarawagi et al. [13] and was then expressed in the form of explanation tables in subsequent work [3, 5]. We use the technique from El Gebaly et al. [3] in our demonstration. We note that there are other data cube exploration techniques such as smart drill-down [7], which can be added to future versions of our tool. Beyond data cubes, there is a wide variety of data exploration, data explanation, and outlier detection approaches (see, e.g., [1, 2, 11]), and it will be interesting to study whether they can be implemented efficiently in our in-browser framework.

Afterburner is based on the compiled query approach to query execution, similar to systems such as HIQUE [10], LegoBase [9, 14], Proteus [8], and HyPer [12]. Our query compilation techniques are relatively standard, with the exception of targeting JavaScript and using query plans that fit into a browser's limited memory.

6 CONCLUSIONS

In this paper, we motivated and described our tool for in-browser data cube exploration. We take advantage of modern browsers, which have become much more than dumb rendering endpoints, to provide a cross-platform, maintenance-free solution for exploring small to medium datasets. We believe that our approach is useful to "amateur data scientists" and non-expert users. In future work, we plan to enhance the effectiveness of our tool by including new data exploration techniques, and improve its efficiency by studying new optimizations within Afterburner itself.

REFERENCES

[1] Azza Abouzied, Joseph M. Hellerstein, and Avi Silberschatz. 2012. Playful Query Specification with DataPlay. *PVLDB* 5, 12 (2012), 1938–1941.

[2] Peter Bailis, Edward Gan, Kexin Rong, and Sahaana Suri. 2017. Demonstration: MacroBase, A Fast Data Analysis Engine. In *SIGMOD*. 1699–1702.

[3] Kareem El Gebaly, Parag Agrawal, Lukasz Golab, Flip Korn, and Divesh Srivastava. 2014. Interpretable and Informative Explanations of Outcomes. *PVLDB* 8, 1 (2014), 61–72.

[4] Kareem El Gebaly and Jimmy Lin. 2017. In-Browser Interactive SQL Analytics with Afterburner. In *SIGMOD*. 1623–1626.

[5] Guoyao Feng, Lukasz Golab, and Divesh Srivastava. 2017. Scalable Informative Rule Mining. In *ICDE*. 437–448.

[6] Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh. 1996. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total. In *ICDE*. 152–159.

[7] Manas Joglekar, Hector Garcia-Molina, and Aditya G. Parameswaran. 2016. Interactive Data Exploration with Smart Drill-Down. In *ICDE*. 906–917.

[8] Manos Karpathiotakis, Ioannis Alagiannis, and Anastasia Ailamaki. 2016. Fast Queries Over Heterogeneous Data Through Engine Customization. *PVLDB* 9, 12 (2016), 972–983.

[9] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. 2014. Building Efficient Query Engines in a High-level Language. *PVLDB* 7, 10 (2014), 853–864.

[10] Konstantinos Krikellas, Stratis Viglas, and Marcelo Cintra. 2010. Generating Code for Holistic Query Evaluation. In *ICDE*. 613–624.

[11] Arnab Nandi. 2013. Querying Without Keyboards. In *CIDR*.

[12] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB* 4, 9 (2011), 539–550.

[13] Sunita Sarawagi. 2000. User-Adaptive Exploration of Multidimensional Data. In *VLDB*. 307–316.

[14] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. 2016. How to Architect a Query Compiler. In *SIGMOD*. 1907–1922.