Analytics for Everyone

by

Kareem El Gebaly

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2018

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	**Daniel Lemire**
	Professor, Department of Science and Technology
	Université du Québec (TÉLUQ)
Supervisors:	**Jimmy Lin**
	Professor, David R. Cheriton School of Computer Science
	University of Waterloo
	Lukasz Golab
	Associate Professor, Department of Management Sciences
	University of Waterloo
	Ashraf Aboulnaga
	Adjunct Associate Professor, David R. Cheriton School of Computer Science
	University of Waterloo
Internal Members:	**Ken Salem**
	Professor, David R. Cheriton School of Computer Science
	Bernard Wong
	Associate Professor, David R. Cheriton School of Computer Science
	University of Waterloo
Internal-External Member:	**Wayne Oldford**
	Professor, Department of Statistics and Actuarial Science
	University of Waterloo

Abstract

Analyzing relational data typically involves tasks that facilitate gaining familiarity or insights and coming up with findings or conclusions based on the data. This process is usually practiced by data experts, such as data scientists, who share their output with a potentially less expert audience (everyone). Our goal is to enable everyone to participate in analyzing data rather than passively consuming its outputs (analytics democratization). With today's increasing availability of data (data democratization) on the internet (web) combined with already widespread personal computing capabilities such a goal is becoming more attainable. With the recent increase of public data, i.e., *Open Data*, users without a technical background are keener than ever to analyze new data sets that are relevant to wide sectors of society. An important example of Open Data is the data released by governments all over the world, i.e., *Open Government*.

This dissertation focuses on two main challenges that would face data exploration scenarios such as exploring open data found over the web. First, the infrastructure necessary for interactive data exploration is costly and hard to manage, especially by users who do not have technical knowledge. Second, the target users need guidance through the data exploration since there are too many starting points.

To eliminate challenges related to managing infrastructure, we propose an in-browser SQL engine (serverless), i.e., a portable database, which we call Afterburner. Afterburner achieves comparable performance to native SQL engines given the same resources on modestly sized data sets. Afterburner uses code generation techniques that target an optimization-amenable subset of JavaScript and employs typed arrays for its columnar-based in-memory storage. In addition, for databases that are too large for the browser, we propose a hybrid architecture to accelerate the performance of data exploration tasks: a one-time SQL query that runs at the backend and SQL queries running in the browser as per user's interactions. Based on a simple hint by the user, Afterburner automatically splits queries into two parts: a backend query that generates a materialized view that is shipped to the browser, and a frontend query per subsequent interaction occur locally against this view. Optimizing queries using local materialized views inside the browser accelerates query latency without adding any complexity to the backend or the frontend.

One common theme among many data exploration tasks revolves around navigating the many different ways to group the data, i.e., exploring the data cube. Thus, to guide the user through data exploration, we apply an information-theoretic technique that picks the most informative parts from the entire data cube of a relational table, which is called Explanation Tables. We evaluate the efficiency and effectiveness of a sampling-based technique for generating explanation tables that achieves comparable quality to an exhaustive technique that considers the entire data cube, with a significant reduction in the run time. In addition, we introduce optimizations to explanation tables to fit the modest resources available in the browser without any external dependencies.

In this, we present an SQL engine and a data exploration guidance tool that run entirely in the browser. We view the techniques and the experiments presented here as a fully functional and open-sourced proof of viability of our proposal. Our analytical stack is portable and works entirely in the browser. We show that SQL and exploration guidance can be as accessible as a web page, which opens the opportunity for more people to analyze data sets. Facilitating data exploration for everyone is one step closer towards analytics democratization where everyone can participate in data exploration, not just the experts.

Acknowledgements

I would like to express my gratitude and appreciation towards my advisors: Jimmy Lin, Lukasz Golab, and Ashraf Aboulnaga.

I would like to extend my gratitude to Jimmy my advisor and my mentor. Jimmy taught me: how to find research ideas that I am passionate about, how to develop interesting research questions, and how to present my work in papers and talks. While working with Jimmy, I learned that work ethic and dedication can co-exist with fun and passion; my work was the beneficiary of such fusion. I only hope that through osmosis qualities that I admire a lot, such as chutzpah, rigor, and vibrancy sneaked to my ways of working, thinking, and living.

I would like to thank Ashraf for his patience, guidance, and generosity. Throughout my Ph.D. career, Ashraf gave me a lot of freedom to pick my research topics. No matter how unconventional my initial ideas sounded like, Ashraf generously and patiently trusted in my work.

I would like to thank Lukasz my academic advisor for his guidance and teachings. Lukasz introduced me to new research areas and guided me through them. I would like to thank, as well, Parag Agrawal, Filip Korn, and Divesh Srivastava, with whom I was privileged to collaborate on Explanation Tables. Such collaboration was a great learning experience that I have personally enjoyed a lot.

Finally, I would like to thank my family and friends that have supported and encouraged me through this.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Analytics for Everyone

The process of analyzing relational data typically involves tasks that facilitate gaining familiarity or insights and coming up with findings or conclusions based on the data. Data analysis requires a certain level of technical competency. Thus, it is usually performed by data experts, such as data scientists or data analysts. In many cases, data experts analyze the data then share their findings and conclusions with their potentially less technical audience (everyone). Our goal is to enable everyone to analyze the data instead of relying solely on experts. This can be called the *democratization of analytics*.

While Relational Database Management Systems (RDBMS) and data analysis platforms, such as Spark alleviate much of the complexities associated with analytics, these tools require specialized technical skills. Knowing SQL, familiarity with certain programming languages, and management or access to infrastructure are all examples of computer science skills that may be required to analyze data.

With the recent increase of publicly available data, i.e., *Open Data*, non-technical users are keener than ever to analyze new data sets that are relevant to wide sectors of society. Thus, our work targets people that do not have computer science background yet want to analyze data. An important example of open data is data released by governments all over the world, i.e., *Open Government*. This dissertation focuses on two main challenges that face everyone, who want to either analyze or share their data exploration tasks over the web.

- First, the infrastructure necessary for interactive data exploration is costly and hard to manage, especially for non-technical users.

- Second, there are too many starting points for data exploration tasks, since the number of possible ways to query a data set is usually large. Thus, users usually need the guidance where to start analyzing the data.

To deal with the first challenge related to having access to tools and managing infrastructure, we propose an in-browser SQL engine (Afterburner), i.e., a portable SQL engine that does not require a backend (serverless). In addition, for databases that are too large for the browser, we propose hybrid architecture: a onetime SQL query that runs at the backend and SQL queries running in the browser as per users' interactions.

One common theme among many data exploration tasks revolves around navigating the many different ways to group the data, i.e., exploring the data cube. Thus, to deal with the second challenge, we apply an information-theoretic technique that picks the most informative parts from the entire data cube of a relational table (*Explanation Tables*). We evaluate the efficiency and effectiveness of a sampling-based technique for generating explanation tables that achieves comparable quality to an exhaustive technique that considers the entire data cube, with a significant reduction in the run time. The intuition is that the most informative parts of the data cube are likely to be important or interesting starting points in data exploration tasks. In addition, we introduce optimizations that allow for creating Explanation Tables under the modest resources available in the browser, again, without any external dependencies. With a portable SQL engine and a data exploration tool on top of it, we tackle the two main identified challenges.

In this, we present an SQL engine and a data exploration guidance tool that run entirely in the browser. We view the techniques and the experiments presented here as a fully functional and open-sourced proof of viability of our proposal. Our analytical stack is portable and works entirely in the browser. We show that SQL and exploration guidance can be as accessible as a web page, which opens opportunities for more people to analyze data sets. Facilitating data exploration for everyone is one step closer towards analytics democratization where everyone can participate in data exploration, not just the experts.

1.2 From Data Democratization to Analytics Democratization

1.2.1 Exploring Open Data

A recent trend, often referred to as data democratization, encourages releasing data on the web to be accessed by everyone. Open data movements, such as the *Open Government* aims for benefiting the society and governments through releasing data on the web [41]. Open Government is led by example by several countries, such as Canada,[1] Australia,[2] the United Kingdom,[3] and the United States of America.[4] In addition to the open government, intergovernmental organizations such as the United Nations[5] and the World Bank [8] are also releasing data on the web for similar purposes. Open data is not limited to governments since non-governmental organization such as scientific and research institutes are also taking part and releasing data on the web. Releasing the data on the web is only the first step towards full data democratization. The full potential of open data is yet to be realized when everyone can readily analyze it, whether on their own or through pre-prepared exploration tasks. Hence, the goal of this dissertation is analytics for everyone.

1.2.2 The OpenAid Use Case

Analytics democratization essentially requires both: access to the data (data democracy) and the ability to analyze it. While on one hand, data democracy is currently striding forward and more data is released openly every day [29], on the other hand, data analytics is far from being mainstream yet and is only performed by data experts. Reasons behind the inherent challenges of analytics tasks include: requiring a deep stack of tools and infrastructure and a wide range of expertise. Analytics democratization has proven to be an essential part of data democratization, i.e., the impact of data democracy is tied to the ability of the citizens to interactively analyze the data. The OpenAid project[6] launched in 2011 by the government of Sweden is a project that targets combating poverty all over the world. This governmental aid project is an open data project since Sweden releases detailed information such as the contributing government agencies, the date when the contributions were made, the receiving countries, and the receiving agencies. The main motivations for releasing this data are ensuring transparency and engaging the Swedish citizens with the aid program led by their government.

In the early stages of the OpenAid project, data was released on the web in a static tabular form, i.e., the data set was made available for download and web-based browsing. In addition to the static data, high-level insights about the data were made available on the web. While this abides by the open data principles—since the data is freely available online—a subsequent study revealed that the data had less impact since fewer citizens engaged with the project—in comparison with later stages of the project. A case study [5] cites two main challenges to citizen engagement at the early stages of the project. First, the lack of interactivity, i.e., the data was

[1]http://open.canada.ca/en/open-data/
[2]http://data.gov.au/
[3]http://www.opengovernment.org.uk/
[4]https://www.state.gov/open/
[5]http://data.un.org/
[6]http://odimpact.org/case-openaid-in-sweden.html

Ministry	Name	PositionTitle	PositionClass	Salary	Year	CashBenefits	NonCashBenefits	Severance
Health	DavidsonJanet	Deputy Minister	Senior Official	2014	577778	83259	9647	0
Health	Janet Davidson	DM Executive Council	Senior Official	2015	457661	81487	4914	0
Justice and Solicitor General	Brooks-Lim Elizabeth	Deputy Chief Med Examiner	Deputy Chief ME	2016	383143	21922	6790	0
Justice and Solicitor General	Jones Tera	Assist Chief Medical Examiner	Pathologist IV	2016	383143	0	5752	0
Justice and Solicitor General	Adeagbo Bamidele	Assist Chief Medical Examiner	Pathologist IV	2016	383143	0	7092	0

Table 1.1: A few records from the Alberta salary disclosure data set (`Salary`).

published without any readily available means of analyzing it such as interactive exploration or visualization. Second, the insights were not readily interpretable by non-experts. The OpenAid use case is an example of how making data available may not be sufficient for engaging the citizens and that analytics democracy can significantly increase the impact of open data. Further research agrees with this observation, suggesting that the readiness of the data to be analyzed is a key challenge to the impact of open data [54]. Thus, the ability to make the best out of the data is related to how much the citizens can analyze it.

In the next section, we describe a data exploration task that requires knowing SQL and access to data analysis tools. The data exploration task is based on data made available by the Canadian government. While we expect data experts, such as data scientists to have the know-how of exploring the data, we do not expect everyone to have such know-how. As we will discuss, the example data exploration task highlights the main challenges that would face non-technical users, such as concerned citizens that find open data sets over the internet. We also outline how our proposed in-browser SQL engine and our portable data exploration tool can address these challenges.

1.2.3 Example: Data Exploration Task — Public Salaries in Alberta

The Canadian province of Alberta makes available data sets over the web through its open government web portal. Again, the main goal is to transparently inform the citizens about the details of governmental activities and to encourage the open engagement based on transparent data. The available data sets include information about various governmental activities, such as salaries or expenses. Table 1.1 shows the dimension attributes and a few example records the *Public Disclosure of Salary and Severance* data set (`Salary`).[7] Each record of the data set refers to a public servant and contains information such as their name, their ministry, their job title, and their job class. Each record also reveals the exact amount of compensation paid to the public servant such as the base salary, cash benefits, and severance. As discussed in the previous section the goal is to go beyond just making the data available and allowing everyone to analyze

[7] `http://open.alberta.ca/opendata/public-disclosure-of-salary-and-severance`

Ministry	**Year**	**PositionTitle**	**PositionClass**	AVG(HighSalary)	COUNT(*)
*	*	*	*	0.19	18023
Justice and Solicitor General	*	*	*	0.42	3865
Human Services	*	*	*	0.13	2089
Education	*	*	*	0.06	1042
...	...	...	...	...	...
*	2015	*	*	0.20	4050
*	2014	*	*	0.19	3557
*	2016	*	*	0.20	3551
...	...	...	...	...	...
*	*	Crown Prosecutor	*	0.49	783
*	*	Barrister and Solicitor	*	0.58	759
*	*	Manager	*	0.00	279
...	...	...	...	...	...
*	*	*	Senior Manager	0.15	5201
*	*	*	Manager	0.00	2705
*	*	*	Executive Manager 1	0.56	1913
...	...	...	...	...	...
...	...	...	...	...	...
Justice and Solicitor General	2014	Barrister and Solicitor	Senior Manager	0.71	119
Justice and Solicitor General	2014	Crown Prosecutor	Senior Manager	0.73	105
Justice and Solicitor General	2013	Crown Prosecutor	Senior Manager	0.63	101
...	...	...	...	...	...

Table 1.2: The data cube of `HighSalary` from `Salary` data set.

the data on their own to increase its impact. The users can only download or browse the data set via a web-based tabular navigation. The tabular navigation form allows users to filter records according to attributes that match a certain condition (e.g., salaries in a certain range). Thus, similar to the OpenAid project, the Open Government portal makes the data available in tabular format, without any means to analyze it.

We consider a case where a data scientist is interested in understanding which public servants receive high salaries. To start her data analytics task, she loads the data set into a familiar RDBMS that supports SQL. Then she identifies which public servants received a high salary, for instance, a salary greater than 150000 per year. This means she creates a new column called `HighSalary`, which is a binary outcome attribute indicating meeting her criteria, i.e., having a value of 1 if the salary is higher than 150000 and 0 otherwise.

There are many possible explanations for why different public servants received high salaries that include: different ministries offer different salaries; different salaries are paid in different years; similarly different job titles and classes get paid differently than others; certain combinations of these dimension attributes have different salaries. This question can be answered by

Ministry	**Year**	**PositionTitle**	**PositionClass**	AVG(HighSalary)	COUNT(*)
Justice and Solicitor General	2012	*	*	0.37	700
Justice and Solicitor General	2013	*	*	0.39	764
Justice and Solicitor General	2014	*	*	0.41	798
Justice and Solicitor General	2015	*	*	0.47	847
Justice and Solicitor General	2016	*	*	0.47	756

Table 1.3: A drill-down by `Year` for the `Justice and Solicitor General` salaries.

navigating the data cube of the data set. Thus, she considers Table 1.2 that shows the average value of the outcome attribute (`HighSalary`) and their count for all possible combinations of dimension attributes of the public `Salary` data set. The data scientist can use the following SQL query to generate the data cube:

```
SELECT  Ministry, Year, PositionTitle, PositionClass,
        AVG(HighSalary), COUNT(*)
FROM    Salary
CUBE BY Ministry, Year, PositionTitle, PositionClass
```

Rows in the data cube correspond to rows of possible GROUP-BY queries over the data set. In this dissertation, we will refer to the rows of the data cube as *patterns*. The first pattern shows the average salary (AVG(`HighSalary`)) and the number of records in this group (COUNT(*)). We use * to denote all possible values, i.e., the first pattern matches the entire 18023 records in the data set. Subsequent patterns correspond to an SQL query with a GROUP-BY over the `Ministry` attribute, followed by patterns corresponding to GROUP-BY `Year`, then GROUP-BY `PositionTitle`, etc. The entire data cube consists of 80495 patterns and corresponds to the union of aggregation queries over all possible subsets of dimension attributes (i.e., 2^4 GROUP-BY queries for this data set).

The second pattern in Table 1.2 (the data cube) indicates the average value for the outcome attribute (`HighSalary`) for public servants in the `Justice and Solicitor General` ministry is 0.42. This means 42% of the records in this group have a high salary which is more than twice the 19% average of the entire data set as indicated by the first pattern in the data cube. This intrigues the data scientist to look more into this group. Suppose she is interested to know the effect of the `Year` on the average value of `HighSalary` for this group, hoping to discover a trend. Table 1.3, shows the average salaries of the ministry grouped by the different values of the dimension attribute `Year`; this break down by `Year` is often referred to as a data cube *drill-down* operation that corresponds to the following SQL query:

```
SELECT   Ministry, Year, AVG(HighSalary), COUNT(*)
```

```
FROM     Salary
WHERE    Ministry = 'Justice and Solicitor General'
GROUP BY Ministry, Year
ORDER BY Year ASC
```

This drill-down reveals, to some extent, an increasing trend in the average of the outcome attribute (`HighSalary`) for the Justice and Solicitor General public servants; the average values in the years 2015 and 2016 are 6-10% points more than earlier years. This zoom-in like operator can be applied repeatedly since there are two more possible drill-down operations, namely, by `PositionTitle` and by `PositionClass`.

Conversely, the fourth pattern from the bottom shown in Table 1.2 reveals an average of 71% for the `HighSalary` for the Senior Managers, Barrister and Solicitor, in the Justice and Solicitor General ministry, in 2014. The average of the outcome attribute is significantly higher (52% points more) than the average of the entire data set which intrigues the data scientist to want to learn more about the less specific aggregation. Again, the data scientist is interested to know how the average value of `HighSalary` varies by `Year`. Table 1.4 shows the average of the outcome attribute for this group, broken down by `Year`. This zoom-out like data cube operation is often referred to as *roll-up* and corresponds to the following SQL query:

```
SELECT   Ministry, Year, PositionTitle
         AVG(HighSalary), COUNT(*)
FROM     Salary
WHERE    Ministry = 'Justice and Solicitor General'
     AND PositionTitle = 'Barrister and Solicitor'
GROUP BY Ministry, Year, PositionTitle
ORDER BY Year ASC
```

Table 1.4 reveals an increasing trend in the average of `HighSalary` for this group; the average values in the years 2015 and 2016 are at least 7% points more than earlier years. Further roll-ups and drill-down can be applied to navigate the cube. In this example, we have shown the iterative nature of data cube exploration: where users can navigate the different levels of aggregation to get more familiar with a new data set of public interest, reveal new insights, or quench an intriguing question about the data.

In this section, we discussed a data cube exploration task based on an outcome attribute of interest for the open government data set for the public servants in Alberta. The data cube exploration scenario requires knowing SQL and access to an SQL engine. In the next section, we will discuss how our proposed in-browser SQL engine coupled with an intuitive user interface would allow everyone who has access to a web browser to navigate the data cube—without depending on an external data analysis tools.

Ministry	Year	PositionTitle	PositionClass	AVG(HighSalary)	COUNT(*)
Justice and Solicitor General	2012	Barrister and Solicitor	*	0.45	124
Justice and Solicitor General	2013	Barrister and Solicitor	*	0.51	139
Justice and Solicitor General	2014	Barrister and Solicitor	*	0.56	154
Justice and Solicitor General	2015	Barrister and Solicitor	*	0.68	150
Justice and Solicitor General	2016	Barrister and Solicitor	*	0.63	148

Table 1.4: A roll-up followed by a drill-down by `Year` for the salaries in the `Justice and Solicitor General` ministry.

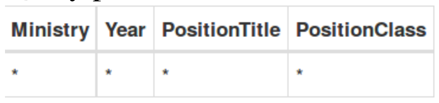
1.2.4 Existing Data Exploration Tools

While data experts have many data exploration tools at their disposal (see Chapter 2 for more details), we do not expect everyone to know how to access and use these tools. As we mentioned earlier, technical literacy and time spent with the data challenge the impact of open data. In this section, we discuss two existing tools, namely, spreadsheets and cloud-based data exploration as a service (*DEaaS*), as an example to highlight the main challenges that would face a concerned citizen of Alberta who does not have the necessary technical knowledge to explore the `Salary` data set. In contrast to the data scientist, the concerned citizen, which we call Bob, does not have a computer science background. He does not know SQL and does not know how to install or access an RDBMS. Thus, we consider two options that are available to him: either using a spreadsheet tool or using a cloud-based service for data exploration. In this section, we discuss the shortcomings of these two options.

First, Bob can rely on simpler data analysis tools, such as spreadsheet software, e.g., Google's Sheets or Microsoft's Excel. We consider spreadsheets because they have a low technical barrier to entry since they depend heavily on point and click user interactions—instead of a specialized syntax. This point and click user interaction model is often referred to as *What You See Is What You Get* or WYSIWYG for short. While spreadsheets are easy to use, they have several limitations compared to the more specialized tools that depend on a specific syntax or commands, such as an RDBMS that requires knowing SQL. Spreadsheets can be easily used to reveal high-level aggregates such as finding the average, minimum, and maximum values; however, they are not preferred when it comes to higher detailed data exploration scenarios.

Navigating the data cube using spreadsheets is challenging because of two main reasons. First, navigating the data cube requires expressing complex analytical queries; this may be too complex for Bob to achieve using spreadsheets. Recall, that Bob does not have the required programming skills to express user-defined functions needed to navigate the data cube in the spreadsheet environment. Second, the data cube exploration process is characterized to be iterative and thus its performance is desired to meet the stringent latency requirements for an interactive expe-

Query pattern:



Ministry	Year	PositionTitle	PositionClass
*	*	*	*

Query in SQL:

```
SELECT    AVG(HighSalary), COUNT(*)
FROM      Salary
```

Figure 1.1: An example query pattern matching all the records in the `Salary` data set.

rience. The performance of the WYSIWYG tools is expected to be slower than RDBMS that is optimized for performance—thanks to the decades of research in query optimization techniques. These two limitations are amongst the main motivations that inspire researchers to propose new alternative user interaction models with RDBMS, such as visual query builders, namely, query building techniques that do not depend on SQL. However, ultimately these techniques generate easy to optimize queries—mainly SQL statements. We discuss such techniques in Chapter 2 in more details.

The second option for data exploration is for Bob to rely on cloud-based data exploration as a service (DEaaS) tools that seamlessly integrate a relational database as a service (RDS) with a web-based visualization layer (e.g., Tableau[8] or Amazon's Quicksight[9]). DEaaS have the advantage of supporting SQL, which allows for more efficient data cube navigation—compared to spreadsheets. In fact, at a high-level DEaaS can be as effective as Afterburner in solving the infrastructure challenge. Currently, DEaaS manages an RDBMS for the users. However, the technology is not mature enough and is not widely adopted. We view our proposal as better suited for democratizing analytics for two main reasons. Firstly, our proposal combines an exploration guidance tool, which solves the second challenge as well. Secondly, Afterburner is open-sourced and portable, which guarantees it will be forever free of charge and allows for adding extensions. Our work allows the open-source data veterans to add features to our proposed stack, for instance, additional tools for data exploration guidance. At their early stages, DEaaS offer access to their proprietary services, conceptually in the future, it should be possible for the DEaaS providers to offer to their users public API's that would allow for their extensibility.

1.2.5 Example: In-Browser Data Cube Navigation

In the previous sections, we have outlined a data exploration task for the `Salary` data set as it would be done by an expert data scientist who has access to an RDBMS and knows SQL. We also highlighted the main shortcomings of the existing tools that are available for non-technical users. In this section, we outline a data exploration task that is based on the same data set using

[8]https://www.tableau.com/
[9]https://quicksight.aws/

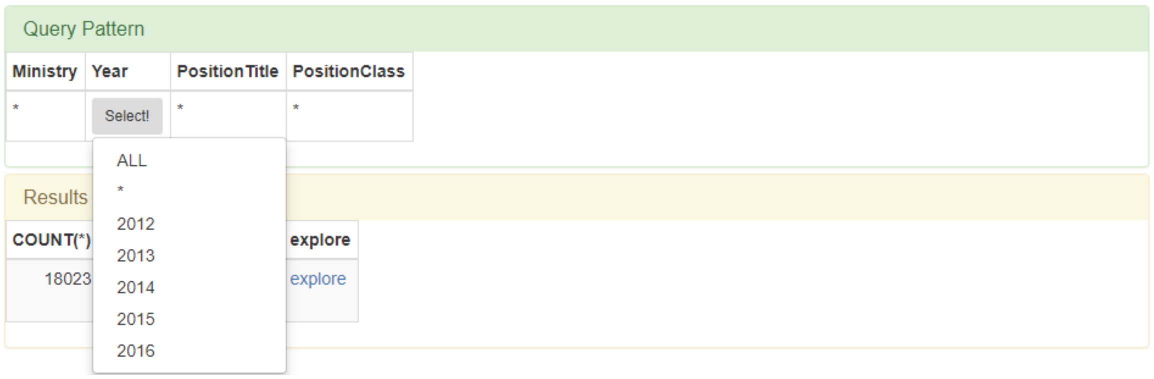


Figure 1.2: A drop-down menu showing the different data navigation options available to the user, using `Salary`.

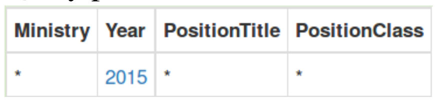
our proposed in-browser SQL engine. We argue the task does not require any computer science background for two main reasons. First, our in-browser SQL engine eliminates the need for any external dependencies. Second, the intuitive data exploration user interface, which encapsulates the required syntax needed to navigate the data cube, eliminates the need for knowing SQL.

Bob can load the data set into our proposed in-browser SQL engine (Afterburner) and start navigating the data cube. As we will show in Chapter 3 Afterburner supports a wide variety of analytical SQL queries, which covers data cube navigation operations, at a comparable latency to state-of-the-art native SQL engines. Afterburner allows for the seamless deployment of both the user interface and the engine since they both readily run inside the web browser and do not require any external dependencies. Thus, he can navigate the data cube based on simple point and click commands such as roll-up and drill-down. While the point and click commands generate SQL queries, Bob does not need to know SQL in order to use the interface.

We now outline a data cube exploration tool using an intuitive user interface that generates SQL queries that run in the browser. One potential starting point to explore the outcome attribute, `HighSalary` in this case, is the first pattern in the data cube shown in Table 1.2. The first pattern is shown to the user in an interactive tabular format illustrated in Figure 1.1.

The top cells in the table show the names of the different dimension attributes in the data set. The bottom cells in the table are active menus that prompt the user for a possible data cube navigation operation. When clicked, a drop-down menu shows all the possible values in the dimension (as they appear in the data set) in addition to the value * (match-any), and value `ALL` that signifies a break down by all possible values (or a drill-down). For instance, the `Year` attribute in the data set has values ranging from 2012 to 2016. Figure 1.2 illustrates the drop-down menu that is shown to the users when they click on the bottom cell of the `Year` attribute.

Query pattern:



Ministry	Year	PositionTitle	PositionClass
*	2015	*	*

Query in SQL:

```
SELECT   Year, AVG(HighSalary), COUNT(*)
FROM     Salary
WHERE    Year = 2015
GROUP BY Year
```

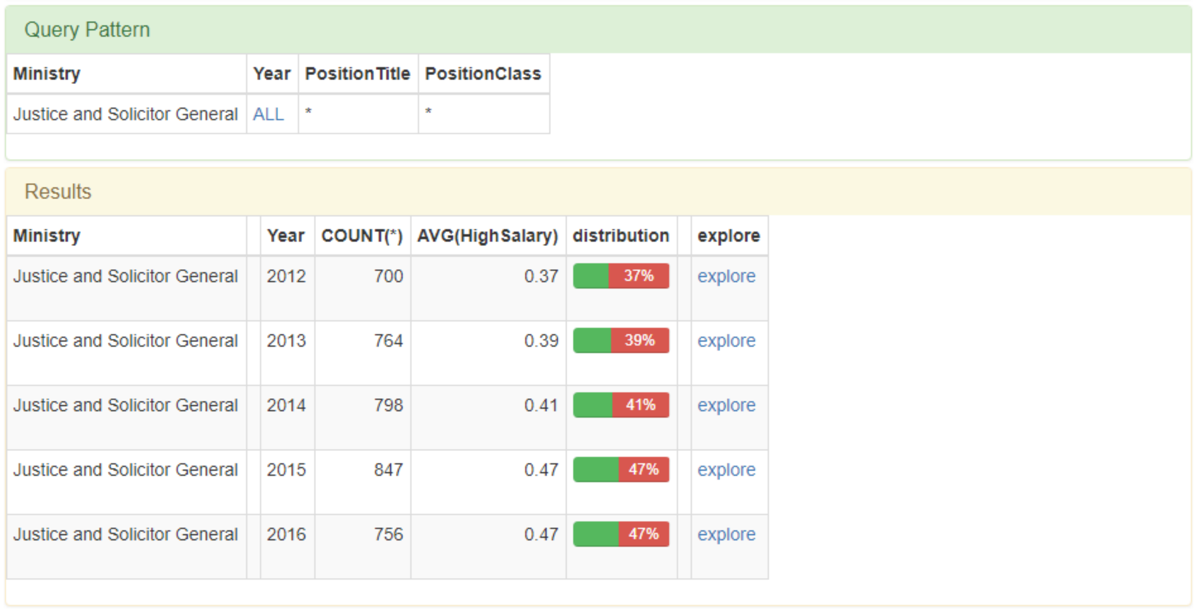
Figure 1.3: An example query pattern matching all the salaries for the year 2015 in the `Salary` data set.

In addition to drill-down, Bob can filter by a specific value for an attribute by picking the value from the drop-down menu. For instance, clicking on the value 2015 means the user is interested in the outcome attribute value for this year. This is equivalent to the fifth pattern shown in the data cube (Table 1.2) and will be represented in our user interface as shown in Figure 3.2.

Alternatively, to navigate to the less specific aggregation group (roll-up) the user can pick the * value in the drop-down menu. As discussed earlier, Bob can use this simple interface to navigate through all the patterns in the entire data cube to explore high salaries. For example, consider that the user wants to express the same drill-down shown in Table 1.3. To achieve this, the user can pick the value Justice and Solicitor General for the `Ministry` attribute, pick the value `All` for the `Year` attribute, and pick the value * for all other attributes. Figure 1.4 shows the output of such interaction in the *Results* panel (bottom). For convenience, the user interface allows the user to pick new data cube patterns from the Results panel to be added to the Query Pattern panel (top), in order to facilitate further navigation.

1.2.6 Portable Data Cube Exploration

Thus far, we have discussed how the in-browser SQL engine, with the help of an intuitive user interface, solves problems related to the analytics infrastructure and technical knowledge barriers. We now address another data exploration challenge which is the large number of patterns in the data cube. As mentioned earlier, the data cube of the public salaries data set (`Salary`) has 80495 patterns. The number of patterns in the data cube is typically large and tends to grow quickly as the number of attributes increase. The large number of patterns in the data cube would overwhelm the user. To tackle this problem, we employ an information-theoretic approach for data exploration, i.e., we pick the patterns that best that can be used to recover the outcome attribute (`HighSalary` in our example) given the dimension attributes. Such summaries are called Explanation Tables; in Chapter 4 we evaluate the efficiency and effectiveness of Expla-



Query Pattern

Ministry	Year	PositionTitle	PositionClass
Justice and Solicitor General	ALL	*	*

Results

Ministry	Year	COUNT(*)	AVG(HighSalary)	distribution	explore
Justice and Solicitor General	2012	700	0.37	37%	explore
Justice and Solicitor General	2013	764	0.39	39%	explore
Justice and Solicitor General	2014	798	0.41	41%	explore
Justice and Solicitor General	2015	847	0.47	47%	explore
Justice and Solicitor General	2016	756	0.47	47%	explore

Figure 1.4: In the browser drill-down by `Year` for the salaries in the `Justice and Solicitor General` ministry which corresponds to Table 1.3.

nation Tables compared to related techniques. In addition, we show how to efficiently generate Explanation tables in the browser.

Explanation Tables depend on information theory to pick the most informative patterns in the data cube, nevertheless, the user does not need to know about information theory to be able to use them. Furthermore, since Explanation Tables are expressed in terms of patterns from the data cube they are readily interpretable by non-technical users. For instance, Table 1.5 shows an explanation table for the `HighSalary` attribute. Similar to the data cube table, we will refer to the rows of an explanation table as patterns. The first pattern in the explanation table reveals the average value for the `HighSalary` attribute in the entire data set, which is 19%. Notice, this is the first pattern in the data cube, i.e., Table 1.3. The second pattern in the explanation table reveals that 56% of the position class Executive Manager 1 have high salaries. The third pattern reveals that 42% of the Justice and Solicitor General ministry have high salaries. Finally, the fourth and fifth patterns reveal that almost 0% of the position classes Manager and Manager Zone 2 have high salaries. The data scientist, as well as Bob, can use this explanation table to get a sense of the most important factors deciding higher salaries in terms of the values of the dimension attributes. In this example, the explanation table reveals likely the most important insights in the `Salary` data set that are: the typical portion of public servants having high salaries and the ministries and job classes having portions that deviate from the typical.

Ministry	**Year**	**PositionTitle**	**PositionClass**	AVG(`HighSalary`)	COUNT(*)
*	*	*	*	0.19	18023
*	*	*	Executive Manager 1	0.56	1913
Justice and Solicitor General	*	*	*	0.42	3865
*	*	*	Manager	0.00	2705
*	*	*	Manager Zone 2	0.00	1754

Table 1.5: An Explanation Table over `HighSalary` from `Salary`.

Let us consider the first two patterns in the explanation table to discuss the intuition behind using information theory to generate Explanation Tables. A straightforward approach to picking the most informative patterns is to the pick the patterns matching the largest groups, i.e., having the highest COUNT. This approach is not effective, since, in real data sets, patterns tend to overlap, i.e., their records intersect. Let us consider another candidate group from the data cube, such as the group with position title of Senior Manager. While the group of Senior Managers is almost twice as large in number than the group of Managers (5201 compared to 2705), the average value of `HighSalary` for Managers deviates much more from the average salary of the entire data set—as revealed by the first pattern in the explanation table. Henceforth, we continue to pick the patterns that deviate the most from the prevalence of the entire data set. Thus, to find the most informative set of patterns from the data cube; the intersections, the sizes, and the average values of the groups must be taken into consideration, and are all considered in the maximum entropy approach employed in Explanation Tables.

In addition to providing readily interpretable summaries, the patterns of an explanation table are good candidates for a starting point for data cube exploration scenarios. As discussed earlier, the most informative parts of the data cube are likely to be important or interesting starting points in data exploration tasks. In this section, we have discussed how our proposed in-browser data exploration stack helps anyone who has access to a web browser to explore open data found on the web.

1.3 Desiderata

Our goal is to enable everyone to take part in the analytics process rather than passively consuming its outputs, i.e., analytics democratization. We would like to clarify we do not aim to remove the data experts from the analytics loop. Rather, we aim for allowing users without a technical background to easily analyze data. In addition, we also aim for allowing data experts to easily curate data analytics tasks to share with their audience over the web. The main premise is that data becomes potentially more useful when analyzed by more people (as discussed in Section 1.1). In the previous section, we highlighted the main challenges to analytics for everyone, in this section, we highlight the main desired features in analytics tasks:

- **Easy to use:** Non-data experts should be able to analyze the data with as little as possible technical knowledge, for instance, users may not know SQL. The user should be isolated—as much as possible—from any details regarding the inner-workings of the tasks. For interactive analytics tasks, the user should not be burdened by figuring any necessary pre-configurations that optimize data access or computations required to realize the analytics tasks. In addition, for data science tasks, there are available decision support tools but they prompt the user for configuration parameters that require understanding how the algorithms work.
- **Easy to interpret:** We desire that data science tasks produce immediately interpretable insights with respect to an outcome attribute. Immediately interpretable insights do not require further steps of analysis or complex reasoning. Using groups that share something in common, such as same job titles, locations, or types, is an intuitive strategy for producing immediately interpretable insights. This can be mapped naturally to categorical dimensional attributes of relational tables. For instance, using the `Salary` data set, `PositionTitle = Manager` is the least likely group to get high salaries, is a self-explanatory insight.
- **Self-contained:** Does not depend on multiple tools, expensive infrastructure, or external dependencies, such as a dedicated backend. Many data analytics pipelines depend on the integration of several tools such as storage engines, execution engines, and frontend engines. Although combining multiple tools is a common practice for the data experts, we require that the non-technical users deal with one self-contained application. This does not only simplify the task for the user but also allows for easily publishing and sharing the analytics tasks with more people.
- **Performance:** The analytics task should maintain a fairly comparable performance in terms of resources used and latency, compared to other existing systems that do not fulfill our list of goals.

Contributions. This dissertation has the following main contributions:

- First, Chapter 3 shows experimentally the viability of fairly complex SQL analytics in the order of tens of millions of records running inside a web browser without any external dependencies. Afterburner [18] uses query compilation techniques, columnar data layout, and small memory footprint operators to support analytical SQL running at latencies comparable to state-of-the-art SQL engines running natively. A portable yet efficient SQL engine allows data scientists as well as users without technical background to easily analyze modestly-sized data sets without requiring any knowledge about managing an RDBMS. Furthermore, this result encourages revisiting alternative deployment architectures for interactive analytical SQL which we study in depth in Chapter 5.

- Second, Chapter 4 evaluates the efficiency and effectiveness of different algorithms that generate Explanation Tables [16]. Since generating Explanation Tables is complex, we evaluate optimized sample-based pruning technique (Flashlight), which requires a small fraction of the runtime of a baseline algorithm that does not apply any pruning (Baseline). Using real data sets, we show that Flashlight generates Explanation Tables that are more informative than existing approaches. We evaluate an RDBMS implementation of Explanation Tables that uses a sequence of SQL views, which simplifies generating Explanation Tables by pushing many of the data management problems to the RDBMS. In addition, we present an in-browser implementation of Explanation Tables which adapts the algorithm to the browser. Our evaluation shows the viability of guiding data exploration tasks in the browser without any external dependencies.

- In Chapter 5, to accelerate interactive analytics for larger data sets, we propose splitting query plans between the backend and the frontend. We outline our engine that seamlessly splits query execution such that one query runs once against the backend to create a materialized view which is used to optimize future similar queries. Future queries run entirely on the frontend. Our technique comes with two classes of benefits: hint-based seamless optimization and decoupling the frontend application from the backend RDBMS. Based on a high-level user hint our engine seamlessly optimizes queries bringing the latency down for a more responsive interactive experience. In addition, decoupling the analytics applications from the backend comes with the following benefits: removing administration overheads associated with maintaining materialized views at the backend, removing network round trip latency per user interaction, and offloading the backend.

This dissertation combines different techniques that enable portable and easy to use yet optimized interactive analytics and data exploration. This is interesting because exploration and interactive analytics tasks usually require a wide suite of expertise and expensive infrastructure not available to many. The main contribution of this is opening the door for democratizing data exploration and analytics and allowing for new unlimited opportunities—that to our knowledge are thought to be not viable before this— by lowering the entry barriers or in other words analytics for everyone.

Chapter 2

Related Work

In this chapter, we discuss existing research and techniques that are related to our work.

2.1 In-Browser Analytics

SQL for everyone: As discussed in the previous chapter, traversing relational data is more suited to SQL because of its expressive power and amenability to optimization by an RDBMS engine, nevertheless, many people without a background in computer science do not know SQL. Thus, recent research came up with several tools such as [9, 30, 31, 33, 35, 42] that allow users who do not know SQL to express queries using alternative ways. Data Tweening [35] and Gesture-Query [30] allow users to use touch interfaces, such as the screens of smart devices, to express queries. Foofah [31] allows users to formulate queries using example records. NaLIR [42] allows users to use natural language to query the database. DATASPREAD [9], is geared towards allowing users to express queries over spreadsheet-like data representation. ETable [33] allows users to express iterative queries using visual interactions over data sets that are presented in a tabular format without using SQL.

There are different ways to isolate data exploration users from SQL. In Chapter 1, we have introduced our simple data cube navigation tool; recall that while the tool generated SQL queries, its users did not need to know SQL. One common theme among these techniques is producing SQL which requires an RDBMS. We view Afterburner as a good complement for all of these techniques since combined with an in-browser SQL engine; SQL data exploration becomes even more accessible to the less expert users (everyone).

Compiled queries: The compiled query approach of Afterburner takes after systems such as HIQUE [38], LegoBase [36, 51], Proteus [34], and HyPer [45], which have recently popularized code generation for relational query processing. With the exception of targeting JavaScript and using plans that fit into a browser's limited storage capability (which to our knowledge is novel), our query compilation techniques are fairly standard. We use a template-matching approach to generate code, for instance, simple SELECT-WHERE queries are converted into `for` loops over the appropriate ranges of a typed array holding the data.

One well-known drawback of query compilation is that compiling generated code using a tool such as `gcc` can overshadow its benefits for short-running queries. Much research has gone into alleviating this issue, such as the use of an intermediate representation such as LLVM [45]. In our work, however, we have found compilation overhead to be negligible, primarily because compilation speed is already something browsers optimize for since all JavaScript code on the web is stored as text.

Query compilation into code usually targets lower level programming languages, such as C, that are imperative. The generated code can be expressed in multiple ways that are logically equivalent but may lead to different query latencies. LegoBase [36, 51], focuses on the optimizing the generated code. At a high-level, it compiles queries into LLVM then applies multi-level code optimizations such as exploiting CPU level optimizations. Afterburner, however, targets asm.js where many of the optimizations that LegoBase exploits do not apply. Thus, in Afterburner our code generation templates depend only on the queries but not the database statistics.

Portable and in-browser data Processing: We are not the only ones to explore the unconventional idea of portable or in-browser data processing. For instance, JScene [43] has recently demonstrated the viability of porting a search engine into JavaScript in order to run completely in the browser without external dependencies. For SQL, while there are some existing tools that support SQL queries in the browser such as Lovefield and sql.js, these tools are limited with respect to the size of the databases and the complexity of the queries they support. In fact, to our knowledge, we are the first to demonstrate the viability of relatively complex queries, such as multiple joins etc., for millions of records inside the browser.

2.2 Guiding Data Exploration Tasks

SURPRISE operator: Explanation Tables are a natural extension to [48, 50] and employ a similar information-theoretic approach for relational data summarization. We shall refer to the n pattern version of the summarization technique proposed in [48, 50] as the *SURPRISE* operator. SURPRISE takes as input a database, the size of summary in terms of patterns (n), and an accuracy threshold ϵ which defines a trade-off between running time and quality (information content of the summary). SURPRISE employs a dynamic programming approach that finds the best summary. However, the SURPRISE operator generates restricted patterns that cannot be partially overlapping. In other words, it generates patterns that are either disjoint or fully contained. We alleviate this restriction in Explanation Tables, which adds to the complexity of the problem. First, disallowing overlapping patterns can significantly simplify the problem, since selecting a pattern rules out all of the patterns it intersects with. Second, the computation of information content of a set of overlapping patterns becomes more challenging as we will discuss in more details in Section 4.1.

Data cube exploration: In our work, we have focused on finding the most informative patterns to present to the user. It is also important to discuss other work that uses different criteria for guiding users. For example, MacroBase [7] finds the group of records that are deemed as outliers. While outlier detection techniques are known to be computationally demanding an important contribution of MacroBase is using sampling-based approaches that detect outliers in fast streams of data. Smart Drill-Down [32] is similar to Explanation Tables since it also finds the patterns in the data cube that deviates from the maximum entropy estimates. However, Smart Drill-Down allows users to tune their criteria of importance, i.e., users can adjust the criteria of picking patterns to their own needs. For example, Smart Drill-Down allows users to pick patterns with a higher number of *.

MacroBase and Smart Drill-Down can complement Explanation Tables, i.e., provide alternative ways to prioritizing patterns or groups of records. Thus, we would like to study incorporating these techniques into future versions of our work.

Scalable pattern mining for relational data: Techniques that are concerned with the scalability of pattern mining are also related to our work. For example, Nandi et al. [44] introduce a framework that facilitates mining the data cube using MapReduce. Feng et al. [19] introduce SIRUM, which is a distributed framework that allows users to use SPARK to mine informative patterns over large clusters. While the main goal of [19, 44] is scalability through distribution over multiple machines, our focus in this thesis is better performance in-browser.

2.3 Accelerating SQL Data Exploration Tasks

Split-Execution: The idea of considering split-execution between different tiers was studied by previous work [12, 14, 20]. Chapter 5 revisits the same idea for accelerating interactive analytical SQL queries given two new conditions: First, the ability to execute analytical queries on the frontend efficiently and with minimal administrative effort; second, allowing the data scientist to use a declarative and easy way (free clause) to define a query template to accelerate. Our idea of “freeing” columns is related to the work of Koudas et al. [37] in “relaxing” join and selection queries, but their goal is to “back off” from queries that return empty results. Also related is the semantic pre-fetching idea of Bowman and Salem [12], who try to predict future queries based on past history; in our case, we require explicit hints from the user.

Materialized views: Taking advantage of materialized views to optimize complex queries is a well-studied technique dating back decades [25, 26, 52, 56]. There are, however, substantial differences with our approach: in most previous work, materialized views are long-lived and carefully-considered optimizations by a database administrator, not built on an ad hoc and per-query basis—which is the approach that we take. For us, materialized views are transient and lightweight, precisely because they are shipped over to the browser and can be discarded when done. This approach exploits our in-browser JavaScript engine, which means that not only is the integration seamless, but subsequent interactions can happen without the backend.

Physical design tuning: In a sense, Afterburner shares similarities with physical design advisers [4, 13, 53] since Afterburner picks materialized views automatically, without requiring special expertise or adding an extra administrative burden. Physical design advisers choose a set of materialized views that optimize a query workload under constraints, such as fitting a storage budget. The query optimizer is heavily engaged in the process of identifying a candidates space to pick from (i.e., depend on the query planner enumeration space [53] to look for possible materialized views). In Afterburner, we apply query re-writing rules to generate the materialized view definition which can generate materialized views that would not be considered by a query planner.

Shared work and shared data. Our work also shares similarities with techniques targeted at accelerating streams of queries by sharing parts of their plans [22, 28]. Similar to this line of work, Afterburner optimizes multiple queries by finding a common subplan (the materialized view). However, the optimization goals and approaches are different. Our goal is to minimize the latency of a family of queries anchored around a single query template for a specific user, while subplan sharing techniques have the goal of improving overall system performance, e.g., higher query throughput, typically across multiple users. Once again, despite superficial similarities, Afterburner targets a completely different point in the design space.

Approximate query answering: Related to using materialized views to optimize queries are techniques that provide approximate query answers that can be computed faster than the exact answers [1, 2, 3, 21, 46]. One main difference between using materialized views and approximate query answers is that Afterburner produces exact answers. We find that currently approximate query answering techniques are limited since; they either work for a limited class of queries, such as [1, 2, 21] or require a large number of offline training queries, such as [46]. At a high-level, approximate query answering approaches focus on optimizing interactive related queries, where the system is able to approximate the distribution of the underlying data. Thus, this line of work shares a similar intuition to ours, however, the techniques are different. In principle, these techniques can be added to Afterburner, which can be a promising extension of our work.

Chapter 3

In-Browser SQL Analytics with Afterburner

3.1 Introduction

In Chapter 1, we have identified two main challenges that face users without data analysis expertise who want to analyze data. In this chapter, we outline our proposed JavaScript-based SQL engine, which we call Afterburner [18], that runs entirely inside a web browser without any external dependencies. Since it is portable and runs inside any web browser, Afterburner solves the problems related to managing an RDBMS or any other data management infrastructure for moderately-sized data sets, which is the first challenge we focus on in this dissertation. In addition, we show that in-browser SQL processing rivals the performance of existing native SQL engines using modestly-sized databases from the staple data warehouse benchmark TPC-H, which is known for its complex analytical queries over a prototypical data warehouse. We show microbenchmarks that explain why Afterburner's latency rivals native SQL processing. This interesting result encourages revisiting common SQL based data exploration deployments for larger databases, which we study in depth in Chapter 5. In addition, the results also encourage investigating implementing information-theoretic techniques that can guide data exploration tasks in the browser, which we present in Section 4.5.

In Section 1.2.3, we have outlined a data exploration task for the data cube of the `Salary` data set. As discussed, data exploration tasks such as navigating the data cube of a table in order to understand more about an outcome attribute of interest are characterized by an iterative process of data cube navigation operations, such as roll-up and drill-down. The data cube navigation operations depend on computing aggregate values and filters. The iterative and interactive nature of data exploration implies stringent latency requirements for the navigation operations to execute. As explained earlier, data cube navigation operators are best expressed in SQL not only for performance reasons but also for the simplicity of expressing a GROUP-BY and filters using SQL. Analytical RDBMS engines are best suited for such data exploration tasks since they are usually optimized for performance.

Recall, in our example for exploring the `Salary` data set, we have considered Bob's case, who does not possess technical skills, i.e., does not know how to install or manage infrastructure that supports SQL, such as an RDBMS or Spark. A lot of previous work that makes analytics infrastructure easier to manage and install already exists; nevertheless, such tools are still relatively hard to manage by non-technical users. It is important to note that, while Bob might also lack other important data analysis skills such as knowing SQL, our focus in this chapter is going to be on the infrastructure problem for several reasons. First, SQL can be generated by applications such as the data exploration tool we proposed in Chapter 1. Recall that the user could navigate the data cube using our intuitive user interface that does not require any knowledge of SQL. This kind of navigation is very common, for instance, Sweden's OpenAid project makes available a graphical visualization of the total value for the aid given grouped by any of the following: granting institution, benefiting country, benefiting institution, etc., which is also another way for exploring the data cube of the OpenAid data set. Second, the problem of allowing users to express analytical queries without knowing any special syntax, e.g., SQL, has received plenty of attention from the research community. The proposed techniques and applications can also benefit from Afterburner since many of these generate SQL queries based on the end user's interactions—i.e., Afterburner can replace the backend for these tools and they become portable as well. We discuss some of these techniques in Chapter 2. In addition, Afterburner can be useful to data experts, such as data scientists, who do not want to deploy an RDBMS in certain situations. For instance, a data scientist may not want to install an RDBMS on certain clients, such as their handheld devices. Thus, the main goal of this chapter is to reduce any knowledge barriers required to deploy or manage an RDBMS, especially for modestly-sized data sets. Namely, this chapter investigates whether it is possible to make data processing as accessible as a web page.

3.1.1 In-Browser Analytics

In this section, we elaborate on the key features that make the web browser an attractive deployment environment for portable analytics. Web browsers have two main advantages: they are ubiquitous and they have already gained high popularity within the data science and data exploration circles.

First, web browsers are widely-available on many clients. Web browsers are deployed on a variety of types of devices that spans phones, tablets, personal computers, laptops, and many smart devices—such as smart fridges, etc. We compare this to other RDBMSs that might have requirements, such as escalated user privileges on the client in order to install, or limited operating system version compatibility, etc. In addition, such ubiquity allows data experts such as data scientists to share their data exploration tasks with everyone over the web. The current version of Sweden's OpenAid project's web page includes a data exploration task that is similar to the `Salary` exploration task we outlined earlier—as both tasks navigate the data cube of a data set. To our knowledge, we are the first to propose an in-browser implementation of an SQL engine that is geared towards modestly-sized data sets. We do not know of any existing implementation of an in-browser SQL based data processing at such scale. This opens the opportunity of having data visualizations over the web for much larger data sets. For instance, Sweden's OpenAid data set is in the order of few thousand records. Our tool supports larger data sets that are in the order of millions of records. In addition, Afterburner allows for even more complex analytical tasks.

Second, web browsers are already the number one choice for many data science applications. We argue that the readiness of the web browser is one of the main drivers that made it a very popular data science and data visualization deployment environment. For today's data experts, the browser has become the shell, especially for interactive analytics. What used to be accomplished via command-line and REPL-based tools is increasingly moving into browser-based notebooks such as Jupyter. Such tools have gained tremendous popularity due to the tight integration of code and execution output, which elevates the analytical process and its products to first class citizens since the notebook itself can be serialized, reloaded, and shared. Notebooks also tie analytics tools into existing software ecosystems (e.g., Python and R), giving data scientists access to a broad range of capabilities. Integrating data processing inside the web browser without any external dependencies simplifies data exploration.

3.1.2 Main Challenges

Analytical data processing inside the web browser is challenging for several reasons that are related to performance and access to resources. We argue that because of these challenges, currently, analytical queries inside the web browser are limited in scale (e.g., small number of records) or in capability (e.g., only simple filters). In this section we elaborate on two of the main challenges to SQL processing inside a web browser.

The first and most important challenge is the performance of JavaScript. While many associate the language flexibility features as the main reason for its popularity, such flexibility makes JavaScript inherently harder to optimize. Features such as just-in-time (JIT) compilation and support for varying data types and dynamic data structures such as arrays and dictionaries add to the complexity of optimizing JavaScript (compared to other languages such as C) and add certain overheads to the runtime environment, e.g., garbage collection and type checking during runtime. Hence, the popular perception that JavaScript is not an efficient language. When it comes to performance-demanding applications, JavaScript inside a web browser is not the environment of choice.

The limited runtime environment inside the web browser is another challenge for SQL processing even for modestly-sized data sets. For instance, JavaScript code running inside the web browser does not have direct access to the machine's resources nor does it have direct access to many of the operating system's services. Such limitations are very good for security reasons since web browsers are usually used to navigate web pages over the open internet; however, they come at a certain performance cost. For instance, JavaScript inside the browser does not have a direct and efficient way to access the client's hard-drive. Thus, to our knowledge, the web browser is usually used as a dumb rendering endpoint in data science and data analytics tasks, i.e., merely a presentation layer.

In the next section, we discuss the main features of our in-browser SQL engine Afterburner that address these challenges. In fact, overcoming the performance challenges of SQL processing inside the web browser is a main technical contribution of this chapter.

3.1.3 Afterburner's Key Features

In this section, we elaborate the key features of Afterburner. Afterburner represents a new design point for data analytics since it can "bring the best of both world" in terms of the benefits of web browsers while overcoming their performance challenges. Afterburner achieves this by exploiting two recent advancements in JavaScript: namely typed arrays and code generation targeting asm.js.

Typed arrays allow Afterburner to implement an in-memory columnar store, which is known to be read-optimized. We use typed arrays for the columnar format storage of the data in memory. For code generation, Afterburner's engine depends on generating code using code templates to implement its SQL operators. For group by and join, afterburner depends on hash-based plans. In addition, Afterburner exploits asm.js, which is a statically typed subset of JavaScript and is amenable to ahead-of-time (AOT) compilation. Finally, JavaScript is fast to compile since JavaScript is usually shipped in text format over the web, so fast compilation received a lot of attention over the course of time. The fast compilation time of JavaScript can be advantageous especially for modestly-sized databases where the overhead of optimizing the compiled queries may not be justified compared to the overall query latency. For instance, in our experiments, other code generation techniques that target C take on average 330 ms for query compilation only; on the other hand, it takes less than 1-2 ms to compile JavaScript queries. Given that the average query latency is 288 ms using JavaScript, the performance benefits of native compilation and optimization may not be justified (for our targeted database sizes).

3.2 Contributions

Contributions. We view our work as having the following main contributions:

- The primary contribution of this chapter is a feasibility demonstration of in-browser analytics. To our knowledge, we are the first to propose this somewhat unconventional idea of embedding an analytical RDBMS inside the browser.
- We describe and evaluate Afterburner, our prototype analytical RDBMS implemented using JavaScript that runs completely in the browser. We detail how our system takes advantage of in-memory columnar storage using typed arrays and query compilation into asm.js.
- Microbenchmarks, as well as end-to-end evaluation, provide a characterization of performance against an existing columnar database and the latest query compilation techniques. Results show that our system is comparable to the state-of-the-art technology for SQL analytics.

3.3 Afterburner Design

We begin by detailing the design of Afterburner, which takes advantage of two key JavaScript features: typed arrays for memory-efficient storage and asm.js for fast compiled queries.

3.3.1 Columnar Storage with 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 the evolution of JavaScript, it became clear that the language needed more efficient methods to manipulate binary data: typed arrays are the answer.

Typed arrays in JavaScript are comprised of buffers, which simply represent untyped binary data, and views, which impose a read context on the buffer. As an example, the following creates a 64-byte buffer:

```
var heap = new ArrayBuffer(64);
```

Before we can manipulate the data, we need to create a view from it. With the following:

```
var hI32 = new Int32Array(heap);
```

we can now manipulate `hI32` as an array of 32-bit integers (e.g., iterate over it with a `for` loop).

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. Figure 3.1 shows the physical memory layout storing the `lineitem` table from the TPC-H benchmark, which we use as a running example. The bottom row represents the raw buffer, while the top three rows represent the various views. A `lineitem` pointer serves as the entry point into a group of 32-bit integer pointers, which represent the offsets of the data in each column (`l_orderkeys`, `l_partkeys`, etc.). Currently, Afterburner supports integers, floats, dates, and strings. For the first three types, values are stored as literals (essentially, as an array). For a column of strings, we store null-terminated strings prefixed with a header of pointers into the beginning of each string, essentially a `(char **)` in C. Intermediate data for query execution in Afterburner are also stored using typed arrays.

In this chapter, we adopt the following naming convention for JavaScript variables: `heap` refers to an instance of a binary ArrayBuffer. Names of views over the typed array begin with an `h`, followed by the first letter of the data type, then the size of the data type in bits. For instance, `hI8`, `hI32`, and `hF32` represent Int8, Int32, and Float32 views over a typed array, respectively.

3.3.2 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. Consider the following fragment of JavaScript for counting the number of records that match a particular predicate on `l_extendedprice`:

```
function count(val){
  var cnt = 0;
  for(var id; id < l_extendedprice.length; id++)
    if (l_extendedprice[id] < val) cnt++;
  return cnt;
}
```

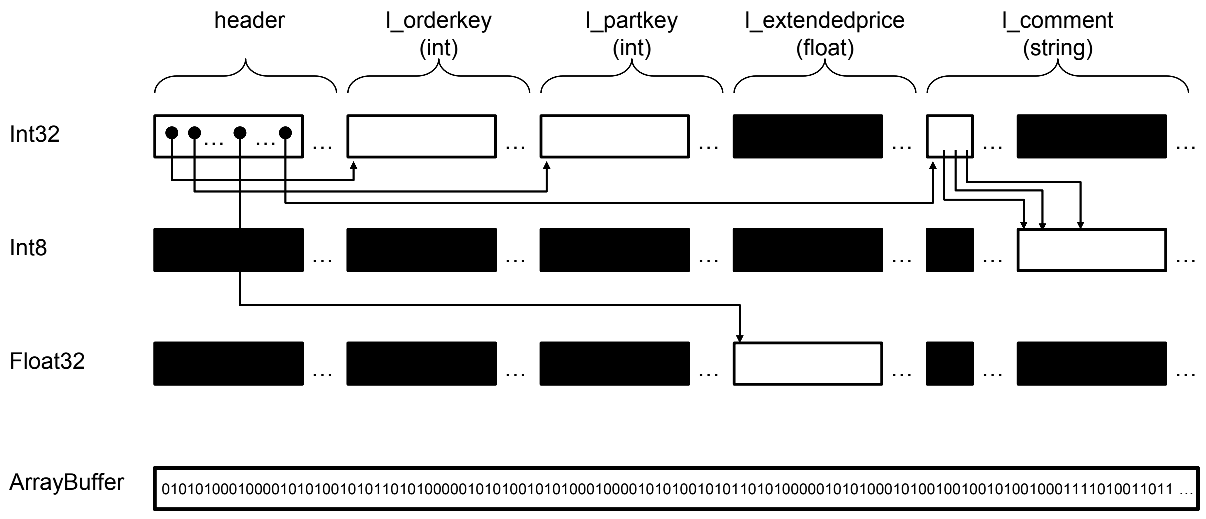


Figure 3.1: Illustration of the physical in-memory representation of the `lineitem` table from TPC-H. Different typed views (top three rows) provide access into the underlying JavaScript ArrayBuffer. Blackened boxes represent invalid data for that typed view.

The equivalent function in asm.js is as follows:

```
function count_asm(val, l_extendedprice, length){
  "use asm";
  val =+ (val);
  length = length|0;
  length = length << 2;
  id = 0;
  while ((id|0) < (length|0)){
    if (+(hF32[((l_extendedprice + id)|0) >> 2]) < +(val))
      cnt = (cnt + 1)|0;
    id = (id + 4)|0;
  }
  return cnt|0;
}
```

The above function takes as parameters: `l_extendedprice`, which is the starting byte offset of the `l_extendedprice` column, and `length`, which is the number of records in that column. The `hF32` variable is a 32-bit float view and thus the byte offsets can be computed by multiplying the index variable `id` by 4 using the shift operator (<<). Asm.js uses type hints, such as `x|0` and `+(x)`, which are applied to variables or arithmetic expressions. The type hint (`x|0`) specifies a 32-bit integer and `+(x)` specifies a 32-bit floating point value. With these hints, asm.js essentially introduces a static type system while retaining backwards compatibility, since in "vanilla" JavaScript these hints just become no-ops.

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 AOT compilation, in contrast to JIT compilation in vanilla JavaScript. The executable code generated by AOT compilers can be quite efficient, through the removal of runtime type checks (since everything is statically typed), the operation on unboxed (i.e., primitive) types, and the removal of garbage collection.

An asm.js module can take three optional parameters, which provide hooks for integration with external JavaScript code: a standard library object, providing access to a limited subset of the JavaScript standard libraries; a foreign function interface (FFI), providing access to custom external JavaScript functions; and a heap buffer, providing a single ArrayBuffer to act as the asm.js heap.[1] Thus, a typical asm.js module declaration is as follows:

```
function AsmModule(stdlib, ffi, heap){
  "use asm";
  // module body
}
```

At a high-level, Afterburner translates SQL into the string representation of an asm.js module (i.e., the physical query plan, through code templates described below), 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 (i.e., the entire database) is passed into the module as a parameter, and the query results are returned by the module. In the next section, we describe Afterburner's fluent API, which allows its users to express queries.

Instead of string-based SQL queries, Afterburner executes queries written using an API that is heavily driven by method chaining, often referred to as a *fluent* API. There is a straightforward mapping from the method calls to clauses in a standard SQL query, so we can view the fluent API as little more than syntactic sugar. However, this query API is quite similar to DataFrames [6], an interface for data manipulation that many data scientists are familiar with today.

To illustrate the similarity between fluent SQL and SQL, we show in Figure 3.2 a query that counts the number of records with `l_extendedprice` less than 55 expressed in fluent

[1]`http://asmjs.org/spec/latest/`

Query in fluent SQL:
```
abdb.select()
  .from('orders')
  .field(count('*'))
  .where(lt('l_extendedprice', 55))
```

Query in SQL:
```
SELECT COUNT(*)
FROM orders
WHERE l_extendedprice < 55
```

Figure 3.2: An example query in fluent SQL.

SQL. On the left side, the JavaScript object `abdb` represents the main entry point to Afterburner's API, which exposes several methods that can be used to express queries. The method `select` does not take any parameters and returns a new empty query object. The method `from` takes a table name as a parameter, which is similar to the SQL clause FROM. The method `field` takes as a parameter a column name, an aggregate function, or a *User Defined Function* (UDF). In order to allow for expressing aggregate functions, such as COUNT, AVG, or SUM, Afterburner exposes methods with similar names, such as `count`, `avg`, and `sum`. In our example, the method call `count('*')` is passed as a parameter to the method `field`, which resembles COUNT(*) in SQL. The method `where` takes as a parameter filter predicates and resembles the SQL clause WHERE. In addition, Afterburner exposes methods that allow for expressing filter predicates, such as the comparison operators =, >, and <. In our example, the method `lt('l_extendedprice', 55)` resembles the filter predicate `l_extendedprice < 55` in SQL.

3.3.3 In-Browser Query Processing

In the previous sections, we have outlined two main features that characterize Afterburner SQL engine, which are columnar storage and code generation. Other than the challenges related to the JavaScript runtime, the web browser imposes other challenges such as the limited memory budget. In this section, we outline Afterburner's code generation templates in more detail. In addition, we discuss two main design decisions that allow for running complex SQL queries inside the browser.

Starting from an SQL query expressed in our fluent-style API, Afterburner generates the string representation of the asm.js code that corresponds to the query. In the current implementation, this is performed based on a small number of fixed code templates in which various subexpressions (e.g., the filter predicate, join key, group by clause, etc.) are plugged. At present, Afterburner has a fixed (hard-coded) physical plan for each class of queries (i.e., it does not perform query optimization). Our implementation supports a wide variety of SQL analytical operations such that it covers all the queries in the TPC-H benchmark. We outline next the main query templates in more detail:

Simple filters. A code template that generates query plans for simple filter–project or filter–aggregate queries. The code template generates a loop that increments a record iterator, which is used in combination with the starting offset of a column to access a particular attribute. Inside the loop, the template can either generate code to materialize a projection or to compute simple aggregates, such as COUNT, AVG, or SUM.

Joins. The code template for supporting filter–project or filter–aggregate queries over an inner join implements a standard hash join. In the build phase, the code loops over one relation to build the hash table. In the probe phase, the generated code loops over the second relation to probe the hash table for matching records, and then either materializes a projection or computes an aggregate. The template currently only supports two-way joins.

Group bys. The code template loops over one relation to build a hash table over the grouping keys. Another loop iterates over the hash table to process the groups.

Joins with Group bys. This code template implements a standard hash join, similar to the Joins template, to build a hash table over the grouping keys. The main difference between the Joins with Group bys template and the Group bys template is that the hash tables store composite keys (a key from each relation in the join) in order to be able to process the groups.

Subqueries. Afterburner handles subqueries by materializing their output which is used as input tables for other operators. In addition, Afterburner supports IS IN subquery clauses by generating code that is similar to the join code templates.

Example Code Generation with the Join Code Template:

As an example, we consider the following query, expressed in fluent SQL, which computes the sum of `l_extendedprice` column from `lineitem`, filtered by a certain order date range:

```
abdb.select()
  .from('lineitem')
  .join('orders')
  .on('l_orderkey', 'o_orderkey')
  .field(sum('l_extendedprice'))
  .where(lt('o_orderdate', date('1995-03-15')));
```

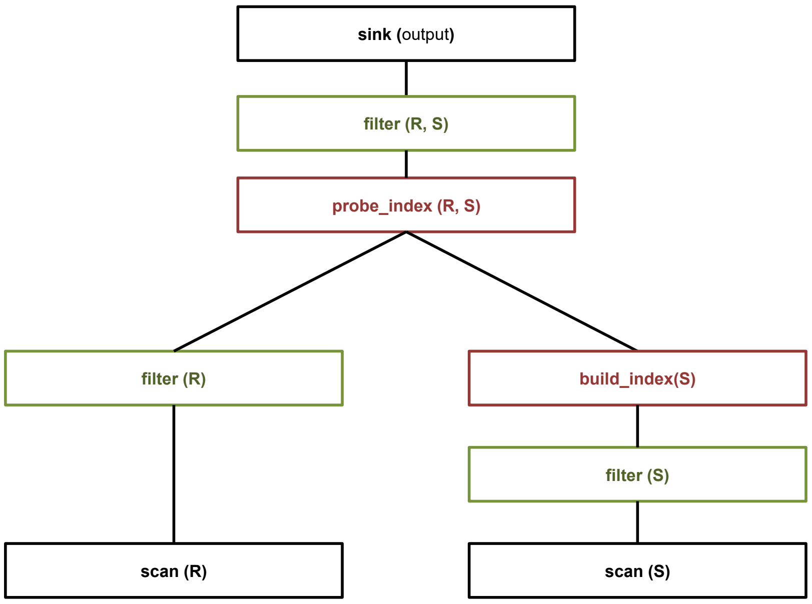


Figure 3.3: A generic physical plan for joins in Afterburner.

In this example query, the above code triggers Afterburner to validate the fluent query and returns its asm.js equivalent code. Afterburner uses simple rules to validate and match queries with code generation templates. In our example, the query has a join and a filter condition but does not have a group by; thus, Afterburner picks the join template for this query. Figure 3.3 shows a generic physical plan for joins. The figure shows low level operators such as `scan` and `filter` for joining two generic relations (`R` and `S`). The low level operators are smaller generic code templates that compose the query code. The join code template depends on `scan`, `filter`, `build_index`, and `probe_index`.

To better describe the physical plan, shown in the figure, we break it down into two stages. At the first stage, a hash index is built over `S` then at the second stage, the relation `R` is scanned to probe the hash index and generate output records. To minimize the query runtime, our query templates prioritize early filtering. The join plan divides the filters in the query into two groups: one that involves a single relation, i.e., either `R` or `S`, and one that involves both relations `R` and `S`. At the first stage, while building the hash index, the filters involving `S` only are applied to minimize any overheads associated with inserting a new entry in the hash table. At the second stage, at each iteration of the loop filters involving `R` only are applied before the hash table probe. Finally, after the hash table probe filters involving both `R` and `S` are applied. This figure summarizes the high-level idea behind the generic physical plan of the compiled queries generated by the join code generation templates.

```
 1: build_join(R, S){
 2:   return '''
 3:     ##build_hash_index(#S, #this.join.S)
 4:     while(1){#R.name = #R.name + 1|0;
 5:       if ((#R.name|0) >= #R.length) break;
 6:       if !##filter(#R) continue;
 7:       ##join_probe(#R.name, #S.name)
 8:         if !##post_join_filter(#R, #S) continue;
 9:         ##generate_record(#R, #S)
10:       }
11:     }  '''
12: }
13: build_hash_index(S, onCol){
14:   return '''
15:   while(1){#S.name = #S.name + 1|0;
16:     if ((#S.name|0) >= #S.length) break;
17:     if !##filter(#S) continue;
18:     ##hash_index_insert(#S.name, #onCol); '''
19: }
```

Figure 3.4: Code generation template for joins.

At a finer level of detail than the generic physical plan, Figure 3.4 shows the code generation template for joins. We start by describing the convention behind the code template, which depends on multi-line strings with substitutions. Triple quotes (`'''`) denote the start and the end of multiline strings, for instance, the string starting in line 2 ends in line 12. A single pound sign (#) denotes a variable substitution, for instance, if `R.iterator = l_rid` the string "`#R.iterator|0`" evaluates to "`l_rid|0`" after substituting `#R.iterator` with the value of the variable. Finally, the double pound sign (`##`) denotes a function call substitution, for instance, the string "`##filter(#R)`" is replaced with the return value of the function filter after passing the value `#R` as a parameter to the function.

The function `build_join`, shown at line 1, is called after validating the query to generate the code equivalent to our example join query, which simply returns a multi-line string that starts in line 2. Line 3, contains a call to the function `build_hash_index`, which is responsible for generating the code of the entire first phase (building the hash index). The `build_hash_index` function requires two parameters which are the name of the relation (`S`) and the name of the column to index (`onCol`).

Lines 4–12 are responsible for the second phase (probing the hash index and generating the output). The second phase iterates from 0 up to the size of relation R. Line 6 calls the function filter to generate the code for all the filters that are only associated with table R. For concreteness, Figure 3.5 shows the output code for the second phase—for ease of exposition, we use alphabetical letters to number the lines of the output code (not to be confused with the line numbers of the code template). Since the query does not have any filter predicates over lineitem, lines a-b control a loop over the l_orderkey from 0 up to the number of records in the lineitem. The variable name l_rid is used to maintain such a record identifier which is then translated into an array index with the relevant data type to access the values of the column as described in Section 3.3.2.

Line 7 generates the code responsible for probing the hash table and iterating over possibly multiple matching values. This line calls the function join_probe which is responsible for generating the lines c-r in the output code. Lines c-d compute a hash key of the values in the l_orderkey column, which are then used to look up record identifiers of matching records in o_orderkey. Lines f-r are responsible for probing the hash table over the o_orderkey column. Matching hash keys do not guarantee that the values are equal since we use a hash function that allows for false positives. Thus, lines s-u use the record identifiers l_rid and o_rid retrieved by probing the hash table to compute memory addresses in order to compare the actual values. Line 8 calls the function post_join_filter which generates the code responsible for complex filtering conditions that involve both columns from lineitem and orders. Line 9 calls the function generate_record which generates a record (or computes an aggregate function). In our example, line v uses l_rid to compute the memory address of the associated l_extendedprice in order to compute the sum.

Key Design Decisions:

Copy-free joins and group by code templates. Afterburner avoids materialization of intermediate results as much as possible in order to fit inside a web browser memory. For instance, it only stores record identifiers in hash tables instead of copying the values to minimize the memory footprint of the hash-based joins and group by operators. Afterburner will translate record identifiers into array indices to access the column values in a lazy fashion. An alternative design that stores record values directly in the hash tables can provide faster performance during the probe phase by avoiding any extra overheads, such as array index translation and random memory access, required to retrieve the values. On the other hand, Afterburner's lazy approach not only requires less memory but also avoids copying values.

Maintaining memory for hash tables. In Afterburner, hash tables use chaining for collision resolution, where the chains are allocated as unrolled linked lists in another memory segment. Since we do not have access to operating system calls such as malloc and free, our system

```
a: while(1){l_rid = l_rid + 1|0;
b:   if ((l_rid|0) >= 6000000) break;
c:   hkey = ((((hI32[(oOffset + (l_rid<<2)) >> 2]|0)));
d:   hkey = hkey & (hashBitFilter|0))|0;
e:   bckt = -1; curr = 0;
f:   if(hI8[(h1db + (hkey >> 3))|0] & (1 << (hkey & 7)))
g:     bckt = hI32[((h1bb + (hkey << 2))|0) >> 2]|0;
h:   while(((bckt|0) > 0)){
i:     if ((curr|0) >= (hI32[bckt >> 2]|0)){
j:       if (bckt = hI32[(bckt + (((h1Size + 1)|0) << 2)|0) >> 2]|0){
k:         curr = 1;
l:         o_rid = hI32[((bckt + (curr << 2))|0) >> 2]|0;
m:       } else
n:           break;
o:     } else {
p:         curr = curr + 1|0;
q:         o_rid = hI32[((bckt + (curr << 2))|0) >> 2]|0;
r:     }
s:     if (!((+((hI32[((oOffset + (l_rid << 2))|0) >> 2]|0)|0))
t:         == (+((hI32[((iOffset + (o_rid << 2))|0) >> 2]|0)|0))));
u:         continue;
v:      sum1 = sum1 + (+(hF32[((8196588 + (l_rid << 2))|0) >> 2]));
w:   }
x: }
```

Figure 3.5: Asm.js code generated by the join template.

must handle all aspects of memory management itself. For instance, Afterburner has designated array segments reserved for the hash tables which are to be used during query processing. In our example compiled query, the variable `h1bb` shown in line `g` holds the starting array index of the segment. At the beginning of a new query execution, the generated code must reset the hash table segment, i.e., removes any previously inserted keys. In our experiments, we have found that resetting hash tables segments takes substantial time, for instance setting the value of 2.6 million keys (2.6 million * 4 byte integers = 10 MB of memory) takes on average 9 ms on our clients which is an unacceptable overhead per operator usage—which can be used multiple times per query.

To minimize the overhead associated with resetting the hash table after each use, we maintain an array of bits that tracks the state of each hash table key. For instance, a hash table that maintains 2.6 million keys requires a bit array of only 300 KB which takes less than 1 ms to

Data type	**Filter**	**Selectivity**	**Column Name**
Integer	`== 0`	`1.0`	`o_shippriority`
Float	`> 555.5`	`1.0`	`o_totalprice`
String	`== '1-URGENT'`	0.2	`o_orderpriority`

Table 3.1: Filter predicates in microbenchmarks.

set to 0. To illustrate this, we consider the generated code shown in Figure 3.5. Line `f` checks whether the bit associated with holding the keys is set to 1 or 0 before checking whether the key exists in the hash table. Only, when the associated bit is set to 1, does the system considers the keys to be valid. This allows hash-based operators to reuse memory segments with little overheads.

Another benefit of the bit array is when using the hash table for group by operators on sparse columns (columns with a small number of unique values relative to the cardinality of the relation, for example, region identifiers). After the insertion phase is complete, the generated code first scans the keys stored in the hash table to check for unique values (i.e., the groups). Scanning the bit array can uncover the inserted keys at a fraction of the time required to scan the entire array segment that stores the actual keys. For dense columns (columns with a large number of unique values relative to the size of the relation), checking the bit array does not add much overhead.

3.4 Evaluation

Experimental validation of our work is divided into two parts. First, we conducted microbenchmarks to understand the performance characteristics of the in-browser execution environment and how it compares to native code execution. Second, we examined the performance of Afterburner on end-to-end SQL analytics within the browser, using the TPC-H benchmark. We compared our system against MonetDB, a well-known column store, and LegoBase, the state of the art in compiled queries for SQL.

3.4.1 Microbenchmarks

In our microbenchmarks, we focused on very simple filter queries that count the number of records matching a predicate on a particular column. We vary the data type of the filtering attribute to integer, float, and string. Table 3.1 shows the names of the columns along with the filters and their selectivity. For these experiments, we scanned the `orders` table from TPC-H data at a scale of 10 GB (15 million records). We examined the following hand-written programs in JavaScript:

- JavaScript without asm.js or typed arrays, which we refer to as J1 for convenience.
- JavaScript without asm.js but with typed arrays, which we refer to as J2 for convenience.
- JavaScript with both asm.js and typed arrays, which we refer to as J3. This takes advantage of all JavaScript optimizations that we described in Section 3.3.

We compared the above conditions against hand-written C++ programs, compiled under the following conditions:

- GCC (g++ 5.4.0) using optimization level `-O3`, which we refer to as GCC-O3 for convenience.
- GCC, but without any optimizations.
- Clang (v 3.9.0), with optimization level `-O3`, which we refer to as Clang-O3 for convenience.
- Clang, but without any optimizations.

Experiments were run on a desktop with a 2.7 GHz Intel i5-5250U processor (4 cores, 3 MB of cache) and 8 GB of RAM, running Ubuntu 16. The JavaScript conditions ran in Mozilla Firefox (v50). For these experiments, we exclude the compilation times for C++ and asm.js. We run each condition five times for warmup runs and report the average runtime for the next five runs. Note warm runs allow the J1 condition to benefit from JIT code caching. We use Firefox for our evaluations since it is a widely adopted open-source browser. In addition, Firefox supports the three JavaScript conditions that we evaluate. In principle, we expect similar performance results for the same JavaScript conditions on any recent and widely used browser. For example, we expect similar performance for conditions J1 and J2 using Google Chrome (within $\pm$10–20% from Firefox).

Results are shown in Table 3.2, where we report query latencies in milliseconds and slowdown with respect to GCC-O3 in parentheses. Comparing GCC-O3 with fully-optimized JavaScript (J3), we see a slowdown of roughly four to five times for filter predicates over integers and floats, but only slowdown of 10% for strings. Detailed analyses show that the performance advantage of GCC-O3 comes from automatic loop unrolling and the generation of SIMD instructions. On the other hand, strings are only marginally slower with J3 because of the extra level of indirection associated with a standard `(char **)` representation of strings in C++, where the query latency is dominated by cache latencies associated with pointer dereferencing. We see the

Condition	**Integer**		**Float**		**String**	
GCC-O3	4.4		4.4		56.4	
GCC	37.0	(8.4×)	47.8	(10.9×)	103.8	(1.8×)
J3	16.9	(3.8×)	22.7	(5.2×)	60.8	(1.1×)
J2	20.1	(4.6×)	25.7	(5.8×)	94.7	(1.7×)
J1	28.6	(6.5×)	28.5	(6.5×)	406.1	(7.2×)
Clang-O3	3.8		4.0		60.4	
Clang	40.4		45.4		97.0	

Table 3.2: Microbenchmarks showing query latencies (ms) under various conditions.

	branches		**mispredicts**		**D1 misses**		**I1 misses**	
Data type	**GCC-O3**	**J3**	**GCC-O3**	**J3**	**GCC-O3**	**J3**	**GCC-O3**	**J3**
Integer	3.7m	45m	37	7600	940k	970k	640	12000
Float	3.8m	45m	26	3400	940k	950k	520	12000
String	33.0m	51m	3.2m	3.3m	3.6m	3.6m	7700	21000

Table 3.3: Performance counters comparing GCC-O3 and J3 conditions.

effectiveness of SIMD instructions and loop unrolling with GCC (without any optimizations), which is actually slower than fully-optimized JavaScript. Comparing J3 with J2 (typed arrays but no asm.js) and J1 (no optimizations), it is clear that both features of JavaScript contribute to performance and in a cumulative fashion. Finally, our results show that Clang performance is on par with GCC performance. This is a nice sanity check as one of the comparison conditions we present in the next section uses Clang.

In Table 3.3, we show the output of CPU performance counters comparing GCC-O3 and J3, using the `perf-stat` profiling tool. Measuring such low-level performance is for GCC-O3 is straightforward since the code runs as a separate process. On the other hand, measuring CPU performance for J3 running inside the browser (Firefox) is challenging because J3 runs as a thread inside the browser's main process. Thus, even in an idle state, without user interactions, the browser process is active (e.g., running event loops) and generating data captured as part of profiling. To minimize such interference, in our measurements we took the following steps: First, we repeated each run 500 times and report the average. Second, we minimized the browser window, leaving only the JavaScript console available to run our benchmark scripts. This helps to avoid spurious interactions.

Examining the number of branches: for GCC-O3, which uses SIMD instructions and is able to process 4 integers or floats at a time, we observe an average 0.25 branches per record, as expected (in these microbenchmarks we are scanning 15 million records). In comparison, J3 averages 3 branches per record, which explains the latency gap. Note that using SIMD is a compiler optimization, and so it is certainly possible that future JavaScript runtimes will be smart enough to take advantage of SIMD instructions also. Due to loop unrolling for integers and floats, there are barely any branch mispredicts for GCC-O3. In terms of data cache misses for integers and floats, we observe roughly the same counts since typed arrays are implemented using native arrays. For instruction cache misses, we see lower counts for GCC-O3 since SIMD code is far more compact. Note, however, that when it comes to strings, the SIMD instructions are no longer applicable, and we observe similar counts for branch mispredicts. GCC-O3 generates code with fewer branches, but latency is dominated by branch mispredicts, which are comparable between GCC-O3 and J3.

In summary, these microbenchmarks show that the JavaScript execution environment inside modern browsers is very efficient, in some cases rivaling native performance. This is not surprising: the prevalence of complex JavaScript applications (e.g., Gmail, Facebook, etc.) in running modern websites means that countless hours have been devoted to optimizing JavaScript performance—we are the beneficiaries of all this effort. Native code is still faster on our microbenchmarks because of SIMD instructions and loop unrolling, but these optimizations may not be possible on a more complex code.

3.4.2 In-Browser Analytics Performance

Our next set of experiments focus on the performance of Afterburner as a stand-alone analytical RDBMS running in the browser. Here we use the TPC-H benchmark for decision support in data warehousing at a scale factor of 1 GB, which corresponds to 6 million records in `lineitem`. This represents a rough upper bound on the amount of data that a commodity desktop or laptop can comfortably hold today. Experiments were performed on the same client machine as in the previous section. We compared Afterburner against two different systems:

MonetDB (v11.23.13) is an open-source analytical RDBMS that takes advantage of columnar storage and vectorized execution. We used the TPC-H test harness written by the developers of the system. For a fair comparison, MonetDB was configured to use a single thread, since code running inside a browser tab is single threaded.

LegoBase [36, 51][2] is an open-source in-memory RDBMS that represents the state of the art in code generation. LegoBase takes as input a representation of a physical plan of a query then generates code for this plan. LegoBase uses its knowledge of the target programming language

[2]The term LegoBase is a bit ambiguous since it refers to several different software components, but in this context we specifically refer to the authors' SIGMOD 2016 paper that we replicate and compare against.

and database statistics to pick the best physical operators for a query (expressed using generated code). In our experiments, we configured LegoBase to target C. We also configured LegoBase to generate code under what the authors call the "compliant" condition which generates query operators that are compliant with the TPC-H benchmark. We measured the time to generate the target C code (`codegen`), time to compile the target code using Clang (`compile`), and query execution latency (which does not include `codegen` and `compile` stages). For all experiments, we used the open-source code available at https://github.com/epfldata/dblab and we sought guidance from the co-authors of the paper to ensure that we were using their system properly. Our runs are generally consistent with the results reported in their papers.

Our specific rationale for comparing Afterburner to these systems: MonetDB is a mature and stable implementation of well-known techniques for analytical data processing. On the other hand, LegoBase represents the "latest and greatest" research on compiled queries. It is best described as a research toolkit comprised of many difference pieces as opposed to a complete RDBMS.

All our measurements were on a warm cache—we first ran each query five times, and then took measurements over the next five trials. For LegoBase, we warmed up the codegen and query compilation in the same way. For Afterburner, the measured latency includes query compilation overhead and all data are explicitly loaded in-memory. For MonetDB, all data are cached in the underlying OS buffer caches.

Table 3.4 shows the latency of all 22 TPC-H benchmark queries comparing Afterburner, MonetDB, and LegoBase. For Q7 and Q9, LegoBase runs out of available physical memory on our client desktop, and thus we show total*, which totals the latencies of all queries except for those queries. For the 20 queries (total*) LegoBase finishes all queries in 3.7 s, MonetDB finishes in 4.4 seconds, while Afterburner finishes in 5.7 s, which is only 1.6$\times$ slower than LegoBase. It is important to note that the latencies for LegoBase do not include code generation or compilation, while for Afterburner we show end-to-end execution latency, which includes query compilation. For all queries MonetDB finishes in 4.8 s, while Afterburner finishes all queries in 6.4 s (only 33% slower).

To be fair, LegoBase techniques may not have been optimized for interactive query exploration since both the codegen and compilation stages *individually* are longer than end-to-end execution with Afterburner. LegoBase is perhaps more suitable for even larger data conditions, where the time spent in codegen and compilation can be amortized over a longer query execution time.

Overall, LegoBase is faster than Afterburner in 13 queries and MonetDB is faster in 11 out of the 22 queries. These results show the feasibility of running an analytical RDBMS completely inside the browser. We expected LegoBase to perform faster than MonetDB for two main reasons. First, Legobase, is a more recent engine that includes several optimization for analytical

SQL queries. Second, MonetDB is optimized for vector based processing which can benefit greatly from multicore CPU usually found on backend servers—recall in our experiments we only restricted MonetDB to use a single core. We also expected LegoBase and MonetDB to perform faster than Afterburner since they are both native techniques. However, the overall expected performance does not prevail in every single considered query, e.g., there is no single engine that performs faster than the others for all the queries in our evaluations. Code generation engines (Afterburner and Legobase) perform significantly better than MonetDB in Q1 since MonetDB's query optimizer picks a plan that depends on materializing the base columns, which is typical for vector based SQL engines. Afterburner does worse in comparison to MonetDB in Q17. We associate this to the limited set of operators in Afterburner that can lead to suboptimal plans that might require additional steps to achieve. In our prototype, we overcome these limitations by materializing parts of the subplan as temporary tables, which can be less efficient than using pipe-lining.

In comparison to Legobase, Afterburner does worse in Q19 because LegoBase generates code which the Clang compiler is able to optimize better against the CPU. Q19 has many complex filter conditions; however, LegoBase expresses these complex filters using not-branching statements. Short-circuited statements might have the benefit of avoiding some processing and data access, however, they are harder to optimize at the CPU level. Such optimizations are not currently available in asm.js which explains the performance gap.

We are, of course, not the first to explore data management inside the browser. As a final set of experiments, we compared Afterburner against two existing SQL-JavaScript solutions:

Lovefield[3] is a relational database for web apps written by Google that uses IndexedDB as a storage layer. To compare Afterburner with Lovefield, we generated an instance of the `lineitem` table using the TPC-H data generator and incrementally ingested data into the system. We were not able to store more than 5000 records before the browser tab crashed. On this data, we attempted to run a TPC-H query. We could not fully express TPC-H Q1 because it does not support complex aggregate such as `sum(l_extendedprice * (1 - l_discount))`. Thus, we removed all unsupported expressions and ran the query, which took 30 ms. On the same query, Afterburner takes 11 ms. It is important to note that even with Afterburner constant overheads per query (e.g., query compilation, asm.js compilation, and memory setting), Afterburner is still much faster on this tiny data set.

Sql.js[4] is a cross-compiled version of SQLite (v3) into asm.js. In our experiments, we were not able to load a TPC-H database of scale factor 1GB because into Sql.js because the database was too large, which caused the browser process to terminate. Thus, we have tried to load the smaller

[3]https://github.com/google/lovefield
[4]https://github.com/kripken/sql.js/

scales of the databases at 10% increments. We only managed to ingest 600k records from the `lineitem` table. For Q1 on this data scale, sql.js takes 3.6 s to execute, compared to 22 ms for Afterburner. We were unable to test more complex queries for instances the ones that involve multiple joins since they took too long (more than one minute per query).

Our experiments show that current in-browser SQL engines such as Lovefield and Sql.js do not support analytical queries for modestly-sized databases in speed that is comparable to the state of the art native techniques. In fact, Sql.js takes 3.6 s to execute TPC-H Q1 which approaches the total execution time of the entire TPC-H queries for the native engines which Afterburner rivals.

In summary, these experiments show that it is possible to build a high-performance analytical RDBMS in JavaScript. While its performance still lags MonetDB and LegoBase, both of which run natively, we find Afterburner's performance quite impressive, considering that it runs *completely in the browser.*

	Afterburner	MonetDB	LegoBase		
			codgen	compile	execute
Q01	115 (8.0)	1269 (10.6)	159 (35.4)	216 (0.6)	57 (0.5)
Q02	68 (8.2)	36 (3.6)	431 (100.9)	351 (3.5)	27 (0.6)
Q03	108 (4.4)	242 (80.7)	328 (89.1)	333 (1.4)	120 (5.5)
Q04	161 (2.1)	121 (2.9)	177 (74)	266 (4.5)	163 (13.5)
Q05	151 (1.6)	141 (6.1)	597 (99.6)	501 (2.9)	73 (2.9)
Q06	36 (0.4)	431 (25.7)	67 (24.9)	161 (11.4)	21 (0.5)
Q07	322 (11.3)	161 (3.0)	510 (62.6)	525 (3.9)	▨
Q08	101 (2.9)	113 (3.0)	1039 (66.3)	612 (3.9)	306 (20.4)
Q09	314 (2.3)	235 (2.1)	710 (94.6)	569 (26.4)	▨
Q10	233 (6.4)	126 (2.5)	382 (21.7)	350 (2.6)	336 (2.0)
Q11	35 (6.0)	42 (1.8)	237 (14.6)	297 (8.8)	14 (0)
Q12	82 (1.4)	119 (2.7)	192 (11)	267 (1.1)	116 (10.4)
Q13	830 (2.4)	243 (4.9)	131 (11.1)	194 (1.4)	56 (0)
Q14	44 (3.5)	40 (0.6)	123 (9.6)	218 (18.1)	25 (1.0)
Q15	50 (1.0)	74 (1.5)	106 (6.8)	213 (1.1)	49 (0.4)
Q16	149 (3.2)	316 (7.0)	433 (18.1)	449 (4.9)	207 (7.4)
Q17	718 (1.9)	185 (1.6)	156 (7.2)	271 (1.3)	278 (15.8)
Q18	446 (69.6)	171 (1.2)	213 (11.1)	261 (1.7)	414 (0.8)
Q19	1951 (62.0)	209 (1.6)	175 (21.8)	288 (9.3)	175 (4.6)
Q20	62 (0.6)	86 (2.7)	436 (20.6)	463 (50.2)	39 (1.4)
Q21	284 (2.3)	369 (10.4)	371 (14.9)	395 (40.4)	1112 (5.2)
Q22	95 (1.1)	118 (11.2)	241 (28.7)	210 (0.9)	68 (0.5)
total*	5719	4451	5994	6316	3655
total	6355	4847	7214	7410	▨

Table 3.4: Query latency (ms) for TPC-H queries, comparing Afterburner with MonetDB and LegoBase. Query latencies are over five trials with 95% confidence intervals (shown between parentheses).

Chapter 4

Interpretable and Informative Explanations of Outcomes

4.1 Introduction

As we have discussed in Chapter 1, in this dissertation we identify and focus two main challenges that face non-technical users who want to analyze data. In the previous chapter, we have discussed how our proposed in-browser SQL engine can solve the infrastructure problem for moderately sized data sets—which is the first challenge.

In this chapter, we turn to the second challenge, namely guiding users through the task of exploring a data cube. We apply an information-theoretic technique, *Explanation Tables*[1] [16], to address this challenge. In addition, Explanation Tables are also a good data summarization tool; being immediately interpretable, Explanation Tables reveal high-level insights that can familiarize users with new data sets.

In Section 1.2.3, we have outlined a data exploration task for the data cube of the `Salary` data set, using our proposed tool which does not require knowledge of SQL. The exploration tool combined with our proposed portable SQL engine (Afterburner) allows everyone to navigate the data cube inside the browser. The goal of the outlined data analysis task is to explore a binary outcome attribute, such as `HighSalary`. Recall, this binary outcome attribute was derived

[1] Explanation Tables is a joint work with Parag Agrawal, Lukasz Golab, Flip Korn, and Divesh Srivastava that was recently published in [16]. In particular, Sections 4.2, 4.3.3, and 4.3.4 describe existing work on which this thesis builds. Thus, the novel contribution of this chapter is an evaluation of efficiency and effectiveness of the proposed optimizations compared to related techniques. In addition, proposing and evaluating in-browser techniques for generating Explanation Tables.

from the Salary attribute, such that `HighSalary` is set to 1 (or Yes) when the Salary attribute is greater than a certain threshold decided by the user (150000 in our example) and is set to 0 (or No) otherwise. However, this does not solve the second challenge which is the large number of options to explore—since the number of patterns in the data cube is large. Data cube exploration tasks tend to be ad-hoc and exploratory, i.e., based on the data cube navigation operators, such as roll-up and drill-down. To guide users through data exploration tasks, we apply an information-theoretic approach that picks the most informative patterns in the data cube of the data set and presents them to the user. These patterns, which are expressed in terms of the actual values in the data, not only constitute a high-level summary of the outcome attribute but also acts as a good "short list" of suggested data exploration starting points. At a high-level, the most informative patterns are likely to be the most important ones as discussed in Chapter 1.

Explanation Tables. To be easy to use, Explanation Tables must satisfy three conditions:

- First, we require that Explanation Tables be informative, i.e., given the values of the dimension attributes of the data set, an explanation table should enable the recovery of the outcome attribute with measurable accuracy.
- Second, they must be interpretable, i.e., able to express insights in terms of meaningful groups (dimension attributes) of records. Groups often overlap in real relational data sets, e.g., the 'Health' ministry and the 'Manager' position title. Thus, we require allowing overlapping patterns to allow for picking the most informative factors affecting the outcome attribute (without restricting the Explanation Tables).
- Finally, we desire that Explanation Tables be efficient to generate for modestly sized data sets.

Informativeness. Intuitively, the more informative the pattern, the better it can be used to recover the values of the outcome attribute accurately. To illustrate the information content of a pattern, consider the following three patterns from the data cube of `Salary`: x_1 = (*, *, *, Manager Zone 2) which has an AVG(`HighSalary`) of 0 and count of 1754, x_2 = (*, *, *, Engineering & Related Level 2) which has an AVG(`HighSalary`) of 0 and count of 626, x_3 = (*, 2015, *, Manager Zone 2) which has AVG(`HighSalary`) of 0 and count of 1014. Then suppose that the file that stores the outcome attribute `HighSalary` is lost. The information content of the three patterns is related to how much they can be used to recover the values in the `HighSalary` attribute with measurable accuracy. Using the first pattern (x_1), we can determine that none of the records with `PositionClass = Manager Zone 2` have a high salary (otherwise AVG(`HighSalary`) for x_1 would be greater than 0), which means that this pattern can recover 1754 values. Similarly, patterns x_2 and x_3 can recover 626 and 1014 values respectively. Thus, we can conclude that x_1 is the most informative, then x_3 followed by x_2.

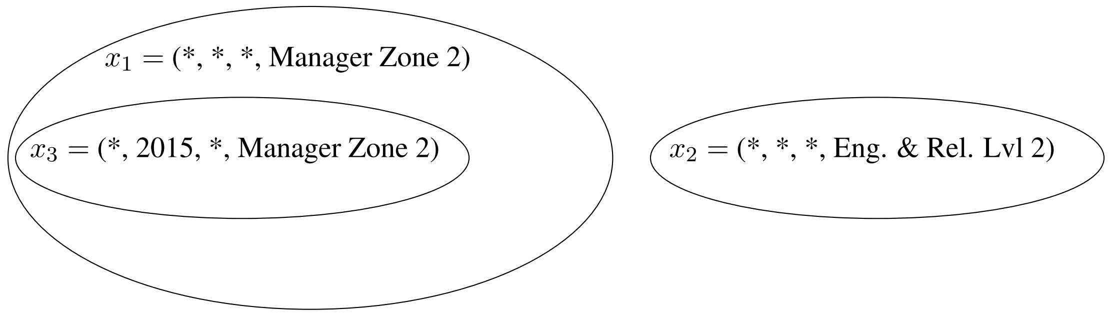

x_1 = (*, *, *, Manager Zone 2)
x_3 = (*, 2015, *, Manager Zone 2)
x_2 = (*, *, *, Eng. & Rel. Lvl 2)

Figure 4.1: Relationship among patterns x_1, x_2, and x_3.

Our previous example illustrates the information content of single patterns. On the other hand, evaluating the information content of a set of patterns is more intricate because the patterns in the data cube usually intersect, i.e., cover common records. To illustrate this, consider the information content of two sets of patterns $T_1 = \{x_1, x_2\}$ and $T_2 = \{x_1, x_3\}$. The first set of patterns (T_1) can be used to recover 2380 records (1754 for pattern x_1 and 626 for pattern x_2) while the second set of patterns (T_2) can only be used to recover 1754 since all the patterns covered by x_3 are already covered by x_1. Figure 4.1 illustrates the relationship among the patterns in this example. Notice that we had to consider the relationship between the records covered by the patterns in order to decide their information content. Notice that in this example we can depend only on the "syntactic" forms of patterns x_1, x_2, and x_3 in order to figure the relationship among them. For instance, we do not need to examine the database in order to identify that all the records that have position class 'Manager Zone 2' in the year '2015' are a subset of the records with the position class 'Manager Zone 2'. Conversely, the records that have position class 'Manager Zone 2' are different from the records that have position 'class Eng. & Rel. Lvl 2'.

As we have illustrated, the relationships among overlapping patterns in the data cube affect their information content. Thus, allowing overlapping patterns, in Explanation Tables, makes the problem more challenging. In our previous example, we were able to identify that pattern x_1 contains pattern x_3 just by inspecting the patterns—and not the data. We now consider other examples where this is not possible. Consider the position title 'Crown Prosecutor' and the ministry of 'Justice and Solicitor Generals'. Without any previous knowledge about the ministry and the position title, we cannot determine the relationship between them. Thus, we must examine the records in the data set in order to identify the relationship between the pattern $x_4 =$ (Justice and Solicitor General, *, *, *) and the records in pattern $x_5 =$ (*, *, Crown Prosecutor, *), i.e., x_4 and x_5 are overlapping patterns. The position title Crown Prosecutor only appears in the records that are in the Justice and Solicitor General ministry, i.e., the pattern x_4 contains all the records in pattern x_5 similar to patterns x_1 and x_3. However, in this case, the relationship between x_4

and x_5 cannot be determined without examining the records in the data set. In other instances, overlapping patterns may not intersect at all or have a high degree of correlation. For instance, the title Barrister and Solicitor appears in the `Salary` data set 758 times, 715 of which (94% of the time) in the ministry of Justice and Solicitor and 43 of which in the ministry of Human Services. Such cases of partial overlap can also affect the information content of patterns and are common in real-world data sets.

Allowing overlapping patterns brings two challenges. First, they increase the complexity of generating Explanation Tables since a wider class of patterns is to be considered. Second, determining the information gain of overlapping patterns is more challenging. However, Explanation Tables with overlapping patterns can potentially be more informative. Recall Table 1.5, the third pattern overlaps with the second, fourth and fifth. Replacing the third patterns with another (that does not overlap) will yield an explanation table that is less informative. To our knowledge, existing techniques that can be adapted to generate Explanation Tables do not generate overlapping patterns.

Typically, the space of options to navigate (patterns in the data cube) is too large for the users. For instance, the `Salary` data set has 18023 records while the data cube has 80495. The large size of the data cube is challenging because it can easily overwhelm the human user. Discovering the most important factors affecting having high salary by navigating the data cube of `Salary` data set (manually) is hard without a tool like Explanation Tables that can prioritize the most important patterns for the user.

Interpretability. As discussed in Section 1.3, the non-technical user may not have the knowledge required to perform complex analyses or apply decision support techniques, yet he wants to explore or familiarize himself with the data. Thus, immediate interpretability becomes even more useful when non-technical users want to summarize, explore, or gain familiarity with a new data set.

The toolkit of the data scientist spans a large suite of tools with varying degrees of complexity for performing such tasks. Data scientists can issue a simple query over the relational data or use more complex decision support techniques, such as a Decision Trees. There are several existing informative techniques, which generate summaries or models that can be used to recover the outcome attribute given the dimension attributes. However, the output models of many of these algorithms are not immediately interpretable. For instance, classification techniques that depend on combining the output of multiple classification algorithms or models have been known to be informative, i.e., given the dimension attributes, their models can recover the outcome attribute with good accuracy [15]. However, these models are not useful for non-technical users. Even though, some of these models generate human readable or graphically representable models (e.g., a group of trees), we do not consider these models immediately interpretable since the user is required to know a great deal about how these algorithms work and how these models were generated (e.g., knowing which algorithm specific configurations were applied) in order to value them.

Explanation Tables are informative summaries of outcome attributes that are readily interpretable because they are expressed in terms of patterns of the categorical attributes of the relational table. Categorical patterns are immediately interpretable because they usually describe groups of records, hence, refer to real-life groups. Explanation Tables' categorical patterns also identify good starting points for a data cube exploration task since that they prioritize the most informative patterns in the data cube. The main premise is that while the idea behind Explanation Tables depends on an information-theoretic framework, its users are not required to know information theory in order to interpret them.

4.1.1 Challenges and Contributions

Many of the existing techniques that can be adapted to our problem either address the information content or interpretability, but not both. For instance, classification algorithms such as Decision Trees are informative, but the patterns they encode do not overlap, and therefore they are not immediately interpretable by non-experts since interpreting Decision Trees requires some knowledge of how they are built. On the other hand, there are various database summarization techniques that generate overlapping patterns and are interpretable [23, 24, 39]. However, these techniques optimize for different objectives instead of the information content, such as the minimum number of patterns covering all the records with outcome 1 using the fewest patterns [23, 24].

This chapter evaluates efficient ways for generating Explanation Tables that are both interpretable and informative. Explanation Tables employ a classical information-theoretic approach to select patterns that provide the most information gain about the distribution of the outcome attribute. A similar approach was used in previous work on data cube exploration, but overlapping patterns were not supported [49]. This restriction simplifies the problem since selecting a pattern rules out all of the patterns it intersects with. Allowing overlapping patterns while maintaining efficiency makes Explanation Tables a better fit for guiding data exploration tasks compared to existing techniques.

In particular, the challenge lies in the size of the search space of overlapping patterns which is the size of the data cube of the data set. Thus, greedily selecting the most informative patterns is challenging even for smaller data sets. This is because the number of patterns in the data cube tends to grow quickly as the number of attributes increase.

One common way to address the efficiency of a technique is to apply it to a subset of the data, i.e., use sampling. In our evaluations, straightforward sampling was not effective for generating

Explanation Tables (we discuss this further in Section 4.3.2). Thus, we evaluate solutions that make better use of sampling to generate Explanation Tables. Namely, *Flashlight*, which is a technique that efficiently only considers candidate patterns generated from a sample and the other, *Laserlight* that is a more aggressive heuristic on a given sample but needs to use larger samples to be effective.

The goal of this chapter is to allow for running Explanation Tables to run in the browser in order to guide data exploration tasks without any external dependencies. Even with the significant efficiency our evaluations show that Flashlight provides, it is still challenging to generate Explanation Tables in the browser. This is mainly because of the slow performance of JavaScript and the limited resources available in the browser. Thus, in this chapter, we introduce a new data structure, which we call *Mnemonic Table* that adapts Explanation Tables to the browser. Our evaluations show the viability of generating Explanation Tables in the browser.

Contributions. The specific contributions of this chapter are as follows:

- Evaluate the efficiency and effectiveness of Explanation Tables compared to related techniques.
- Compare the quality of sample-based techniques to the baseline algorithm which considers every pattern in the data cube. Our evaluations show that Flashlight produces Explanation Tables that are very close to the baseline in terms of quality but only require a small fraction of the running time.
- Present an optimized operator (Mnemonic) which adapts Explanation Tables to the browser. Our evaluations show the viability of guiding data exploration scenarios using information theory in the browser without any external dependencies.

The remainder of this chapter is organized into three parts. First, we present the necessary background to define Explanation Tables in Sections 4.2 and 4.3. Second, we present our evaluation of Explanation Tables in Section 4.4. Third, Section 4.5 presents and evaluates our in-browser implementation of Explanation Tables.

4.2 Background

This section presents the necessary background to define Explanation Tables [16]. Table 4.1 lists the symbols used in this chapter.

Sybmbol	Explanation
D	Relational table
s	A uniform sample of D
A_i	A dimension attribute of D
d	Number of attributes in D (not including the outcome attribute)
$\mathcal{X}$	Set of all possible patterns (the data cube)
t	A record from D
x	A pattern
$\asymp$	Matches, e.g., $(a_1, *) \asymp (a_1, b_1)$
$y(t)$	The value of the binary outcome attribute associated to record t
$\varepsilon^*(t)$	A maximum entropy estimate of $y(t)$ given by explanation table T
$S_D(x)$	Support set of x over D
$Y_D(x)$	Average value of the binary outcome attribute y for records matching x
$E_D(x)$	Average estimated value of outcome y for records matching x based on ε^*

Table 4.1: Symbols used in this chapter.

4.2.1 Example Data Set – `Workout` Relation

To illustrate the intuition behind the information-theoretic framework Explanation Tables follows, consider Table 4.2, which shows an athlete's workout log, with each record containing an id, the time and day of the week of the workout, the meal eaten before the workout, and an outcome attribute indicating the outcome of the workout, i.e., whether a target goal was met. Table 4.3 illustrates a corresponding explanation table. It contains a list of patterns that specify groups of records; patterns consist of attribute values or the wildcard symbol "*" that denotes all possible values. Each pattern is associated with the number of records it matches (COUNT(*)) and the fraction indicating the portion of the records having the outcome attribute equal to Yes (or 1 or True). The first pattern in the explanation table matches all the records in Table 4.2 and indicates that, on average, half the workouts were successful. The second pattern reveals that all Saturday workouts were unsuccessful, while the last two patterns state that eating bananas or oatmeal led to successful workouts 75% of the time.

4.2.2 Definitions

Consider a relational database D with a schema of the form $[A_1, A_2, ..., A_d, y]$, that is a list of dimension attributes and y which is a binary outcome attribute. A pattern x is expressed

id	**day**	**time**	**food**	**goal met?**
1	Fri	Dawn	Banana	Yes
2	Fri	Night	Salad	Yes
3	Sun	Dusk	Oatmeal	Yes
4	Sun	Morning	Banana	Yes
5	Mon	Afternoon	Oatmeal	Yes
6	Mon	Midday	Banana	Yes
7	Tue	Morning	Salad	No
8	Wed	Night	Burgers	No
9	Thu	Dawn	Oatmeal	Yes
10	Sat	Afternoon	Nuts	No
11	Sat	Dawn	Banana	No
12	Sat	Dawn	Oatmeal	No
13	Sat	Dusk	Rice	No
14	Sat	Midday	Toast	No

Table 4.2: The `Workout` database.

in terms of the dimension attributes of D and is a member of its data cube ($\mathcal{X}(D)$), where $\mathcal{X}(D) \subseteq (dom(A_1) \cup \{*\}) \times \cdots \times (dom(A_d) \cup \{*\})$, and $dom(A_i)$ is the domain of attribute A_i. A pattern x will *match* a record t if for every attribute A_i, either $x[A_i] =$ '*' or $t[A_i] = x[A_i]$ and we use the operator "$\asymp$" to denote this match ($t \asymp x$). Thus, each pattern in $\mathcal{X}(D)$ matches ($\asymp$) at least one record in D. In this chapter, we will use D, database, or data set to refer to the relational database.

Let the *support set* of pattern x, which is $S_D(x)$, be the set of records in D that match x, i.e., $S_D(x) = \{t \in D : t \asymp x\}$ and the *support* (count) of x be $|S_D(x)|$. We denote the fraction of records where y equals 1 for records matching x as $Y_D(x)$, i.e., $Y_D(x) = \frac{1}{|S_D(x)|} \sum_{t \asymp x} y(t)$. An explanation table T is a summary of the outcome attribute of interest y expressed as a list of patterns. Each pattern in T has an associated fraction of matching records that the outcome attribute y has a value of 1 ($Y_D(x)$) in addition the support of the pattern ($|S_D(x)|$). Thus, an explanation table is a list of rows in the following format: $x, Y_D(x), |S_D(x)|$. In this chapter, we will use the term pattern to refer to an Explanation Table's row when the context is clear.

day	time	food	goal met?	COUNT(*)
*	*	*	.5	14
Sat	*	*	0	5
*	*	Banana	.75	4
*	*	Oatmeal	.75	4

Table 4.3: An explanation table over the `Workout` database.

4.2.3 Maximum Entropy Principle

The patterns of an explanation table have summarized values of the outcome attribute ($Y_D(x)$). Intuitively, the more informative an explanation table the more accurate it can be used to estimate y. To estimate y, Explanation Tables uses the maximum entropy principle [10] which states that the probability distribution which best represents the current state of knowledge, subject to known constraints, is the one with the highest entropy. A distribution with lower entropy assumes information not provided by the explanation table. Given an explanation table T, the maximum entropy estimation ε^* of y, over all $t \in D$, is selected from the ε which maximizes:

$$\sum_{t \in D} -\varepsilon(t) \cdot \log(\varepsilon(t)) \tag{4.1}$$

subject to

$$\begin{array}{ll} \forall_{t \in D} & 0 \leq \varepsilon(t) \leq 1 \\ \forall_{x \in T} & \sum_{t \in S_D(x)} \varepsilon(t) = |S_D(x)| \cdot Y_D(x) \end{array}$$

which constrains the $\varepsilon(t)$'s covered by each pattern x to be consistent with the fraction of records having label 1 for that pattern. Intuitively, the ε^* contains estimates for the unknown probabilities that generated y.

Example 1. To illustrate the main idea behind the maximum entropy applied to the data cube patterns consider the pattern (*, *, Salad) from `Workout` database. The pattern matches two records in Table 4.2 which are t_2 and t_7, i.e., its $|S_D(x)| = 2$. One of the two records has `goal met?` $= 1$ and the other record has `goal met?` $= 0$, and thus its $Y_D(x) = \frac{1}{2}$. There are many possible ways to estimate the values of `goal met?` using this pattern without violating the constraints in equation 4.1. Let us consider three possible estimations (ε's): $\varepsilon_1(t_2) = 1$ and $\varepsilon_1(t_7) = 0$, $\varepsilon_2(t_2) = \frac{1}{3}$ and $\varepsilon_2(t_7) = \frac{2}{3}$, and $\varepsilon_3(t_2) = \frac{1}{2}$ and $\varepsilon_3(t_7) = \frac{1}{2}$. Notice that the three estimations do not violate the constraint imposed by the pattern, which states that the sum of estimates of the matching records should be equal to 1. However, only one ε maximizes equation 4.1, i.e., $\varepsilon^* = \varepsilon_3$. It turns out that the maximum entropy estimate, in this case, is the most uniform. This is intuitive to our application because there is no reason to assign a higher value to the estimates of t_2 over t_7 —given that the true value of the outcome attributes ($y(t)$) is unknown and no patterns are considered.

Example 2. In the previous example, we illustrated the maximum entropy principle for one pattern. In this example, we illustrate with an explanation table. Recall the `Workout` database from Table 4.2 and suppose we are given an explanation table T with only the first two patterns from Table 4.3, (*, *, *) and (Sat, *, *), call them x_1 and x_2, respectively. Using T to estimate y we require that $\sum_{t \in S_D(x_1)} \varepsilon(t) = 7$ and require that $\sum_{t \in S_D(x_2)} \varepsilon(t) = 0$. Given the constraints imposed by x_2, we know $\varepsilon^*(t \in S_D(x_2)) = 0$ for all the Saturday records, which means that the sum of the $\varepsilon(t)$ values of all remaining (non-Saturday) records must be 7. Thus, the solution is to divide the value evenly by the remaining records, i.e., for all $t \in S_D(x_1) \setminus S_D(x_2)$ we set $\varepsilon^*(t \in S_D(x_2)) = \frac{7}{9}$.

In this example, maximizing Equation 4.1 is simple, but it may require complex optimization for large Explanation Tables. Thus, Explanation Tables apply the classical iterative scaling technique. Extended to our problem, a key result from [10] states that the distribution of ε^* has the following form:

$$\varepsilon^*(t) = \frac{e^{\sum_{x \in T : x \asymp t} \lambda(x)}}{1 + e^{\sum_{x \in T : x \asymp t} \lambda(x)}} \tag{4.2}$$

where $\lambda(x)$ is a multiplier value associated with each pattern. The idea behind iterative scaling is to refine the values until Equation 4.2 gives $\varepsilon^*(t)$ values that match (or converge to within a small fraction, say 0.01) the constraints implied by the explanation table, i.e., for all $x \in T$ $\sum_{t \in S_D(x)} \varepsilon(t) = |S_D(x)| \cdot Y_D(x)$ holds. Applied to our example, for x_1 = (*, *, *) iterative scaling gives $\lambda(x_1) = 1.122$ and for $x_2 =$ (Sat, *, *) we get $\lambda(x_2) = 9$. Thus, for non-Saturday records, which only match x_1, we get $\varepsilon^*(t) = \frac{e^{1.1822}}{1+e^{1.1822}} = 0.77$ and for Saturday records, which match both patterns, we get $\varepsilon^*(t) = \frac{e^{1.1822-9}}{1+e^{1.1822-9}} = 0.00004$.

4.2.4 Quantifying Information

This section discusses how to quantify the amount of information an explanation table provides regarding an outcome attribute y. In our evaluations, we compute the Kullback-Leibler (KL) divergence between the values of the outcome attribute y and the maximum entropy estimates ε^*, denoted as $D_{KL}(y \| \varepsilon^*)$. The intuition is that we desire the estimates to be as close as possible to the actual labels in the data. For a given record t, the KL-divergence is defined as:

$$D_{KL}(y(t)\|\varepsilon^*(t)) = y(t)\log\left(\frac{y(t)}{\varepsilon^*(t)}\right) + (1-y(t))\log\left(\frac{1-y(t)}{1-\varepsilon^*(t)}\right)$$

which measures the relative entropy or the expected number of bits required to encode $y(t)$ given $\varepsilon^*(t)$. For a given table D, the KL-divergence is defined as the summation of KL-divergence between its records and their estimates, which is:

$$D_{KL}(y\|\varepsilon^*) = \sum_{t\in D} y(t)\log\left(\frac{y(t)}{\varepsilon^*(t)}\right) + \sum_{t\in D}(1-y(t))\log\left(\frac{1-y(t)}{1-\varepsilon^*(t)}\right)$$

By fixing the value of $0\log(0)$ to 0, since y is binary, $D_{KL}(y\|\varepsilon^*)$ can be simplified to:

$$\sum_{t\in D|y(t)=1} \log\left(\frac{1}{\varepsilon^*(t)}\right) + \sum_{t\in D|y(t)=0} \log\left(\frac{1}{1-\varepsilon^*(t)}\right)$$

4.2.5 Estimating Information Gain

Unfortunately, generating optimal Explanation Tables is known to be NP-hard, since there exists a polynomial-time reduction from the Vertex Cover problem to the Explanation Tables Generation problem [16]. Thus, in this chapter, we evaluate greedy heuristics for generating Explanation Tables. A straightforward greedy approach would examine the space of all possible patterns occurring in D (i.e., the data cube $\mathcal{X}(D)$) and select the one with the highest information gain at each step. This requires computing ε^* via iterative scaling for each candidate pattern assuming that it is added to T then computing $D_{KL}(y\|\varepsilon^*)$.

Rather than running iterative scaling for each candidate pattern to compute a new $D_{KL}(y\|\varepsilon^*)$, the evaluated greedy heuristics compute an estimated information gain as follows [16]. Consider $E_D(x)$ that is a maximum entropy estimate (given T) of the fraction of records in the support set of x with y equals to 1, i.e., $E_D(x) = \frac{1}{|S_D(x)|}\sum_{t\asymp x}\varepsilon^*(t)$. We compare this with the fraction of records with y equals 1 in the support set of x that is $Y_D(x)$. The larger the difference, weighted by the size of the support set of x, the more beneficial it is to add x to the explanation table so that the outcomes of records in the support set of x can be estimated more precisely. Formally, $gain(x)$ is defined as follows:

$$|S_D(x)|\cdot\left(Y_D(x)\log\left(\frac{Y_D(x)}{E_D(x)}\right) + (1-Y_D(x))\log\left(\frac{1-Y_D(x)}{1-E_D(x)}\right)\right) \tag{4.3}$$

4.3 Algorithms

So far, we have presented the necessary background required to define Explanation Tables and discussed how to quantify their information content. In this section, we outline the algorithms for generating Explanation Tables. In Section 4.3.1, we outline an overall greedy framework that we call *Baseline*. We describe a straightforward sampling approach in Section 4.3.2 that we call SampleCube. We present two existing improvements over SampleCube called *Flashlight* and *Laserlight* in Sections 4.3.3 and 4.3.4, respectively. We summarize the differences among the evaluated algorithms in Section 4.3.5.

Algorithm 1 Baseline algorithm for generating Explanation Tables.

```
Require: database $D$, explanation table size $\mathcal{N}$
1: $T \leftarrow \{(*, \ldots, *)\}$
2: $\varepsilon^* \leftarrow Y_D(*, \ldots, *)$
3: For $i$ in 1 to $\mathcal{N}$ do
4:     $x_{max} \leftarrow \arg\max_{x \in \mathcal{X}(D)} gain(x)$
5:     $T \leftarrow T \cup \{x_{max}\}$
6:     update $\varepsilon^*$ based on $T$ via iterative scaling
7: End For
```

4.3.1 Baseline: A Basic Greedy Framework

Algorithm 1 (Baseline) shows a greedy framework to generate Explanation Tables, which will serve as a template for all the algorithms presented in this section. Line 1 initializes the explanation table T by adding the all-wildcard pattern, i.e., the pattern that matches all the records in database D. Line 2 assigns the ε^* estimates to all the records in D, i.e., the fraction of records with $y = 1$ (i.e., $Y_D(x_1)$) where $x_1 = (*, \ldots, *)$. The loop in lines 3–7 adds one pattern at a time into T up to $\mathcal{N} - 1$ patterns. Line 4 uses Equation 4.3 over the space of all possible patterns occurring in D and selects the one with the highest estimated information gain. Line 5 then adds to T the pattern with the highest information gain, which is the pattern that gives the lowest $D_{KL}(y \| \varepsilon^*)$ value. Line 6 updates ε^* based on the new pattern added to T in this iteration of the loop, i.e., it runs iterative scaling to compute new $\lambda(t)$ values for the existing patterns and the new pattern, as described in Section 4.2, and computes a new $\varepsilon^*(t)$ for each record in D using Equation 4.2.

Query 1 Generating all candidate patterns and their gain estimates.

```
SELECT (D.$A_1$, ..., D.$A_d$) AS $x$,
       gain(COUNT(*), AVG($y$), AVG($\varepsilon^*$))
FROM D
CUBE BY D.$A_1$, ..., D.$A_d$
ORDER BY gain DESC
LIMIT 1
```

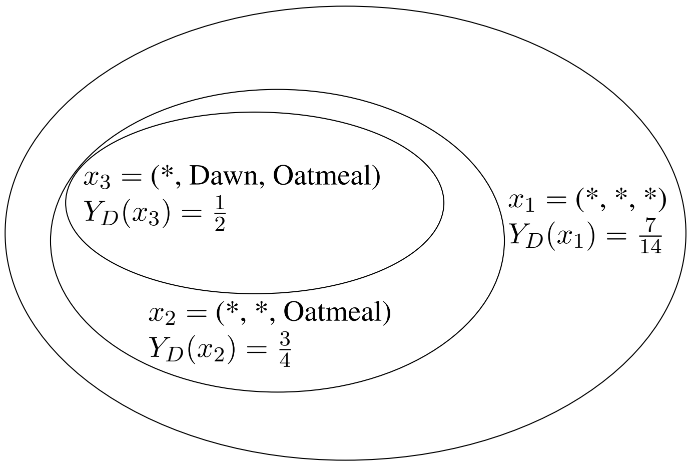


Figure 4.2: Relationship among patterns x_1, x_2, and x_3.

Query 1 shows an SQL implementation of line 4. The user-defined function `gain` computes the information gain estimation as shown in Equation 4.3. The values of ε^* are maintained as an additional attribute of D. For each pattern x, the `gain` function takes as input COUNT(*), which is $|S_D(x)|$, the average value of y, which is $Y_D(x)$, and the average value of ε^*, which is $E_D(x)$. Note the use of the `CUBE BY` operator to compute the gain of every pattern in $\mathcal{X}(D)$, and the ORDER-BY and LIMIT clauses to select the pattern with the highest `gain`.

The number of possible patterns can be very large ($2^d \times |D|$ in the worst case). Thus, Query 1 can be prohibitively expensive, for instance, it ran for over three hours on one of the data sets that we use in Section 4.4, which has 1.5 million records, 9 columns and 78 million patterns. Furthermore, gain from Equation 4.3 is not amenable to existing optimizations for computing aggregates over data cubes, such as [11, 55]. For instance, consider the three patterns shown in Figure 4.2 from the `Workout` database in Table 4.2 x_1 = (*, *, *), x_2 = (*, *, Oatmeal), and x_3 = (*, Dawn, Oatmeal). Notice that x_1 contains x_2 which contains x_3. The pattern x_1 and x_3 contain exactly the same fraction of records with y equals 1 which is $\frac{1}{2}$, but the fraction of records with y equals 1 covered by x_2 is $\frac{3}{4}$. Suppose x_1 already exists in the explanation table; then the information gain from x_3 is zero. However, if x_2 is added next, then x_3 will have a non-zero gain because the maximum entropy estimates ε^* will be based on the assumption that all Oatmeal records, including the Oatmeal Dawn ones covered by x_3, have a similar distribution of the outcome attribute y.

Notice that line 4 is the bottleneck as it needs to compute $gain$ for a very large number of patterns each iteration to compute the new $gain$ estimates based on the newly added pattern to T; line 6 is relatively inexpensive as it only runs iterative scaling once, after selecting the new pattern. Since we cannot prune patterns from consideration throughout the execution of Algorithm 1, we evaluate sample-based techniques that improve performance at the expense of accuracy.

4.3.2 A Straightforward Sampling Approach

As discussed in the previous section, line 4 is the main performance bottleneck for Algorithm 1 because the number of candidate patterns in the data cube of the table ($\mathcal{X}(D)$) is large. Thus, in this section, we consider a straightforward sampling approach that generates Explanation Tables. The straightforward sampling is similar to Baseline but considers only the patterns in the data cube of a sample, i.e., $\mathcal{X}(s)$, where s is a uniform random sample drawn from table D. The sample follows the same schema of table D thus it preserves all the values of the attributes (A_i) and the outcome attribute (y). Query 2 shows an SQL implementation of line 4 for the straightforward sampling approach. We call this straightforward sampling approach *SampleCube*.

Query 2 SampleCube pattern generation with $gain_s$ estimates.

```
SELECT    (t.$A_1$, ..., t.$A_d$) AS $x$,
          gain(COUNT(*), AVG($y$),AVG($\varepsilon^*$))
FROM      s
CUBE BY   s.$A_1$,..., s.$A_d$
ORDER BY gain DESC
LIMIT     1
```

It is plausible to use the data cube of the sample at a candidate space for two main reasons. First, at the cost of accuracy, Query 2 is expected to run faster than Query 1 (Baseline) since the size of $\mathcal{X}(s)$ is expected to be smaller than $\mathcal{X}(D)$. Second, the more informative patterns tend to have a higher support and thus are more likely to appear in a uniform random sample. For instance, recall the `Workout` database, there are four Oatmeal records and only one Rice record in Table 4.2, which means that an Oatmeal record is more likely to be included in the random sample than a Rice record. Hence, the pattern (*, *, Oatmeal) which can improve the ε^* of more records than (*, *, Rice) is more likely to be considered by SampleCube. While informative patterns are more likely to appear in a sample than less informative patterns, bad samples that miss the most informative patterns can occur. In order to alleviate the effects of bad samples, at every iteration, a new sample is drawn. To illustrate this idea, consider the Explanation Table shown in Table 4.3, which required three iterations to generate. The most informative pattern, which is (Sat, *, *), is more likely to appear in one of the three samples than in one. Furthermore, in our evaluations, we have found that the running time overhead of drawing a new sample at every iteration is minimal.

Effectiveness of SampleCube

To assess the effectiveness of SampleCube this section quantifies the trade-off between running time and quality. We compare the information gain from Explanation Tables using SampleCube with different sample sizes to the information gain of Baseline. Figure 4.3 shows the evaluation of both techniques using a data set called `Upgrade` which contains 5795 airline ticket records from the United Airline. Each record contains details such as origin airport, destination airport, flight date and time, etc., and the outcome attribute indicates whether the passenger was upgraded to business class or not. In Section 4.4, we discuss the `Upgrade` data set and our experimental setup in more details and compare the performance of SampleCube to other techniques using different data sets as well. We ran SampleCube five times, each time choosing a different random sample of a given size, and report the average numbers across the five runs. The sample size is shown on the lower x-axis and it varies from 128 to 5795. The upper x-axis shows the running time to generate an explanation table with 50 patterns: Baseline took just over 2000 seconds whereas the running time of SampleCube increases as the sample size increases. Notice that the upper x-axis is linear and was used to decide the positions of the numbers on the lower x-axis. The y-axis shows the average information gain of the generated explanation tables. We define this as the improvement in D_{KL} compared to an explanation table with only the all-wildcards pattern. The horizontal line at an information gain of just over 1000 corresponds to Baseline.

Evaluating SampleCube Candidate patterns

As shown in Figure 4.3, SampleCube requires a relatively large sample size to approach the information gain of Baseline. This is not effective because it takes a sample size of 4096 (roughly 70% of table D) to achieve an average information gain of 900 (90% of the Baseline information gain). At this sample size, SampleCube takes 75% of the time taken by Baseline to achieve 90%

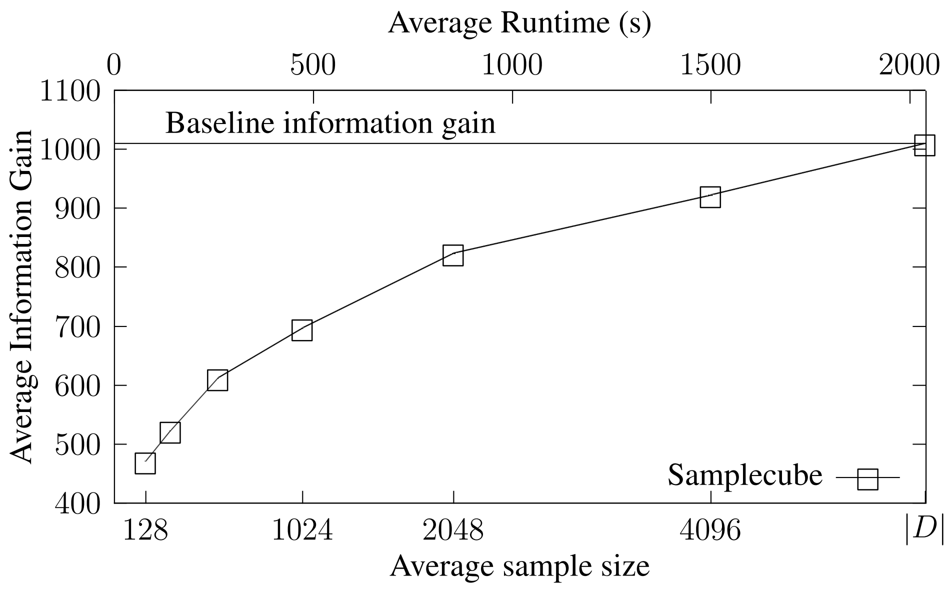


Figure 4.3: Information gain of SampleCube using `Upgrade`.

of Baseline's information gain. This is because the number of patterns considered by SampleCube is close to the number of patterns considered by Baseline. We observed that SampleCube considers the patterns selected by Baseline but does not add them to the explanation table. The reason behind this is that SampleCube picks the patterns that improve the accuracy of ε^* estimates over the sample and not over table D. Finally, at a sample size equal to the entire database, i.e., $|s| = |D|$, the running time and quality of SampleCube are equal to Baseline which is expected since Query 2 becomes equivalent to Query 1 when $s = D$. Thus, SampleCube is not an effective optimization over the baseline; we discuss this problem further in Section 4.3.4.

Recall Equation 4.3, which estimates the improvement in $\varepsilon^*(x)$, i.e., information gain over D. The Equation requires the values $S_D(x)$, $Y_D(x)$, and $E_D(x)$ but Query 2 uses $S_s(x)$, $Y_s(x)$, and $E_s(x)$. Thus, we define $gain_D(x)$ to be the result of Equation 4.3 using the fraction of records that have $y(t)$ equals to 1 in the support set of x, i.e., $Y_D(x)$, and the estimates $\varepsilon^*(x)$. We also define $gain_s(x)$ to be the result of Equation 4.3 using $S_s(x)$, $Y_s(x)$, and $E_s(x)$ computed from the sample s, which is an estimate of the improvement in ε^* over s. Thus, the reason behind the drop in the information gain of the patterns chosen by SampleCube, which uses $gain_s$ compared to Baseline which uses $gain_D$ lies in how they both estimate information gain.

To verify this claim, let us consider another implementation for line 4, that is effectively a hybrid of Baseline and SampleCube. The new algorithm considers the same candidate pattern space as SampleCube, which is $\mathcal{X}(s)$, but uses $gain_D$. We refer to the new algorithm as Sample-

Query 3 Computing $gain_D$ for sample patterns.

```
SELECT (X.$A_1$,..., X.$A_d$) AS $x$,
       gain(COUNT(*), AVG($y$), AVG($\varepsilon^*$))
FROM D, $\mathcal{X}(s)$ AS X
WHERE  ((X.$A_1$ ISNULL) OR (X.$A_1$ = D.$A_1$)) AND ...
       AND ((X.$A_d$ ISNULL) OR (X.$A_d$ = D.$A_d$))
GROUP BY X.$A_1$,..., X.$A_d$
ORDER BY gain DESC
LIMIT 1
```

Cube $gain_D$ and the original as SampleCube $gain_s$. Query 3 shows the SQL implementation of line 4 of SampleCube $gain_D$. To compute $gain_D$, we join $\mathcal{X}(s)$ with D to obtain $S_D(x)$, $Y_D(x)$, and $E_D(x)$ needed to compute $gain_D$. In this query, we use the values NULL to represent the wildcard symbol used in the patterns.

Figure 4.4 shows the results using the `Upgrade` data set, with the sample size on the x-axis and the average information gain over five runs with different random samples on the y-axis. In our evaluations, we ensured that both algorithms considered the same patterns by ensuring they have the same samples, i.e., a larger sample size includes all the records in the smaller sample size. Since both algorithms consider the same candidate pattern space ($\mathcal{X}(s)$), the only difference is how both algorithms compute $gain$. The difference in the information gain is significant on this data set since SampleCube $gain_D$ approaches the information gain of Baseline with a sample size as small as 16 records. This confirms that restricting the candidate pattern space to $\mathcal{X}(s)$ is an effective optimization. However, using $gain_s$ leads to bad estimations of information gain over D. Computing $gain_D$ for all the patterns in $\mathcal{X}(s)$ (Query 3) is expensive because it requires joining the cube of the sample ($\mathcal{X}$) with the D. We discuss this in more details in Section 4.4.3. The next section present Flashlight, which is a more efficient algorithm for computing $gain_D$ for patterns in $\mathcal{X}(s)$.

4.3.3 A Sample-Based Pruning Approach

As explained in the previous section, sampling can be used to generate good Explanation Tables, but only when combined with accurate gain estimates obtained by taking the entire database into account. This section describes the Flashlight algorithm that produces the same output as Query 3, but instead of joining inputs of sizes $|\mathcal{X}(s)|$ and $|D|$, it only needs to join inputs of sizes $|s|$ and $|D|$. Since $\mathcal{X}(s)$ can be much larger than s, Flashlight significantly improves the performance of generating Explanation Tables, as we will show using our evaluations in

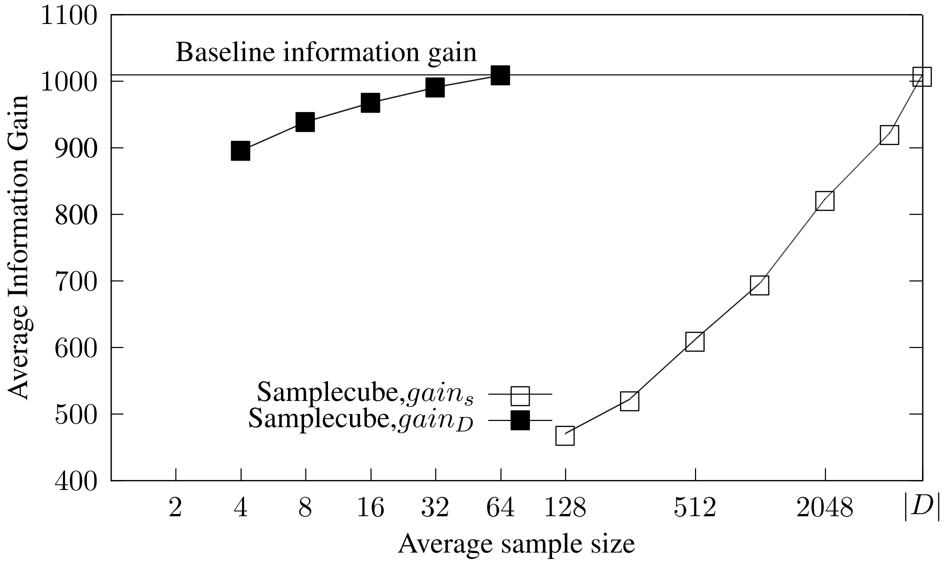


Figure 4.4: SampleCube information gain using $gain_s$ and $gain_D$ for `Upgrade`.

Section 4.4. First, we need the following definitions. Let $x = lcap(t_1, t_2)$ be the lowest common ancestor pattern of records t_1 and t_2, then $x[A_i] = t_1[A_i]$ when $t_1[A_i] = t_2[A_i]$ and $x[A_i] = *$ otherwise. The $lcap(t_1, t_2)$ gives the "tightest" or most specific pattern that matches both records. For instance, $lcap$((Fri, Dawn, Banana),(Fri, Night, Salad)) is (Fri, *, *) and $lcap$((Fri, Dawn, Banana),(Sat, Midday, Toast)) is (*, *, *).

We also define the lowest common ancestor cross product, $LCA(D_1, D_2)$, of two relational tables, D_1 and D_2, as $\bigcup_{t_i \in D_1} \bigcup_{t_j \in D_2} lcap(t_i, t_j)$. We call this operator the Lowest Common Ancestor cross product or $LCA(D_1, D_2)$ because for every two corresponding records in $D_1 \times D_2$ we generate $lcap$ of both records.

Table 4.4 shows a sample s of the `Workout` database, and Table 4.5 shows the $LCA(s, D)$ of the database using this sample. Each row of Table 4.5 corresponds to one of the 14 records from the `Workout` table and each column to one of the three records from the sample. Notice that Table 4.5 assumes a duplicate-preserving version of LCA, meaning that the same lowest common ancestor pattern can be derived from more than one pair of records from `Workout` and a record from the sample. For instance, the underlined pattern (*, *, Banana) appears in the $LCA(s, D)$ five times.

Algorithm Description

The intuition behind Flashlight is that $LCA(s, D)$ can be used to compute $gain_D$ without computing the costly cross product of $\mathcal{X}(s)$ with D that is done in Query 3. Flashlight uses Query 4 to implement line 4 of the greedy algorithm for selecting the next pattern for T, i.e., computing $gain_D$ of all the records in $\mathcal{X}(s)$ and choosing the one with the highest gain. Query 4 uses a sequence of view calls to generate $\mathcal{X}(s)$ with $gain_D$ estimates. The following worked example explains the main idea behind Flashlight. Consider the `Workout` database from Table 4.2 and the sample from Table 4.4. Assume the (*, *, *) pattern has just been added to T, meaning that every record in D now has a ε^* of 0.5; recall that we assumed an additional column of D to store ε^*.

The first view in Query 4, called `LCAG`, computes the cross product $LCA(s, D)$, followed by a GROUP-BY aggregation on each unique pattern that computes the following: COUNT(*), SUM($D.y$), and SUM($D.\varepsilon^*$). Table 4.6 shows these aggregates for the 13 distinct patterns. The pattern (*, *, Banana) appears five times in the $LCA(s, D)$, and its records in D are:

(Fri, Dawn, Banana) which has $y = 1$

(Sun, Morning, Banana) which has $y = 1$

(Mon, Midday, Banana) twice which has $y = 1$

(Sat, Dawn, Banana) which has $y = 0$

Thus, the COUNT of the pattern is five and the SUM(y) is four since the pattern (Mon, Midday, Banana) is counted twice. Similarly, since at this point every record in D has $\varepsilon^* = 0.5$, the sum of ε^* is 2.5, which includes counting (Mon, Midday, Banana) twice.

An important observation is that the view `LCAG` aggregates do not correspond to the support or the $Y_D(x)$ and $E_D(x)$ values of the patterns. For instance, the support of (*, *, Banana) in D is four records, but its count in the LCA is five.

day	time	food	goal met?
Sun	Morning	Banana	Yes
Thu	Dawn	Oatmeal	Yes
Sat	Dawn	Banana	No

Table 4.4: A three records sample from the `Workout` database.

Query 4 Optimized query for generating $\mathcal{X}(s)$ (Flashlight).

```
CREATE VIEW LCAG AS (
SELECT CASE D.$A_1$ = s.$A_1$ THEN s.$A_1$ ELSE NULL, ...
       CASE D.$A_d$ = s.$A_d$ THEN s.$A_d$ ELSE NULL,
       COUNT(*) as ct, SUM(D.$y$) AS s$y$, SUM(D.$\varepsilon^*$) AS s$\varepsilon$
FROM s, D
GROUP BY s.$A_1$, ..., s.$A_d$ )

CREATE VIEW XLCA AS (
SELECT (LCAG.$A_1$, ...,  LCAG.$A_d$) AS $x$,
       SUM(LCAG.ct) AS sct,
       SUM(LCAG.s$y$) AS s$y$,
       SUM(LCAG.s$\varepsilon$) AS s$\varepsilon$
FROM LCAG
CUBE BY LCAG.$A_1$, ...,  LCAG.$A_d$)

CREATE VIEW  FL AS (
SELECT (PL.$A_1$, ..., PL.$A_d$) AS $x$,
       COUNT(*) AS ct,
       PL.sct/ct AS sct,
       PL.s$y$/ct AS s$y$,
       PL.s$\varepsilon$/ct AS s$\varepsilon$
FROM XLCA AS PL, s
WHERE ((PL.$A_1$ ISNULL) OR (PL.$A_1$ = s.$A_1$)) AND ...
        AND ((PL.$A_d$ ISNULL) OR (PL.$A_d$ = s.$A_d$))
GROUP BY PL.$A_1$, ..., PL.$A_d$, sct, s$y$, s$\varepsilon$)

SELECT (FL.$A_1$, ...,  FL.$A_d$) AS $x$,
        gain(sct, s$y$/sct, s$\varepsilon$/sct)
FROM FL
ORDER BY gain DESC
LIMIT 1
```

id	$y(t)$	**database \ sample**	(Sun, Morning, Banana)	(Thu, Dawn, Oatmeal)	(Sat, Dawn, Banana)
1	1	(Fri, Dawn, Banana)	$\underline{\text{(*, *, Banana)}}$	(*, Dawn, *)	(*, Dawn, Banana)
2	1	(Fri, Night, Salad)	(*, *, *)	(*, *, *)	(*, *, *)
3	1	(Sun, Dusk, Oatmeal)	(Sun, *, *)	(*, *, Oatmeal)	(*, *, *)
4	1	(Sun, Morning, Banana)	(Sun, Morning, Banana)	(*, *, *)	$\underline{\text{(*, *, Banana)}}$
5	1	(Mon, Afternoon, Oatmeal)	(*, *, *)	(*, *, Oatmeal)	(*, *, *)
6	1	(Mon, Midday, Banana)	$\underline{\text{(*, *, Banana)}}$	(*, *, *)	$\underline{\text{(*, *, Banana)}}$
7	0	(Tue, Morning, Salad)	(*, Morning, *)	(*, *, *)	(*, *, *)
8	0	(Wed, Night, Burgers)	(*, *, *)	(*, *, *)	(*, *, *)
9	1	(Thu, Dawn, Oatmeal)	(*, *, *)	(Thu, Dawn, Oatmeal)	(*, Dawn, *)
10	0	(Sat, Afternoon, Nuts)	(*, *, *)	(*, *, *)	(Sat, *, *)
11	0	(Sat, Dawn, Banana)	$\underline{\text{(*, *, Banana)}}$	(*, Dawn, *)	(Sat, Dawn, Banana)
12	0	(Sat, Dawn, Oatmeal)	(*, *, *)	(*, Dawn, Oatmeal)	(Sat, Dawn, *)
13	0	(Sat, Dusk, Rice)	(*, *, *)	(*, *, *)	(Sat, *, *)
14	0	(Sat, Midday, Toast)	(*, *, *)	(*, *, *)	(Sat, *, *)

Table 4.5: Computing the $LCA(s, D)$ of the `Workout` database with the sample from Table 4.4.

The second view, which is called `XLCA`, applies the CUBE BY operator to `LCAG` which generates all the patterns in $\mathcal{X}(s)$, and computes the summation of the aggregates computed by `LCAG`. The patterns and aggregates are shown in the four leftmost columns of Table 4.7. For instance, to compute the SUM(count) value for pattern (*, *, Banana) in Table 4.7, the values of the count column of four patterns from Table 4.6, that are (*, *, Banana) (with count 5), (*, Dawn, Banana) (with count 1), and (Sat, Dawn, Banana) (with count 1) and of (Sun, Morning, Banana) (with count 1) are summed. Thus, its count in Table 4.7 is eight, and SUM(y) and SUM(ε^*) are computed similarly.

The view `XLCA` has 20 patterns which are the set of patterns $\mathcal{X}(s)$ or the data cube of s. `XLCA` corrects the count and sum aggregates so they correspond to $|S_D(x)|$, $Y_D(x)$ and $E_D(x)$. The main intuition behind the Flashlight savings is noticing that we do not need to check the database in order to compute the aggregates ($|S_D(x)|$, $Y_D(x)$ and $E_D(x)$), alternatively, we can check the sample. For instance (*, *, Banana) matches two records in s (shown in Table 4.4), so we only divide the aggregates we have computed by two. This gives the corrected count $=$ 4, SUM(y) $=$ 3 and SUM(ε^*) $=$ 2. From these sums we derive the averages as follows: Y_D(*, *, Banana) $= \frac{\text{SUM}(y)}{\text{count}} = \frac{3}{4}$ and E_D(*, *, Banana) $= \frac{\text{SUM}(\varepsilon^*)}{\text{count}} = \frac{2}{4}$. The rightmost four columns of Table 4.7 show the "frequency in sample" of each candidate pattern x (i.e., how many records in s it matches), the corrected count (i.e., $|S_D(D)|$, $Y_D(x)$ and $E_D(x)$). This step is realized by the view called `FL`.

The final step is to select the pattern from $\mathcal{X}(s)$ with the highest $gain_D$. We have all the aggregates needed for this, and we evaluate Equation 4.2 for each candidate pattern. In this

Pattern	COUNT(*)	SUM(y)	SUM(ε^*)		**Data cube**
(*, *, *)	21	9	10.5	$\rightarrow$	(*, *, *)
(*, *, Banana)	5	4	2.5	$\rightarrow$	(*, *, Banana) ,(*, *, *)
(*, Dawn, *)	3	2	1.5	$\rightarrow$	(*, Dawn, *) ,(*, *, *)
(Sat, *, *)	3	0	1.5	$\rightarrow$	(Sat, *, *) ,(*, *, *)
(*, *, Oatmeal)	2	2	1	$\rightarrow$	(*, *, Oatmeal), (*, *, *)
(*, Dawn, Banana)	1	0	.5	$\rightarrow$	(*, Dawn, Banana),(*, Dawn, *),(*, *, Banana) ,(*, *, *)
(*, Dawn, Oatmeal)	1	0	.5	$\rightarrow$	(*, Dawn, Oatmeal),(*, Dawn, *),(*, *, Oatmeal),(*, *, *)
(*, Morning, *)	1	0	.5	$\rightarrow$	(*, Morning, *),(*, *, *)
(Sat, Dawn, *)	1	0	.5	$\rightarrow$	(Sat, Dawn, *),(Sat, *, *),(*, Dawn, *),(*, *, *)
(Sat, Dawn, Banana)	1	0	.5	$\rightarrow$	(Sat, Dawn, Banana),. . . , (Sat, *, *),(*, Dawn, *),(*, *, *)
(Sun, *, *)	1	1	.5	$\rightarrow$	(Sun, *, *),(*, *, *)
(Sun, Morning, Banana)	1	1	.5	$\rightarrow$	(Sun, Morning, Banana),. . . , (*, Morning, *),(*, *, Banana),(*, *, *)
(Thu, Dawn, Oatmeal)	1	1	.5	$\rightarrow$	(Thu, Dawn, Oatmeal),. . . ,(*, Dawn, *),(*, *, Oatmeal),(*, *, *)

Table 4.6: Computing aggregates over the $LCA(s, D)$ from Table 4.4.

round, the pattern (Sat, *, *) has the highest gain of $5(0 + \log(\frac{1}{0.5})) = 5\log(2)$, so it is added to the explanation table.

Complexity Analysis

We now discuss the computational complexity of Flashlight, which is $O(|s|\,|D| + |\mathcal{X}(s)|\,|s|)$. Recall that Query 3 requires $|\mathcal{X}(s)|\,|D|$ operations to compute $gain_D$ for patterns in $\mathcal{X}(s)$. On the other hand, the view LCAG requires $|s|\,|D|$ operations to compute $LCA(s, D)$, which is a cross product followed by a GROUP-BY. The XLCA step needs $|\mathcal{X}(s)|$ operations to generate ancestor patterns. The view FL needs $|\mathcal{X}(s)|\,|s|$ operations to join ancestor patterns with the sample to compute the corrected aggregates. Our analysis is based on two assumptions: first, the RDBMS will always pick a nested loop operator to realize the joins in the Flashlight query; second, the RDBMS will always pick hash based operators to compute GROUP-BY. In our evaluations we have validated that this was the case with PostgreSQL.

4.3.4 Sampling with Pruning

In the previous sections, we have presented SampleCube which considers a pruned set of candidate patterns that are based on a sample's data cube ($\mathcal{X}(s)$). We also discussed Flashlight an improvement over SampleCube which shares the same candidate patterns but uses accurate estimates for ε^* improvement over D, i.e., $gain_D$. In this section, we discuss another improvement over SampleCube that optimizes its running time instead of its accuracy. The Laserlight algorithm works with a reduced set of patterns $LCA(s, s)$ and uses $gain_s$ to estimate $gain_D$.

Pattern	SUM(count)	SUM(y)	SUM(ε^*)	**Freq. in sample**	**Corrected** count	$Y_D(x)$	$E_D(x)$
(*, *, *)	42	21	21	3	14	.5	.5
(*, *, Banana)	8	6	4	2	4	.75	.5
(*, *, Oatmeal)	4	3	1.5	1	4	.75	.5
(*, Dawn, *)	8	4	4	2	4	.5	.5
(*, Morning, *)	2	1	1	1	2	.5	.5
(Sat, *, *)	5	0	2.5	1	5	0	.5
(Sun, *, *)	2	2	1	1	2	1	.5
(Thu, *, *)	1	1	.5	1	1	1	.5
(*, Dawn, Banana)	2	1	1	1	2	.5	.5
(*, Dawn, Oatmeal)	2	1	1	1	2	.5	.5
(*, Morning, Banana)	1	1	.5	1	1	1	.5
(Sat, *, Banana)	1	0	.5	1	1	0	.5
(Sun, *, Banana)	1	1	.5	1	1	1	.5
(Thu, *, Oatmeal)	1	1	.5	1	1	1	.5
(Sat, Dawn, *)	2	0	1	1	2	0	.5
(Sun, Morning, *)	1	1	.5	1	1	1	.5
(Thu, Dawn, *)	1	1	.5	1	1	1	.5
(Sun, Morning, Banana)	1	1	.5	1	1	1	.5
(Sat, Dawn, Banana)	1	0	.5	1	1	0	.5
(Thu, Dawn, Oatmeal)	1	1	.5	1	1	1	.5

Table 4.7: Computing aggregates over the data cube of the sample from Table 4.4.

The main idea behind the Laserlight algorithm is that $\mathcal{X}(s)$ contains many redundant patterns on s, i.e., many patterns match the same set of records in s. Recall Table 4.6, which lists all 20 patterns in $\mathcal{X}(s)$ using s from Table 4.4, with $gain$ computed based on D from Table 4.2. Consider the following two patterns from $\mathcal{X}(s)$: $x_1 =$ (Sat, *, *) and $x_2 =$ (Sat, Dawn, *). They have the same support set over s as they both match the same single record with id = 11. As a result, they will have the same information gain estimation. Similarly, two patterns that overlap significantly will have a very similar gain. The idea behind Laserlight is to avoid considering redundant patterns, which are common in real data sets due to correlations and therefore speeds up the process of choosing the pattern with the highest gain.

To accomplish this, we revisit the idea of lowest common ancestor patterns, which we used in Flashlight to efficiently compute $gain_D$ for patterns in $\mathcal{X}(s)$. The insight is that the pattern set $LCA(s, s)$ does not contain any redundant patterns, as each such pattern is the lowest descendant for which a unique pair of records co-occurs in the same support set and, hence, mutually nonredundant [16]. Laserlight draws a random sample s and runs the Baseline algorithm on the

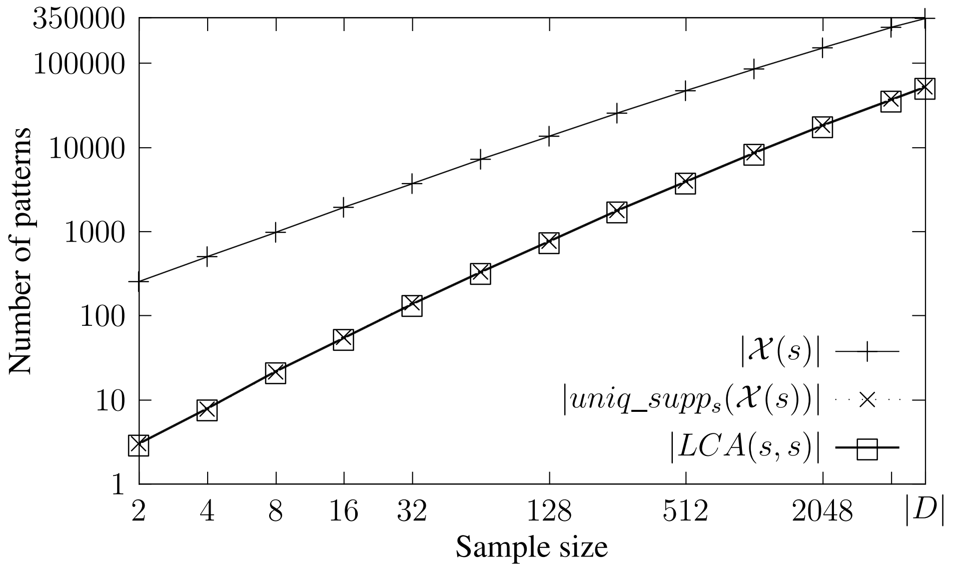


Figure 4.5: Number of patterns as a function of sample size using `Upgrade`.

sample, similar to SampleCube $gain_s$, but with line 4 choosing the best pattern only from those in $LCA(s, s)$.

To quantify how many patterns in $\mathcal{X}(s)$ are redundant, Figure 4.5 plots three quantities as functions of the sample size over the `Upgrade` data set, calculated as averages over five runs with different uniform samples; note that both axes are logarithmic. The first is $|\mathcal{X}(s)|$, which is 7229 for $|s| = 64$ and grows to approximately 350,000, which is $|\mathcal{X}(D)|$. The other two lines are very similar: the number of patterns with unique support sets in $\mathcal{X}(s)$, labeled "$|unique\_supp_s(\mathcal{X}(s))|$" and the size of $LCA(s, s)$. At $|s| = 64$, there are only roughly 330 of these, meaning that most patterns in $\mathcal{X}(s)$ are in fact redundant in this example. This means that Laserlight can be significantly more efficient in practice than SampleCube $gain_s$.

Laserlight uses Query 5 to implement line 4. The query consists of two views `LCAG` and `LL`. The first view, `LCAG`, computes the unique patterns in $LCA(s, s)$, recall that a pattern may appear more than once in the LCA cross product. The second view, `LL` computes a cross product between the $LCA(s, s)$ and a record from s, to compute $gain_s$ for every pattern. To illustrate how Laserlight works, we outline the following worked example using the same `Workout` database from Table 4.2 and s consists of the six records from Table 4.8. In this example, we assume that T is in its initial state, i.e., $T = \{(*\ ,*\ ,*\ )\}$. Notice that at this point ε^* for all the records in s is $\frac{2}{3}$, which is the fraction of 1's in the sample, which is different from the 1's fraction in D, as would have been computed by Baseline and Flashlight. For the ease of exposition, we assume

Query 5 Query for generating $LCA(s, s)$ (Laserlight).

```
CREATE VIEW  LCAG AS (
SELECT CASE s1.$A_1$ = s2.$A_1$ THEN s1.$A_1$ ELSE NULL, ...
       CASE s1.$A_d$ = s2.$A_d$ THEN s1.$A_d$ ELSE NULL
FROM s AS s1, s AS s2
GROUP BY s1.$A_1$, ..., s1.$A_d$ )

CREATE VIEW  LL AS (
SELECT (X.$A_1$,..., X.$A_d$) AS $x$,
        gain(COUNT(*), AVG(s.$y$), AVG(s.$\varepsilon^*$))
FROM LCAG AS X, s
WHERE  ((X.$A_1$ ISNULL) OR (X.$A_1$ = s.$A_1$)) AND ...
        AND ((X.$A_d$ ISNULL) OR (s.$A_d$ = D.$A_d$))
GROUP BY X.$A_1$, ..., X.$A_d$
ORDER BY gain DESC
LIMIT 1 )
```

that the table storing the sample s includes an additional attribute with the values of $\varepsilon^*(t)$.

To select the next pattern into T, the first step is to compute $LCA(s, s)$, which is shown in the leftmost column of Table 4.9. There are only six patterns in this set, as compared to 20 in $\mathcal{X}(s)$. In the second step, we compute the cross product $LCA(s, s)$ (LCAG) with s to compute the following three aggregates for each pattern: the support $|S_s(x)|$ and the fractions $Y_s(x)$ and $E_s(x)$, which we show in Table 4.10. The final step is to select the pattern with the highest $gain_s$, which turns out to be (Sat, Dawn, Banana). Its gain is $0 \log 0 + 1 \log(1 \ / \ \frac{1}{3}) = \log 3$.

Complexity Analysis

Computing the pattern set $LCA(s, s)$ takes $|s|^2$ operations and joining this set with the sample to compute gains requires $|LCA(s, s)|\,|s|$ operations. Since $\mathcal{X}(s)$ can be much larger than $LCA(s, s)$, as we showed earlier, Laserlight can be much more efficient than SampleCube, whose complexity is $|\mathcal{X}(s)|\,|s|$. Our analysis is based on two assumptions: first, the RDBMS will always pick a nested loop operator to realize the joins in the Laserlight query; second, the RDBMS will always pick hash based operators to compute GROUP-BY. In our evaluations we have validated that this was the case with PostgreSQL.

day	time	food	goal met?
Fri	Night	Salad	Yes
Sun	Morning	Banana	Yes
Mon	Midday	Banana	Yes
Thu	Dawn	Oatmeal	Yes
Sat	Dawn	Banana	No
Sat	Dusk	Rice	No

Table 4.8: A Six records sample from `Workout` database.

id	$y(t)$	**Sample \ Sample**	(Fri, Night, Salad)	(Sun, Morning, Banana)	(Mon, Midday, Banana)	(Thu, Dawn, Oatmeal)	(Sat, Dawn, Banana)	(Sat, Dusk, Rice)
2	1	(Fri, Night, Salad)	(Fri, Night, Salad)	(*, *, *)	(*, *, *)	(*, *, *)	(*, *, *)	(*, *, *)
4	1	(Sun, Morning, Banana)	(*, *, *)	(Sun, Morning, Banana)	(*, *, Banana)	(*, *, *)	(*, *, Banana)	(*, *, *)
6	1	(Mon, Midday, Banana)	(*, *, *)	(*, *, Banana)	(Mon, Midday, Banana)	(*, *, *)	(*, *, Banana)	(*, *, *)
9	1	(Thu, Dawn, Oatmeal)	(*, *, *)	(*, *, *)	(*, *, *)	(Thu, Dawn, Oatmeal)	(*, Dawn, *)	(*, *, *)
11	0	(Sat, Dawn, Banana)	(*, *, *)	(*, *, Banana)	(*, *, Banana)	(*, Dawn, *)	(Sat, Dawn, Banana)	(Sat, *, *)
13	0	(Sat, Dusk, Rice)	(*, *, *)	(*, *, *)	(*, *, *)	(*, *, *)	(Sat, *, *)	(Sat, Dusk, Rice)

Table 4.9: Computing aggregates over $LCA(s, s)$.

4.3.5 Algorithms Summary

We conclude this section with a summary of the evaluated algorithms, which are shown in Table 4.11. The algorithms differ in the set of candidate patterns considered and the method for computing gain. Since $LCA(s, s) \subseteq \mathcal{X}(s) \subseteq \mathcal{X}(D)$, Baseline is the most accurate (i.e., it should have the highest information gain), but the slowest. Flashlight should be much faster but less accurate, followed by Laserlight, which should be the fastest but likely the least accurate. Flashlight generates the same Explanation Tables as SampleCube $gain_D$ but is faster.

4.4 Evaluation

This section presents our evaluation results. We compare the information content of Explanation Tables, i.e., improvement in D_{KL} , against existing method including Decision Trees and database summaries (Section 4.4.2). Furthermore, we compare the proposed algorithms for generating Explanation Tables in terms of running time, scalability, and information content (Section 4.4.3).

Pattern	SUM(count)	AVG(y) ($Y_s(x)$)	AVG(ε^*) ($E_s(x)$)
(*, *, *)	6	0.67	0.67
(*, *, Banana)	3	0.67	0.67
(*, Dawn, *)	2	0.5	0.67
(Sat, *, *)	2	0	0.67
(Fri, Night, Salad)	1	1	0.67
(Mon, Midday, Banana)	1	1	0.67
(Sat, Dawn, Banana)	1	0	0.67
(Sat, Dusk, Rice)	1	0	0.67
(Sun, Morning, Banana)	1	1	0.67
(Thu, Dawn, Oatmeal)	1	1	0.67

Table 4.10: Computing aggregates over $LCA(s, s)$ using the sample from Table 4.8.

Algorithm	**Candidate Pattern Space**	**Gain estimation**
Baseline	$\mathcal{X}(D)$	$gain_D$
SampleCube $gain_s$	$\mathcal{X}(s)$	$gain_s$
SampleCube $gain_D$	$\mathcal{X}(s)$	$gain_D$
Flashlight	$\mathcal{X}(s)$	$gain_D$
Laserlight	$LCA(s, s)$	$gain_s$

Table 4.11: A comparison of algorithms for generating Explanation Tables.

4.4.1 Experimental Setup

All evaluations were run on a 3.3 GHz AMD FX-6100 processor with 16 GB RAM and 6 MB of cache running Ubuntu 12. The evaluated algorithms are implemented inside PostgreSQL (v9.1). No personally identifiable information was gathered or used in conducting this study. Our analysis is based on the following anonymous data sets.

- `Upgrade`: Contains information about 5795 United Airline ticket records obtained from `https://www.diditclear.com/`. Each record contains seven attributes about the tickets (origin airport, destination airport, date, booking class, etc.) and a binary attribute indicating whether the passenger was upgraded to business class. This data set has $|\mathcal{X}(D)|$ =346,532.

	`Upgrade`	`Adult`	`Income`
Flashlight ($\|s\| = 16$)	47 s	252 s	0.6 h
DT:Multiway	1021 s	2599 s	56 h
DT:Binary	6120 s	8348 s	N/A
SURPRISE_cmp	61 s ($\epsilon = 0.2$)	243 s ($\epsilon = 0.1$)	N/A
SURPRISE_bst	88 s ($\epsilon = 0.08$)	875 s ($\epsilon = 0.02$)	N/A
Cover_bst	240 s ($\hat{c} = 0.99$)	2051 s ($\hat{c} = 0.6$)	N/A

Table 4.12: Runtime to generate summary tables.

- `Adult`: U.S. Census data containing demographic attributes such as occupation, education and gender, and a binary outcome attribute denoting whether the given person's income exceeded 50,000. This data set was downloaded from the UCI archive at `archive.ics.uci.edu/ml/datasets/Adult/` and contains 32561 records, 8 attributes and its $|\mathcal{X}(D)|$ =436,414.

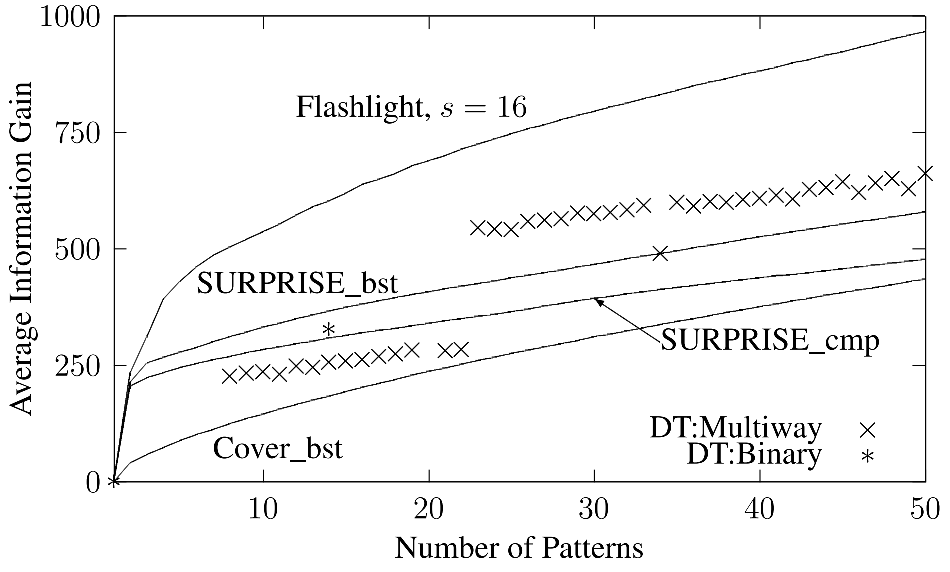
- `Income`: U.S. Census data containing demographic attributes such as the number of children and marital status, and a binary outcome attribute indicating whether the given person's income exceeded 100,000 dollars. This data set was downloaded from IPUMS-USA [47] at `https://usa.ipums.org/usa/data.shtml` and contains roughly 1.5 million records, 9 attributes and 78 million patterns in its data cube.

Throughout this section, we measure the information content of Explanation Tables and other summaries in terms of the average information gain. As we explained in Section 4.2.4, this is the improvement in D_{KL} compared to having only the all-wildcards pattern and knowing only the fraction of records in the whole data set having outcome 1.

4.4.2 Related Approaches

In this section, we compare Explanation Tables with three related approaches: Decision Trees, the SURPRISE operator for data cube exploration [49], and Cover Summaries, which is the summarization technique from [23, 24] that finds the fewest patterns covering most of the records with outcome 1.

Decision Trees: We used the Weka J48 implementation [27] of the C4.5 algorithm to represent Decision Trees, which provides confidence parameters that control pruning and the number of



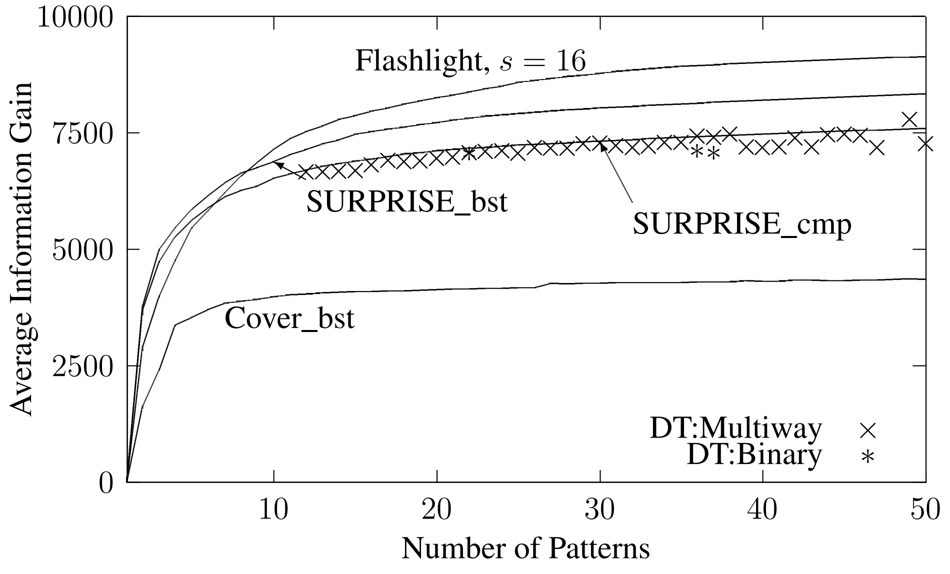


Figure 4.6: Quality of Explanation Tables vs. different methods using `Upgrade` (top) and `Adult` (bottom).

records a leaf node can cover. We consider both multi-way split Decision Trees in which a node can have many children (DT:Multiway) and Binary Split Decision Trees in which a node has exactly two children (DT:Binary). Since there is no direct way to enforce Decision Tree size in Weka, a common approach for finding the best Decision Tree parameters is to perform a grid search over the parameter space. In our evaluations, we varied confidence in the range [0.01–0.59] and the minimum number of records per leaf in the range [2–20]. To compare the information content of Decision Trees and Explanation Tables for the same Explanation Tables sizes ($|T|$), we counted the number of leaves in the generated trees, i.e., we consider every path from a leaf node to the root as a pattern. In addition, we report the information gain as computed by Weka. As a sanity check, we applied the same method we use to compute the information gain for Explanation Tables to DT:Multiway. We considered paths from leaves to nodes as patterns, then, applied iterative scaling to generate ε^*, then, computed $D_{KL}(y\|\varepsilon^*)$, which yielded the same information gain values.

SURPRISE operator: Similar to Explanation Tables, the SURPRISE operator attempts to find the most informative patterns in the data cube. However, the SURPRISE operator does not allow partially overlapping patterns—similar to Decision tree. As shown in [48], a bottom-up dynamic programming algorithm can be used to compute optimal (in terms of information content) nonoverlapping summaries for a given size (n). The main premise behind the algorithm introduced in [48] is that the database D can be split into two (D_1 and D_2), and the most informative—non-overlapping—summary for D can be found by combining two summaries; one summary of size $n - m$ for D_1 and another summary of size m for D_2. Notice that this does not hold for non-overlapping summaries. The starting point of the algorithm is the bottommost level of the data cube, i.e., the records of the data set. At this point, the most informative summary is the pattern which represents the database records. Then, the algorithm considers the next level, i.e., patterns with a single *, then patterns with two *, etc., up to the highest most level which contains a single pattern (the all wildcard pattern). To avoid many unnecessary re-computations, a tree structure is maintained. For efficiency, an accuracy threshold (ϵ) is applied, which is used for pruning the tree, i.e., patterns that are not very promising are removed at earlier stages. A lower value of ϵ indicates a finer algorithm resolution but slower running time. To fairly represent SURPRISE, we have chosen two values for each data set: SURPRISE_cmp, which uses ϵ that gives comparable running time to Flashlight, and SURPRISE_bst, which uses ϵ that yields the best possible information gain given our CPU and memory resources.

Cover Summaries Similar to Explanation Tables, the summaries from [23, 24] allow overlapping patterns. A key result from [23] shows that Cover summaries that find the minimum number of patterns from the data cube that satisfy a certain condition up to a certain confidence threshold is NP-Hard. However, Golab et al. [23] provide a near optimal approximation algorithm which we use in our evaluation (we use the implementation provided by the authors). Adapted to our problem, we fix the condition to be $y = 1$. In our evaluations, we vary the confidence threshold to the following {0.4, 0.5, . . . , 0.9, 0.95, 0.99}, and report the running time of the one with the best information gain (Cover_bst). After generating the summaries, we apply the maximum entropy principle to derive $\varepsilon^*(t)$ for each record t in D and compute information gain.

Explanation Tables We evaluate Flashlight with a sample size of 16, which we observed to offer a good trade-off between running time and information gain (recall Figure 4.3). The reported running times and information gains are averages of five runs with different samples. In our evaluations, we observed that the information gain did not vary greatly by varying the samples, using the sample size of 16. The minimum and maximum information gains are within 3% from the average information gain for the three data sets.

Figure 4.6 shows the average information gain using `Upgrade` and `Adult` as a function of the number of patterns (or, in the case of Decision Trees, the number of leaves). DT:Multiway and DT:Binary are represented as individual points on the graphs, corresponding to the numbers of leaves we were able to obtain using the grid search explained above. Flashlight outperforms all other approaches except for small numbers of patterns (below 8) using the `Adult` data set, in which case SURPRISE has a slightly higher information gain. Upon further inspection, we observed that the problem is with the gain formula from Equation 4.3. The SURPRISE operator computes the exact KL-divergence for candidate patterns because it is feasible for non-overlapping patterns. One of the first patterns chosen by SURPRISE_bst gives a high improvement in KL-divergence, but a relatively modest gain, as calculated by Equation 4.3, and therefore is not chosen by Flashlight. We omit the results for the largest data set, `Income`, because many of the competing techniques ran out of memory.

Table 4.12 shows the corresponding running times; assuming each technique is required to generate 50 patterns. For DT:Multiway and DT:Binary, we report the total running time of the grid search since it is not possible to directly control Decision tree size. "N/A" indicates that the algorithm ran out of memory and did not produce any output. In this section, we showed using real data sets that Flashlight offers the best information gain, running time and scalability out of the tested algorithms with a reasonably small sample size (e.g., 16).

4.4.3 Comparison of Algorithms for Generating Explanation Tables

Running Time

We start by showing the efficiency improvement of Flashlight as compared to SampleCube $gain_D$ and Baseline. Figure 4.7 shows the average running time (over five runs with different random sample sizes) of Flashlight and a traditional plan corresponding to the plan for Query 3 chosen by PostgreSQL, with the sample size on the x-axis. In our evaluations, we will refer to Query 3 as *Traditional Plan*. For comparison, the horizontal lines at the top of the figures show the running time for Baseline, which considers all possible patterns. The three figures shown in Figure 4.7 correspond to `Upgrade` with $|T|$ = 50, `Adult` with $|T|$ = 50, and `Income` with $|T|$ = 20 (from top to bottom). We chose a smaller explanation table size for `Income` in order to keep the total running time of our evaluations manageable. The performance improvement of Flashlight over the traditional plan is significant especially for the larger sample sizes where Flashlight becomes several orders of magnitude faster. As a reference point, we include the running time of Baseline, which does not use sampling, shown as a horizontal line. Our evaluations also show the savings of Flashlight compared to Baseline. For instance, for `Income` Flashlight takes 108 s per pattern while Baseline takes 10,000 s which is a 93$\times$ improvement in the running time with comparable information gain.

Savings of Laserlight execution strategy: We also compared the running time of Laserlight and SampleCube $gain_s$, expecting the former to be much faster because its space of candidate patterns is smaller. We found that Laserlight was several times faster for sample sizes under 1000. For instance, using `Upgrade` and a sample size of 256, Laserlight took under 40 seconds while SampleCube $gain_s$ took 141 seconds.

Running Time vs. Information Gain

In the following set of experiments, we consider both the running time ("cost") as well as the information gain ("quality") for the proposed algorithms. To calculate the best possible information gain, we ran each algorithm with different sample sizes between 2 and 8192, and for each sample size we generated Explanation Tables of sizes between one and 50 patterns for `Upgrade` and `Adult` and between 1 and 20 for `Income`. Figure 4.8 show the best average information gain (averaged over five runs with different samples) for various running time budgets. The figure shows `Upgrade` at the top, then `Adult`, then `Income` at the bottom. Each point represents a combination of $|T|$ and $|s|$ that gave the best information gain at that running time budget for the given algorithm (labelled Flashlight_bst, Laserlight_bst and SampleCube_bst). We also plot separately a line for Flashlight with a sample size of 16, on which each point corresponds to a different number of patterns ($|T|$), to show that Flashlight performs well even at this small sample size. For comparison we also plot horizontal lines to indicate the information gain of Baseline, but we do not show its running time which was extremely high as we have shown in Figure 4.7.

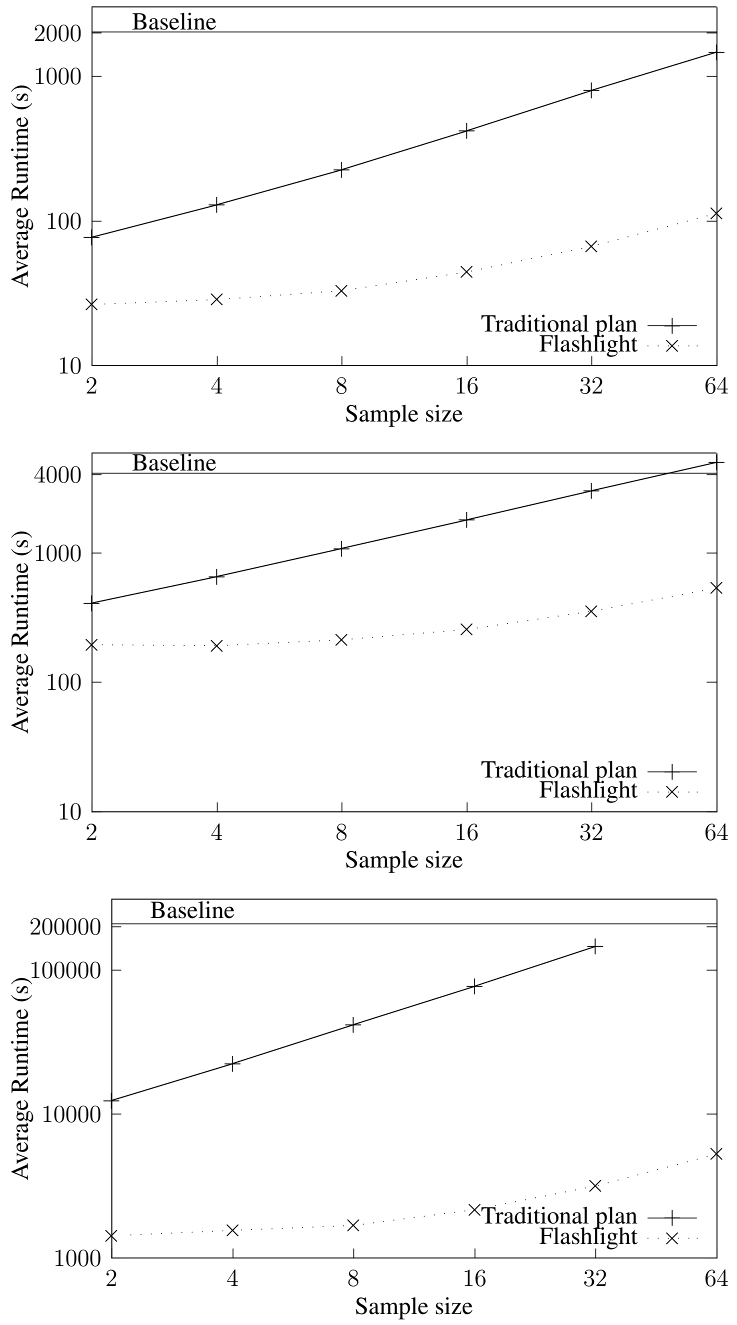
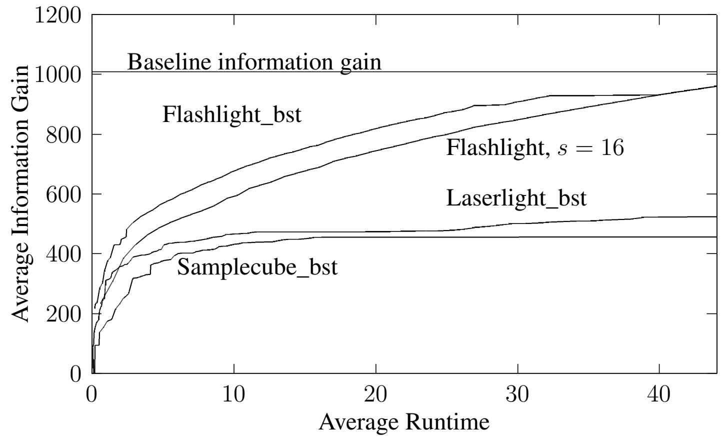
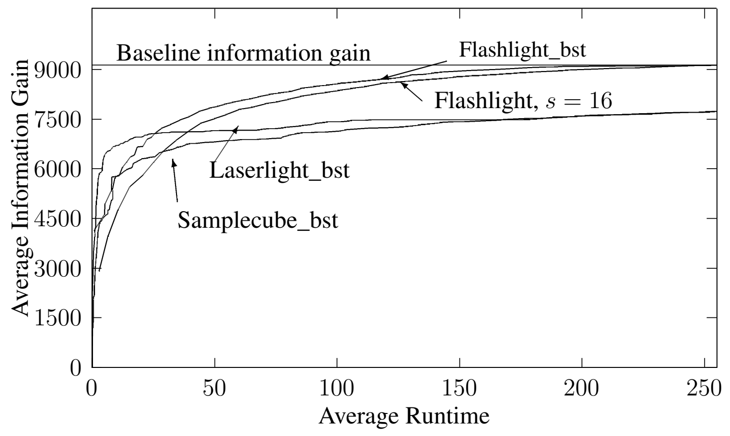


Figure 4.7: Flashlight compared to traditional execution plan using `Upgrade`, `Adult`, and `Income` (from top to bottom), $|T| = 50$ for `Upgrade` and `Adult` and $|T| = 20$ for `Income`, average of 5 runs.





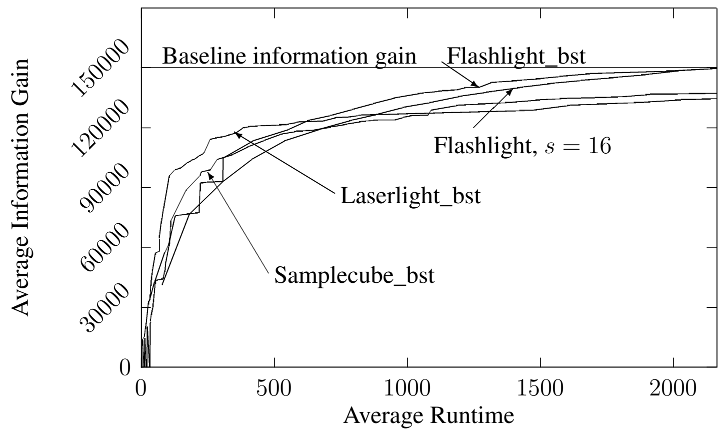


Figure 4.8: Information gain vs. running time for all algorithms using different parameter combinations. For `Upgrade` and `Adult` $|T| = 50$ and for `Income` $|T| = 20$.

We make two observations from Figure 4.8. First, the information gain of Flashlight is comparable to that of Baseline for large running times (i.e., large $|T|$); however, Flashlight is significantly faster. In fact, this is also true for a small sample of 16. Second, for sufficiently large running times, Flashlight offers the best trade-off in running time versus information gain. Only for smaller running times and for the larger data sets does Laserlight become the method of choice. Specifically, Laserlight wins for running times under 28 seconds using `Adult` and running times under 619 seconds using `Income`. Finally, in our evaluations, SampleCube $gain_s$ was never the preferred method. Based on our evaluations, we recommend using Flashlight for most use cases and only using Laserlight for use cases where speed is more important than accuracy, for example if the time budget is small.

Scalability

In order to investigate the scalability of Flashlight and Laserlight with respect to the data set size and the number of dimensions, we use the data set with the largest number of records and attributes, which is `Income`. To characterize the effect of the number of records ($|D|$), we create smaller versions of `Income` by sampling from it. We ensure a containment relationship among the samples of `Income`, e.g., all the records in the 50% sample are included in the 60% sample. Figure 4.9 shows the effect of varying the size of the portion of `Income` Flashlight with $|s| = 16$ and $|T| = 20$ and Laserlight with $|s| = 512$ and $|T| = 20$. The running time of Flashlight grows linearly with $|D|$ since one of its steps needs to compute $LCA(s, D)$. In addition, the running time of Laserlight grows very slowly because it operates over s and not D. We note that the little growth in the running time of Laserlight is because it draws a new sample from D at every iteration and the evaluated implementation scans D to generate a new sample. In the final experiment, we vary the number of dimensions (d). Again, we take `Income` and project out one dimension at a time. Results are shown in Figure 4.9 (bottom). Laserlight is not affected by increasing the number of dimensions, but the running time of Flashlight increases roughly linearly. This is because $|\mathcal{X}(s)|$, which is the candidate pattern space of Flashlight grows more rapidly with d than $|LCA(s, s)|$, which is the candidate pattern space of Laserlight.

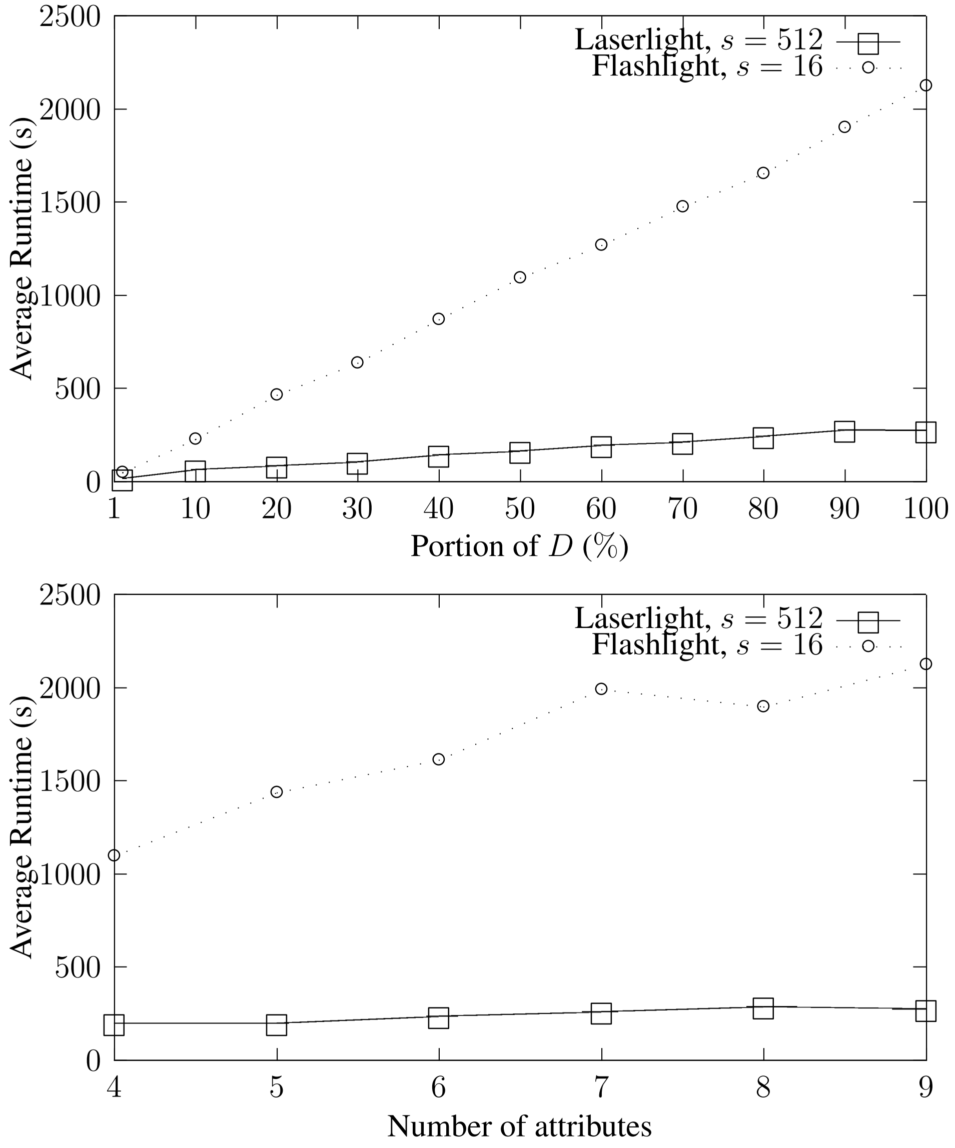


Figure 4.9: Effect of varying the size of the database (top) and varying the number of dimension attributes (bottom) on Explanation Tables running time using `Income`.

4.5 In-Browser Implementation

As we have demonstrated recently [17], we were able to run Explanation Tables in the browser in order to guide users to the most informative patterns in the data cube. Back to our running example, using `Salary`, Figure 4.10, shows an explanation table generated in-browser. As discussed in Chapter 1, while in-browser analytics are easy to use by everyone since they eliminate any external dependencies that might require a technical background, it brings performance challenges because of the limitations of the browser environment. The browser imposes the following two main limitations: slower computation and limited memory. In this section, we present our adaptation to Flashlight that makes use of some of the optimizations and lessons learned from Chapter 3 to generate Explanation Tables inside the browser. Namely, our optimizations are based on employing Afterburner's columnar storage that is based on typed arrays and a newly introduced data structure that computes Flashlight, which uses mapping to compute aggregates quickly.

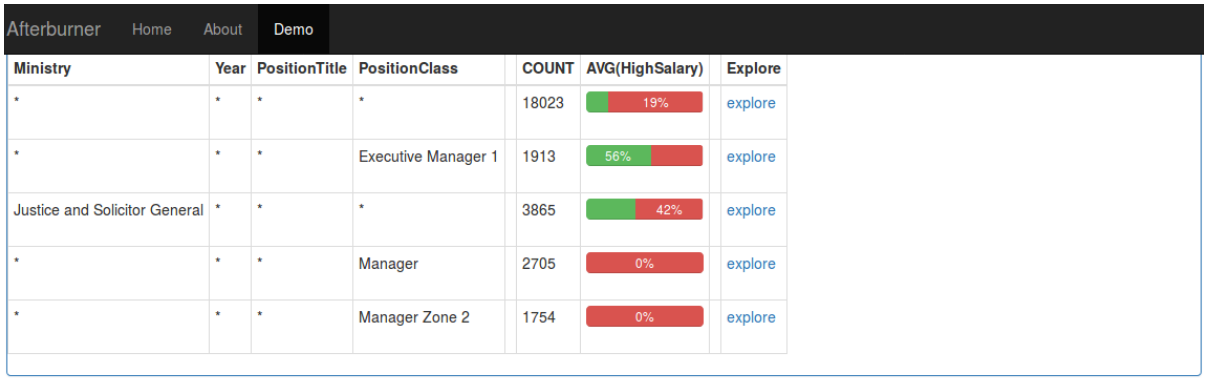
4.5.1 Main Challenges

In our preliminary evaluations, we have tested an implementation of Flashlight based on JavaScript objects. As expected, based on our microbenchmarks from Section 3.4.1, JavaScript objects did not perform well. For example, this implementation failed to generate the first pattern using our largest data set (`Income`) on the same client setup described in Section 5.4 since the browser tab crashed.

Hence, our JavaScript implementation of Flashlight uses the in-memory storage of Afterburner instead of JavaScript objects. To address the slower speed inside the browser, we introduce a new data structure to Afterburner which accelerates computing Query 4. Specifically, the new optimization targets how we compute aggregates over the data cube of a sample. According to our initial evaluations, the optimizations have shown to be promising in terms of speed and memory footprint. For example, in our evaluation, the data structure requires less than 1 MB of additional memory and roughly 1 minute to generate an Explanation Table with 5 patterns using `Income`.

4.5.2 In-Browser Optimizations

To be able to run Flashlight in the browser we introduced a new data structure to Afterburner, which we call *Mnemonic Tables*. The new data structure depends on Afterburner's columnar in memory storage, which is based on JavaScript typed arrays. Similar to Afterburner's physical



Ministry	Year	PositionTitle	PositionClass	COUNT	AVG(HighSalary)	Explore
*	*	*	*	18023	19%	explore
*	*	*	Executive Manager 1	1913	56%	explore
Justice and Solicitor General	*	*	*	3865	42%	explore
*	*	*	Manager	2705	0%	explore
*	*	*	Manager Zone 2	1754	0%	explore

Figure 4.10: In the browser explanation table using `Salary`.

operators, Mnemonic Tables do not require materializing intermediate results in order to minimize memory footprint. In addition, Mnemonic Tables use a direct mapping (instead of hash tables) to compute aggregates, which comes with the following two benefits. First, it eliminates the need for computing hash values per insertion (Flashlight requires $2^d \times |D|$ insertions in the worst case). Second, aggregate values of related patterns are stored in close by memory locations which increases locality of reference. The Mnemonic Table uses a mapping that is based on the format of the pattern and the sample record that generated it to compute aggregates over $\mathcal{X}(s)$, instead of using hash-based aggregates. So, it is as if each pattern can use a simple Mnemonic device to figure where its aggregate counters are stored in-memory instead of the more expensive hash-based "routing".

Recall that Flashlight computes $gain_D$ over the patterns in the data cube of a sample (i.e., $\mathcal{X}(s)$), and recall that computing gain depends on three aggregate values which are: COUNT(*), SUM($D.y$), and SUM($D.\varepsilon^*$). A straightforward implementation of a physical operator that computes these aggregates is a nested loop cross product between s and D (to generate $LCA(s, D)$) followed by a data cube operator and a hash-based GROUP-BY operator. However, the GROUP-BY computation over the data cube of the cross product would be challenging inside the browser because of the limitations in the available memory and the slower runtime. Thus, we introduce the following optimized data structure.

Mnemonic Tables: The main intuition behind the Mnemonic Tables depends on noticing that: $|\mathcal{X}(s)| \leq 2^d \times |s|$. Thus, we prepare a typed array with the size of $2^d \times |s|$ for each of the three required aggregate values. Applied to `Workout` from Table 4.2 and the three record sample from Table 4.4, $2^d \times |s| = 2^3 \times 3 = 24$. Notice that $|\mathcal{X}(s)| = 20$ as shown in Table 4.7. This is because some of the patterns in $\mathcal{X}(s)$ can be generated from multiple sample records. In this example, (*, *, *) is generated three times, (*, *, Banana) twice, and (*, Dawn, *) twice. The mapping between candidate patterns and their associated sample can be illustrated using the following table:

psid	(Sun, Morning, Banana)	(Thu, Dawn, Oatmeal)	(Sat, Dawn, Banana)
000	(*, *, *)	(*, *, *)	(*, *, *)
001	(*, *, Banana)	(*, *, Oatmeal)	(*, *, Banana)
010	(*, Morning, *)	(*, Dawn, *)	(*, Dawn, *)
011	(*, Morning, Banana)	(*, Dawn, Oatmeal)	(*, Dawn, Banana)
100	(Sun, *, *)	(Thu, *, *)	(Sat, *, *)
101	(Sun, *, Banana)	(Thu, *, Oatmeal)	(Sat, *, Banana)
110	(Sun, Morning, *)	(Thu, Dawn, *)	(Sat, Dawn, *)
111	(Sun, Morning, Banana)	(Thu, Dawn, Oatmeal)	(Sat, Dawn, Banana)

The first columns in the table show a numeric identifier in binary representation, which we call `psid`. Next, each column shows a record from the sample and the candidate patterns in its data cube. Notice the mapping between the form of the candidate patterns and `psid`; if we replace every * with a `zero` and every literal value with a `one`. For example, consider the first sample record (Sun, Morning, Banana), its sixth candidate pattern (`psid` = 101) takes the form ("literal",*,"literal").

Mnemonic Tables take advantage of this mapping to aggregate values for the candidate patterns. Continuing with the same example, Figure 4.11 illustrates the Mnemonic Tables we use to compute the aggregates over the data cube of the sample—which should yield the same aggregate values computed in Table 4.7. On the right side, with solid borders, three typed arrays storing the aggregate values. And on the left side, with dotted borders, columns that illustrate the idea behind the mapping, i.e., these values are shown here for clarity and are not stored in the mapping table.

In the same figure, the group of columns called patterns shows all the candidate patterns in $\mathcal{X}(s)$—including duplicates. Next, the column called `srid` indicates the record id in the sample (starting from 0). For example, the pattern (Sun, *, Banana) is generated by the first record in the sample and its `srid` is 0. The next column shows the `psid` of the pattern. Then the next column shows the array indices in binary format. Given a candidate pattern, we concatenate its `srid` to its `psid` in order to find its associated array index. For example, the pattern (Sun, *, Banana) has the array index of 00101, and thus its associated aggregates are stored in the sixth element of the arrays.

Finally, we scan the three arrays once to find the array index of the candidate pattern with highest gain estimation. After finding the array index with the maximum estimated information gain, we use the same mapping to recover the pattern—recall that we do not store the patterns. This is done by splitting the array index into two parts: `srid` which identifies the sample record and `psid` which is used to recover the form of the pattern. In our example, the array index of

patterns			srid	psid	array offset (binary)	count array	$Y_D(x)$ array	$E_D(x)$ array
*	*	*	00	000	00000	14	.5	.5
*	*	Banana	00	001	00001	4	.75	.5
*	Morning	*	00	010	00010	2	.5	.5
*	Morning	Banana	00	011	00011	1	1	.5
Sun	*	*	00	100	00100	2	1	.5
Sun	*	Banana	00	101	00101	1	1	.5
Sun	Morning	*	00	110	00110	1	1	.5
Sun	Morning	Banana	00	111	00111	1	1	.5
*	*	*	01	000	01000	14	.5	.5
*	*	Oatmeal	01	001	01001	4	.75	.5
*	Dawn	*	01	010	01010	4	.5	.5
*	Dawn	Oatmeal	01	011	01011	2	.5	.5
Thu	*	*	01	100	01100	1	1	.5
Thu	*	Oatmeal	01	101	01101	1	1	.5
Thu	Dawn	*	01	110	01110	1	1	.5
Thu	Dawn	Oatmeal	01	111	01111	1	1	.5
*	*	*	10	000	10000	14	.5	.5
*	*	Banana	10	001	10001	4	.75	.5
*	Dawn	*	10	010	10010	4	.5	.5
*	Dawn	Banana	10	011	10011	2	.5	.5
Sat	*	*	10	100	10100	5	0	.5
Sat	*	Banana	10	101	10101	1	0	.5
Sat	Dawn	*	10	110	10110	2	0	.5
Sat	Dawn	Banana	10	111	10111	1	0	.5

Figure 4.11: Illustration of the physical in-memory representation of the Mnemonic Tables.

10100 with prefix 10 and suffix 100 has the highest gain. The prefix 10 refers to the third sample record, which is (Sat, Dawn, Banana). The suffix 100 is used to recover the form of the pattern (Sat, *, *), which is added to the explanation table as per Algorithm 1.

Additional discussion. In this section, we introduced Mnemonic Tables in order to accelerate computing Flashlight in the browser. Other than accelerating Flashlight, we expect that Mnemonic Tables can accelerate other data cube operations in an RDBMS. Namely, Mnemonic

Tables can be used to accelerate tasks where the data cube or a part of the data cube is required to be computed, for example, in cases where data cube pruning techniques such as [11, 55] are not applicable.

4.5.3 In-Browser Flashlight

In this section, we present our in-browser implementation of Explanation Tables, which we refer to as `ETJS`. Our in-browser implementation of Explanation Tables realizes Algorithm 1 using vanilla JavaScript, i.e., does not use asm.js optimizations. However, `ETJS` makes use of Afterburner's typed array storage in order to access the data set and uses Mnemonic Tables to compute Flashlight.

Instead of SQL, `ETJS` uses imperative JavaScript code to realize Query 4 (i.e., Flashlight) and Algorithm 2 highlights the main idea behind the code. Recall that the main idea behind Flashlight is picking the pattern with the highest information gain estimate using Equation 4.3. Recall also that $gain$ required three aggregate values per pattern which are, the COUNT(*), SUM($D.y$), and SUM($D.\varepsilon^*$). Lines 1–3 initialize the arrays used for computing the three aggregates. Lines 4–6 generate all the patterns in the LCA cross product of the sample and the database (i.e., $LCA(s, D)$) by calling the method `lcapbin` in a nested loop. Instead of returning a materialized pattern, the method `lcapbin` returns the numeric identifier of the pattern, i.e., its `psid`. For example, `lcapbin`((Fri, Dawn, Banana),(Fri, Night, Salad)) returns 100. The loop in line 7 generates the `psid` of all the patterns in the data cube of the LCA cross product. This is possible since the `psid` reserves the form of the pattern. For example, the data cube of the pattern with `psid` of 101 contains four patterns with `psid`'s of: 101, 001, 100, and 000, respectively. Line 8 computes the array offset which corresponds to each candidate pattern by concatenating the sample identifier (`srid`) to the `psid`. Lines 9–11 compute the three aggregate values that are required to estimate the information gain over all the patterns in $\mathcal{X}(s)$.

After computing the aggregate values, the pattern with the highest information gain estimation can be found using a sequential scan of the aggregate values. Lines 12–13 initialize the variables `maxpsid` and `maxgain` that are used to keep track of the `psid` and estimated information gain of the best pattern so far. Lines 14–18 loop over all of the aggregated values. Finally, line 19 uses the direct mapping which we described in Section 4.5.2 to translate the `psid` into a materialized pattern.

4.5.4 Evaluation

In this section, we evaluate `ETJS` and compare it to the in-RDBMS implementation of Flashlight. For the in-RDBMS implementation, we use the same implementation of Explanation Tables used

Algorithm 2 In-Browser Flashlight

```
Input: Database $D$, Sample $s$, Number of dimentions $d$
 1: count[aoff] $\leftarrow$ [0, $\cdots$, 0]
 2: sumYD[aoff] $\leftarrow$ [0, $\cdots$, 0]
 3: sumED[aoff] $\leftarrow$ [0, $\cdots$, 0]
 4: for $t_1 \in D$:
 5:   for $t_2 \in s$:
 6:     lcapb $\leftarrow$ lcapbin($t_1$, $t_2$):
 7:     for psid $\in \mathcal{X}$( lcapb):
 8:       aoff $\leftarrow$ $t_2$.id ++ psid
 9:       count[aoff] $\leftarrow$ count[aoff] + 1
10:       sumYD[aoff] $\leftarrow$ sumYD[aoff] + $t_1.y$
11:       sumED[aoff] $\leftarrow$ sumED[aoff] + $t_1.\varepsilon^*$
12: maxpsid $\leftarrow$ 0
13: maxgain $\leftarrow$ 0
14: for i $\in$ {0, $\cdots$, $2^d \times |s|$ - 1}:
15:   currgain $\leftarrow$ gain(count[i], sumYD[i], sumED[i])
16:   if currgain $>$ maxgain:
17:     maxpsid $\leftarrow$ i
18:     maxgain $\leftarrow$ currgain
19: return psidToPattern(maxpsid)
```

in Section 4.4, which we refer to as `ETPG`. For this set of experiments we use a desktop with a 2.7 GHz Intel i5-5250U processor (4 cores, 6 MB of cache) and 8 GB of RAM, running Ubuntu 16. `ETJS` ran inside Mozilla Firefox (v51) and `ETPG` ran inside PostgreSQL (v9.5).

Table 4.13 shows the average running time to generate Explanation Tables over ten trials using a sample size of 16. For `Upgrade` and `Adult` we used Explanation Table size of 50 while for `Income` we used an Explanation Table size of 20. Again, we used a smaller Explanation Table size for the largest data set in order to minimize the total running time of our experiments. For all the data sets `ETJS` is faster than `ETPG` with factors of 16$\times$, 9$\times$, and 5$\times$ for `Upgrade`, `Adult`, and `Income`, respectively. We expected `ETJS` to be faster despite using vanilla JavaScript because of the saving of the Mnemonic operator. We also note that Query 4 is not the only bottleneck, since at each iteration of Algorithm 1 also computes iterative scaling. We also expected that the relative benifits of using Mnemonic Tables to decrease for larger data sets since the total time for Flashlight decrease compared to iterative scaling in larger databases.

	Upgrade	**Adult**	**Income**
`ETJS`	2.1	21.0	306.1
`ETPG`	34.1	200.0	1550.9

Table 4.13: Average running time (s) over ten trials to generate an explanation table of sizes 50, 50 and 20 for `Upgrade`, `Adult`, and `Income` with s =16.

These results are promising since they show the viability of running Explanation Tables in the browser for data sets that are as large as `Income`, with 1.5 million records. Furthermore, for the smallest two data sets, the time to generate a pattern in the browser is less than one second. This shows the viability of using information theory to guide interactive data exploration in the browser.

In this chapter, we have motivated using Explanation Tables to guide users through data cube exploration tasks. We have presented the necessary background in order to describe the algorithms which we used in our evaluations. Our evaluations showed the efficiency of Flashlight compared to Baseline and that Explanation Tables are more informative compared to other existing techniques. Finally, we have presented and evaluated a new data structure that adapts Explanation tables to the browser. Our evaluations demonstrated the viability of running Explanation Tables in the browser without any external dependencies for modestly sized data sets.

Chapter 5

Accelerating Interactive SQL Data Exploration

5.1 Introduction

In the previous chapters, we have proposed techniques that tackle the main two challenges that would face non-technical users (everyone) who want to analyze an open data set with little previous experience in data analysis techniques. In Chapter 3 we introduced Afterburner our in-browser SQL engine that solves the first challenge (managing infrastructure) with a performance that rivals native analytics. Our results detailed in Chapter 3 are exciting because to our knowledge we are the first to show the viability of in-browser SQL analytics. These results encouraged us to revisit deployment architectures for interactive SQL data exploration tasks. In our previous chapters, we have focused on modestly-sized databases. In this chapter, however, we focus on larger databases that cannot fit entirely on the client side [18]. Namely, we focus on accelerating the latency of analytical SQL queries over larger databases that reside in a cloud-based backend. Techniques that optimize interactive SQL data exploration currently exist; however, our proposed system builds on our proposed in-browser SQL engine to provide a simple yet optimized architecture. In addition, we propose an intuitive technique for creating materialized views that supports the ad-hoc nature of data exploration. We compare several alternative deployment architectures to our proposed architecture and show the benefits of ours in terms of performance and simplicity.

We consider a new deployment architecture for interactive SQL data exploration. Our proposed architecture is characterized by splitting queries into two: a onetime query that runs at the backend and repeated queries that run in the front-end inside the web browser. Our technique accelerates interactive SQL data exploration tasks using an intuitive and a simple to use layer since it generates a onetime backend query which ships a materialized view to the front-end. Then, it uses query re-writing to optimize subsequent queries against the local materialized view. The front-end queries run inside the web browser without any external dependencies. Thus, the technique accelerates the latency of queries without introducing any additional complexity such as additional administration tasks at the backend or complicating client management to accommodate for an RDBMS at the frontend. Furthermore, our system transparently manages both creating the materialized views and subsequent query rewrites at the frontend; thus data scientists do not need to be burdened with query re-writing. While other techniques for recommending materialized views exist, our hint based system is more suited for ad-hoc data exploration compared to others since it is based on hints provided by the user, i.e., can be created on demand.

5.1.1 Main Contributions

Contributions. We view this chapter as having two main contributions:

- Propose a new deployment architecture for interactive data exploration and compare it to alternative architectures empirically demonstrating its advantages. Our proposed architecture is novel since we are the first to propose optimizing queries using an in-browser SQL engine.
- Outline an in-browser query acceleration layer which supports a wide class of interactive SQL analytics. The novel technique allows data scientist to provide a hint regarding the focus of data exploration, and our system automatically splits query execution across the backend and the browser. Our approach is novel since users can accelerate queries on-demand using a high-level hint.

5.2 Deployment Architectures for Interactive Data Exploration

5.2.1 Client-Server Architecture for Interactive Data Exploration

Data science applications have been influenced greatly by two recent trends: browser-based tools and cloud-based backends. Data scientists are increasingly depending on the browser in their data exploration tasks, especially for interactive analytics on large data sets. While previously they depended mainly on command-line and REPL-based tools, today they are increasingly moving into browser-based notebooks such as Jupyter.[1] Browser-based tools have gained popularity due to their tight integration of both the code and the output. This simplifies the analytical process greatly since the notebook itself can be serialized, reloaded, and shared. Recently, Notebooks are becoming a popular frontend layer for analytics tools such as the existing software ecosystems of R and Python that enable data scientists to access a broad suite of data management tools. The other current trend is the popularity of using the cloud as a backend for analytics. Currently, backend analytics are increasingly centralized into the cloud, exemplified by various database-as-a-service offerings (Amazon Aurora and RDS, Azure SQL, etc.), fully-managed data analytics stacks (Google BigQuery), as well as more general dataflow frameworks (Google Cloud Dataflow, Amazon Spark EMR).

Given these two trends, the standard deployment architecture follows the pattern of client-server which consists of a browser frontend connected to a cloud backend. In all existing implementations that we are aware of, the browser is simply a dumb rendering endpoint, i.e., all query execution is handled by backend servers residing in the cloud. However, as we have shown in Chapter 3, modern browsers are capable of supporting SQL analytics at latencies that are comparable to native engines.

We propose a novel approach based on split-execution for supporting interactive SQL analytics in the cloud, focusing on the common scenario in which a data scientist works with a "query template" and issues a sequence of queries that vary only in the predicates on certain columns. We outline our system that depends on a simple hint from the data scientist. Our approach is novel because it allows users to accelerate queries on-demand based on a high-level hint. Thus, when the user asks to optimize a new query during her data exploration, our system uses the simple hint provided by the user to automatically split queries into two parts: a backend query that generates a materialized view and a lightweight query against this view. Our system is novel in that the materialized view is shipped to the browser so that all subsequent interactions occur locally.

The main goal of our proposed architecture is optimizing the performance of interactive SQL exploration of backend databases without adding any complexity to the deployment architecture. First, our system seamlessly optimizes queries using materialized views, bringing the latency down for a more responsive interactive experience. The materialized views are based on user hints which is effective for ad-hoc data exploration scenarios since it allows for optimizing the performance of new queries that the system did not see before—we compare our hint based optimization to existing techniques for recommending materialized views in more details in Chapter 2. Second, using Afterburner allows for a simpler architecture that does not incur administrative burdens on the backend or require a complex frontend deployment. In addition, our system decouples the analytics applications from the backend which has several benefits

[1] http://jupyter.org/

such as removing network roundtrip latency per user interaction, working offline when a connection is not available, isolating the performance of the workloads of the different data scientists, offloading work from the backend.

In this chapter, we consider and discuss both the latency and complexity of our proposed deployment architecture compared to others. We also present the design of our proposed hint based system for accelerating SQL queries. Our experiments show that our proposed deployment architecture offers a new design point with optimized latencies that match the more complex deployment architectures but at simplicity that rivals the simplest architecture.

In this section, we discussed the main reasons behind the popularity of browsers as de facto frontend for analytics tasks such as SQL data exploration. Hence, we focus on the browser as a client. We also highlighted the main idea about our proposed deployment architecture and proposed tool. In the next section, we outline an example data exploration scenario as a concrete use case. In addition, we discuss and compare different alternative deployment scenarios for data exploration.

5.2.2 Alternative Deployment Architectures for Interactive Data Exploration

The work of data scientists often involves tight interaction cycles with the data, particularly for exploratory tasks. This chapter focuses on and optimizes for the common scenario where a data scientist rapidly issues a sequence of SQL queries that differ only in the predicates in the WHERE clause. As a running example, consider Q6 from the widely used TPC-H benchmark for decision support in data warehousing. Here, we directly quote from the benchmark definition:

> This query quantifies the amount of revenue increase that would have resulted from eliminating certain companywide discounts in a given percentage range in a given year. Asking this type of "what if" query can be used to look for ways to increase revenues.

This query is shown in Figure 5.1 for clarity. As part of the data exploration, a data scientist would be interested in the results of the same "query template", but with different predicates on the WHERE clause: different date ranges, different levels of discount, etc. In fact, this is essentially what happens in dashboards and report generators for pre-specified query templates, although interactive data exploration requires support for ad hoc query templates.

To support such data exploration, we desire three important properties: low latency, i.e., queries run fast, simplicity of frontend deployment, i.e., minimizing the effort of data scientists

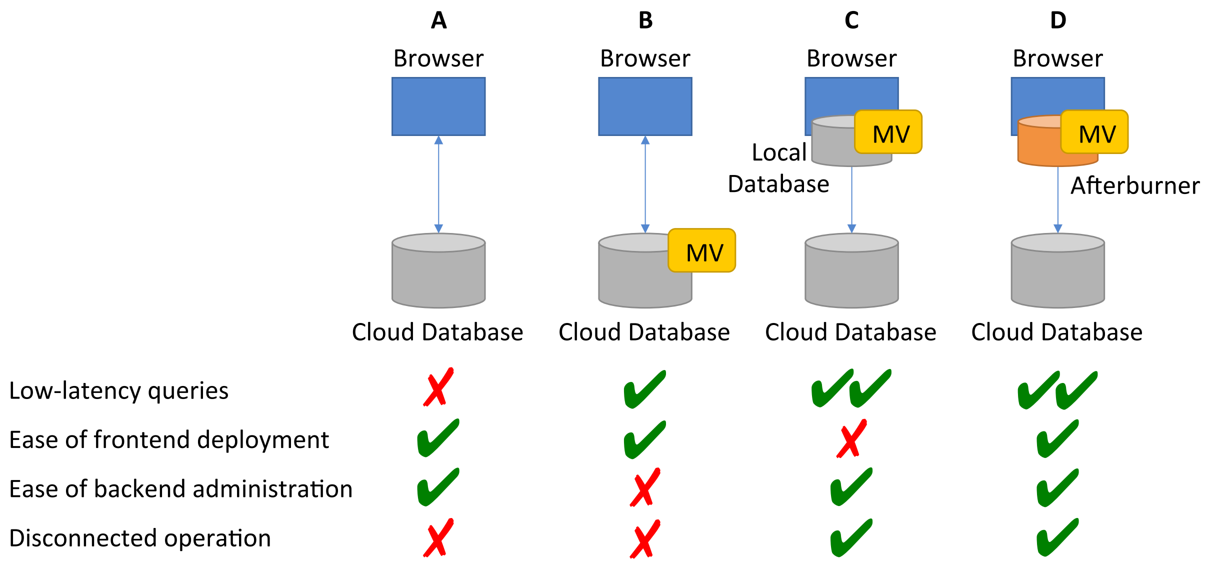
```
SELECT SUM(l_extendedprice * l_discount)
       AS revenue
FROM   lineitem
WHERE  l_shipdate >= DATE '1994-01-01'
   AND l_shipdate < DATE '1995-01-01'
   AND l_discount BETWEEN 0.05 and 0.07
   AND l_quantity < 24;
```

Figure 5.1: Query 6 from the TPC-H benchmark for decision support in data warehousing.

in managing their own client setup, and simplicity of backend administration, i.e., minimizing the effort of data warehouse administrators to support data scientists. With this in mind, let us consider a number of deployment scenarios, summarized in Figure 5.2:

Deployment A. This serves as the reference architecture described earlier in Section 5.2.1, with a browser client connected to a cloud analytics backend. More generically, the browser frontend can be a fully-interactive notebook, an interactive dashboard, or some visual business intelligence tool. For concreteness, we assume the backend is a cloud-based analytical RDBMS, although it is a generic stand-in for an analytics backend (for instance, a Spark cluster). With this setup, the data scientist re-issues the same query template with different bindings, leading to sub-optimal query latencies because each SQL query is treated independently. However, frontend deployment is simple since the client is just a browser, as is backend administration since the RDBMS does not need to maintain transient state.

Deployment B. An obvious improvement to Deployment A is to accelerate analytics using a materialized view (MV). Using materialized views to optimize analytical queries is an effective and well known technique that reduces query latency. Compared to Deployment A, such an optimization retains the simplicity of the frontend deployment since it does not alter how the client is deployed, but it introduces new administrative burdens on the backend. We mention a few example system administration tasks that this deployment might imply: firstly, managing user roles that define which data scientists are able to create materialized views; secondly, managing user quotas that ensure equitable use of backend resources; thirdly, managing storage space of the materialized views, i.e., deciding whether the materialized views are transient or can they persist over long periods of time (managing transient materialized views adds an extra level of complexity since it requires deciding on the mechanisms by which they are garbage collected). Moreover, in this deployment, a materialized view maintenance strategy needs to be considered. Again, materialized view maintenance adds another level of complexity since maintaining materialized views as new data arrives usually impacts the system performance. Although this deployment provides better query performance, it sacrifices backend administration simplicity in



	A	B	C	D
Low-latency queries	✗	✓	✓✓	✓✓
Ease of frontend deployment	✓	✓	✗	✓
Ease of backend administration	✓	✗	✓	✓
Disconnected operation	✗	✗	✓	✓

Figure 5.2: Trade-offs between various deployment scenarios to support interactive analytics in the cloud.

terms of the number of policies and technical issues that need to be addressed.

Deployment C. Another improvement to Deployment A is to move the materialized view to the client, i.e., stored in a local RDBMS. This further accelerates query performance since once the materialized view is copied over to the client, all subsequent queries run locally. Compared to Deployment B, this approach eliminates the backend administration issues described earlier because each data scientist can manage their own client machine in terms of allocating storage, cleaning up old results, etc.

This approach, however, complicates frontend deployment. We mention a few example issues that may arise. Firstly, data scientists are required to keep up to date with the latest version of the RDBMS. For instance, for a team of data scientists to be able to share their work (e.g., a notebook) they must ensure the compatibility of their local RDBMS versions. Secondly, it may be the case that some members of the data science teams are not able to install a local RDBMS on their client. Thirdly, usually different data scientists might have different client setups, for instance, different operating systems and other idiosyncratic configurations across many machines in an enterprise setting. In fact, the headache of managing these issues is what drove organizations to the cloud, i.e., Deployment C is a step backwards in terms of frontend simplicity.

Deployment D. The central value proposition of this chapter is that we propose new deployment architecture that retains the frontend and backend simplicity of Deployment A but offers performance optimizations comparable to B and C. We propose Deployment D, which is an improvement over Deployment C since it uses Afterburner that runs entirely in the web browser. Using Afterburner, the local database deployment becomes as simple as loading a webpage. As we have shown, Afterburner offers comparable latency to a local database, i.e., rivals Deployment C in performance. In this chapter, we experimentally show how Afterburner compares in end-to-end query latency to Deployment C.

In addition to the previously mentioned advantages, Afterburner offers the following benefits. First, Afterburner allows disconnected operation when access to the cloud is unavailable—a data scientist might, for instance, download a number of materialized views and then board an airplane, disconnect from the network, and continue working. Deployment C offers this flexibility also, but not A or B. Second, Afterburner supports multi-device deployment, e.g., on tablets and even mobile phones—since it is just JavaScript. The size of the data that can be manipulated, of course, is limited by the physical device, but the user experience is exactly the same. As we have demonstrated [18], Afterburner works on iPads, Android tablets, and both iOS and Android mobile phones. This feature is not supported by Deployment C, since a local database may not be available on all devices.

5.3 Split-Execution

5.3.1 Example:

As discussed, our work tackles the common scenario in interactive SQL data exploration where a data scientist executes a sequence of queries that differ only in the predicates in the WHERE clause. As an example of this scenario, in this section we use Q6 from the TPC-H benchmark, which examines "what-if" revenue missed because of discounts, to illustrate this idea. The data scientist might be interested in exploring revenue missed under different date ranges (i.e., different predicates on the `l_shipdate` column). Instead of issuing a different SQL query each time, she can use the following materialized view to answer this query for all date ranges:

```
CREATE MATERIALIZED VIEW Q6MVsql AS (
SELECT SUM(l_extendedprice * l_discount) AS f1,
       l_shipdate
FROM  lineitem
WHERE l_discount BETWEEN 0.05 and 0.07
  AND l_quantity < 24
GROUP BY l_shipdate );
```

The materialized view has two columns, a precomputed sum (`f1`) grouped by `l_shipdate` and the associated `l_shipdate`. Q6 can be computed using this view by applying the filter predicate to the `l_shipdate` column and computing a sum over the precomputed sums (`f1`), expressed as follows:

```
SELECT SUM(f1) AS revenue
FROM   Q6MVsql
WHERE  l_shipdate >= DATE '1994-01-01'
   AND l_shipdate < DATE '1995-01-01';
```

Afterburner is able to automatically and transparently rewrite a SQL query into two separate queries based on a hint provided by the data scientist. We introduce a FREE clause that allows the data scientist to specify the column which she wishes to issue follow-up SQL queries with different predicates over that column (`l_shipdate` in this case). Afterburner then generates the two queries above, which we refer to as:

- The **materialized view query** or $\texttt{Q}_{MV}$, which generates the appropriate materialized view.
- The **view query** or $\texttt{Q}_{V}$, which is the new query rewritten against the materialized view.

The data scientist can express Q6 using SQL, and provide a hint using the FREE clause over `l_shipdate` as follows:

```
SELECT SUM(l_extendedprice * l_discount)
       AS revenue
FROM   lineitem
WHERE  l_discount BETWEEN 0.05 and 0.07
   AND l_quantity < 24
FREE l_shipdate;
```

In Afterburner, this can be expressed in fluent SQL using the `free` operator over `l_shipdate` as follows:

```
Q6MVjs=abdb.select()
     .from('lineitem')
     .field(as(sum(mul('l_extendedprice','l_discount')),
              'revenue'))
     .where(between('l_discount',0.05,0.07))
     .where(lt('l_quantity', 24))
     .free('l_shipdate');
```

Calling `free` in the last line runs the materialized view query at the backend and copies it to the frontend. After this, she can interactively explore the data by adding more filters on the `l_shipdate` column (let's call it `Q6js`), which can be executed in the frontend by calling the `exec` method.

```
Q6js.exec(Q6MVjs);
```

The parameter `Q6MVjs` is passed to the `exec` method as the materialized view to use.

This scenario, using Afterburner's API, illustrates how Afterburner creates a materialized view transparently, i.e., the data scientist did not have to define the SQL query definition for the materialized view. In addition, she did not have to rewrite her query against the materialized view. This is desirable because she can work with multiple materialized views at the same time, working from the same query template with minimal modifications. We emphasize that these materialized views are *local* with respect to the data scientist and not updated as the original data sources change since, in our usage scenario, the views are intended to be transient.

5.3.2 Materialized View Query

Figure 5.3 illustrates the intuition of how Afterburner splits the execution of a query. On the left, we show a simple plan for Q6, starting with a `scan` operator that produces records based on the projected columns from the `lineitem` table. This is followed by a `filter` operator, which applies the predicates to the records, then a `group by` operator that applies the requested aggregation function. Finally, the `sink` operator produces the output. Although we can materialize the output of any operator, save it, and then ship it to the browser, then apply the rest of the operators, this may not be sufficient to "free" up the column.

To free the column, we transform the query in a way that delays the `l_shipdate` filter (shown on the right side of Figure 5.3). This transformation requires passing the `l_shipdate` column to the `group by` operator. Next, we materialize the output and ship it to the browser and then apply the rest of the plan. When the user changes the predicates on the `l_shipdate` column, only the local part of the plan must be changed—and can be executed in the browser using Afterburner.

Now that we have illustrated the intuition behind the materialized view generation, we discuss how Afterburner generates the SQL definition of the materialized view. Targeting SQL to create materialized views (instead of a physical plan, for example) comes with some benefits, which includes widening the applicability of our techniques to any backend that supports SQL (e.g., Spark). In addition, using SQL allows the backend to adopt the best plan to answer the query according to its own query optimizer.

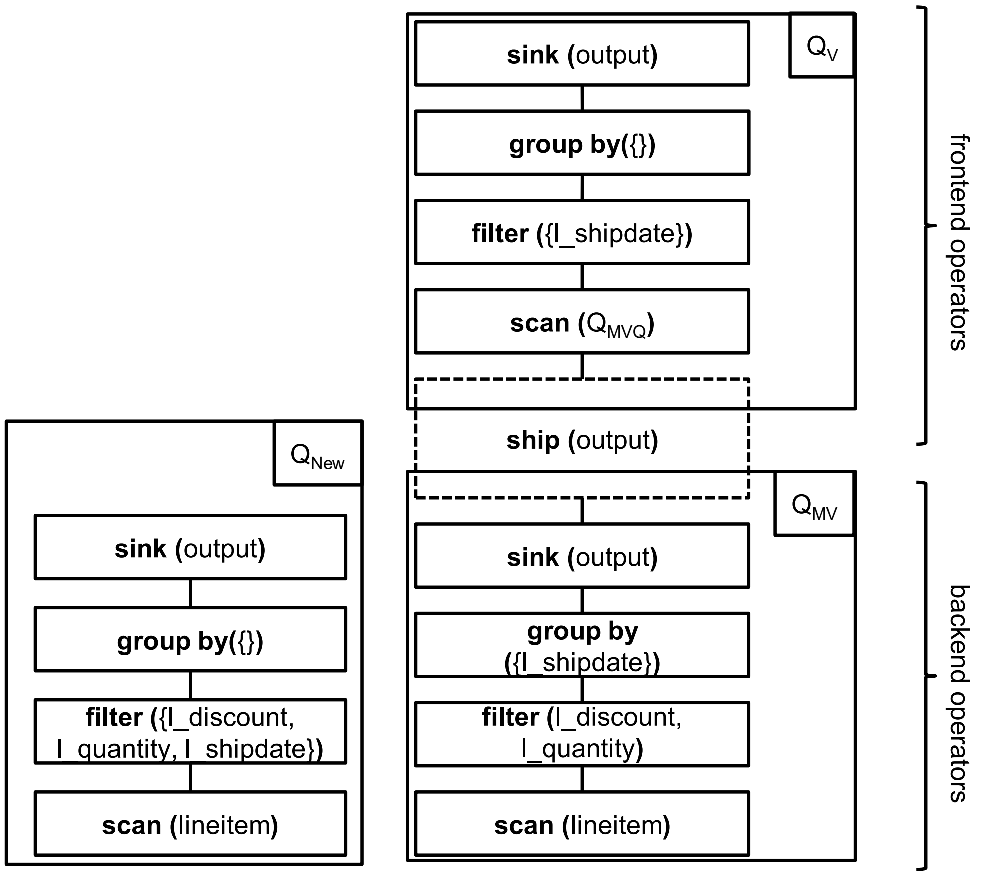


Figure 5.3: Original (left) and split-execution (right) plans.

Materialized view query generation is shown in Algorithm 3, which finds the materialized view definition (Q_{MV}) based on an original query (Q_o) and a valid column to free (`cfree`). Symbols used in this chapter are summarized in Table 5.1. Line 1 initializes an empty query template (Q_{MV}). Line 2 adds each term in the SELECT list of Q_o to Q_{MV}. For example, in Q6 the select clause of the original query contains the aggregated value SUM(`l_extendedprice * l_discount`) which is added to the materialized view query. In addition, line 2 adds the column in the FREE clause to the SELECT list of Q_{MV}, which is `l_shipdate`. This step is necessary to be able to apply predicates on `cfree` at the frontend. Lines 3–4 add the terms in the FROM and GROUP-BY lists of Q_o to Q_{MV}. The terms in the FROM clause can be any valid term, i.e., a table name or a subquery.

In principle, the materialized view Q_{MV} may inherit filter predicates on the `cfree` column from the original query Q_o, which limits subsequent queries. For example, the original query can

Symbol	**Explanation**
$\mathtt{Q}_o$	Original query with a FREE clause.
`cfree`	A column set to be free in the FREE clause of $\mathtt{Q}_o$.
$\mathtt{Q}_{MV}$	Materialized view query which is an SQL definition of the materialized view.
$\mathtt{Q}_n$	A new query submitted with a corresponding $\mathtt{Q}_{MV}$ to use.
$\mathtt{Q}_V$	A new query generated by Afterburner to run in the frontend.
Q.SELECT	The list of terms in the SELECT clause of query Q. A term in the SELECT clause can be, a constant value, an attribute name, or an aggregation function over an attribute name.
Q.FROM	The list of terms in the FROM clause of query Q. A term in the FROM clause can be a table name or a subquery.
Q.WHERE	The list of predicates in the WHERE clause of query Q.
Q.GROUP-BY	The list of columns in the GROUP-BY clause of query Q.

Table 5.1: Explanation of symbols used in this chapter.

specify a certain date range, which will limit subsequent queries on `l_shipdate` to be within that date range. However, in our implementation, we employ a simpler approach and do not consider filters on `cfree`, which simplifies the FREE clause. Thus, line 5 removes any instance of `cfree` from the WHERE list.

Lines 6–8 check for aggregates in the SELECT list of $\mathtt{Q}_{MV}$ and rewrite the aggregation functions based on predefined rules using the `rewriteAgg` method. For example, we rewrite AVG in terms of SUM and COUNT in order to be able to derive the AVG using the materialized view. Our system supports the standard set of aggregates: COUNT, SUM, AVG, MIN, and MAX. We describe our implementation of the `rewriteAgg` method in more details in section 5.3.4. If an aggregate exists, this requires adding the `cfree` column to the GROUP-BY list of $\mathtt{Q}_{MV}$, which is done in lines 9–10. Notice that we do not include the terms in the ORDER-BY and the LIMIT clauses during the materialized view creation.

For some queries and columns, delaying filtering might lead to materialized view definitions that do not suit our split-execution scenario due to physical limitations on the backend or the frontend. The materialized view query can exhaust the resources of the backend, for example, when the size of the intermediate output exhausts the available storage on the backend. The size of the final materialized view may also not fit into memory available at the frontend. These two considerations must be addressed when deciding which columns to free.

Our current implementation depends on a set of rules for deciding which columns can be freed. We disallow freeing columns that interact with other columns because they are likely to exhaust physical resources. Namely, a column cannot be in a condition on multiple columns such

Algorithm 3 Generate MVQ

```
Input: Query $Q_o$, Column cfree
 1: $Q_{MV} \leftarrow$ new Query()
 2: $Q_{MV}$.SELECT $\leftarrow Q_o$.SELECT $\cup$ {cfree}
 3: $Q_{MV}$.FROM $\leftarrow Q_o$.FROM
 4: $Q_{MV}$.GROUP $\leftarrow Q_o$.GROUP
 5: $Q_{MV}$.WHERE $\leftarrow Q_o$.WHERE $\setminus$ {cfree}
 6: for c $\in Q_{MV}$.SELECT $\wedge$ isAggregate(c):
 7:     c $\leftarrow$ rewriteAgg(c)
 8:     hasAggregate $\leftarrow$ true
 9: if hasAggregate:
10:     $Q_{MV}$.GROUP $\leftarrow Q_{MV}$.GROUP $\cup$ {cfree}
11: return $Q_{MV}$
```

as a join condition or a complex predicate involving other columns. We also disallow freeing a column mentioned in an OR clause. In addition, our system allows for subqueries but disallows freeing a column referenced in the subquery.

5.3.3 View Query

At query time, Afterburner must validate whether a materialized view can be used to answer a query. This is accomplished using rules that work on a parse-tree representation of an SQL query, which can be evaluated efficiently (negligible time in all of our evaluations). Since we expect only a small number of materialized views at the frontend, and that the data scientist is issuing queries in rapid succession, we believe that a simple rule-based approach is sufficient.

As demonstrated in our API, our task is query validation against a single materialized view. An important result from Larson and Zhou [40] is that a set of rules can be used to verify query coverage for the query class SPJOG (Select, Project, Join, and Outer join queries with a possible Group by). Thus, for a new query (Q_n), a query optimizer should be able to rewrite it against the local materialized view (Q_{MV}) transparently. In our prototype, we use simpler conditions to validate the query against a materialized view because our goal is to accelerate queries at the frontend—as opposed to general-purpose query optimization.

Our set of rules is simpler because of two main reasons. First, our prototype does not consider partial matching plans, which are plans that involve both the materialized view and base relations. While mixed plans can potentially improve overall query latency, we do not consider them in

Algorithm 4 Generate VQ

```
Input: Query $Q_n$, Query $Q_{MV}$
 1: $Q_V \leftarrow$ new Query()
 2: $Q_V$.SELECT $\leftarrow Q_n$.SELECT
 3: for c $\in Q_V$.SELECT $\wedge$ isAggregate(c):
 4:     c $\leftarrow$ deriveAgg(c)
 5: $Q_V$.FROM $\leftarrow$ {$Q_{MV}$.name}
 6: for p $\in Q_n$.WHERE $\wedge$ p $\in$ {cfree}:
 7:     $Q_V$.WHERE $\leftarrow Q_V$.WHERE $\cup$ {p}
 8: $Q_V$.GROUP $\leftarrow Q_n$.GROUP
 9: $Q_V$.ORDER $\leftarrow Q_n$.ORDER
10: $Q_V$.LIMIT $\leftarrow Q_n$.LIMIT
11: return $Q_V$
```

this setup since we do not assume access to the base tables at the client. Second, as we have mentioned in the previous section, we do not allow for predicates on the cfree column at the backend query. This simplifies our frontend plans to plans involving a single relation (which is Q_{MV}) with exactly matching predicates (other than predicates on cfree). Thus, the only differences allowed are in the GROUP-BY and the WHERE clause, and only involving the cfree column.

Materialized view matching conditions. When a new query is issued at the frontend with a corresponding materialized view, Afterburner uses the following conditions to validate whether Q_{MV} can be used to answer (matches) Q_n:

```
cond 1: $Q_n$.SELECT $\subseteq$ $Q_{MV}$.SELECT
cond 2: $Q_n$.FROM = $Q_{MV}$.FROM
cond 3: $Q_n$.JOIN = $Q_{MV}$.JOIN
cond 4: $Q_n$.WHERE \ {cfree} = $Q_{MV}$.WHERE
cond 5: $Q_n$.GROUP \ {cfree} $\subseteq$ $Q_{MV}$.GROUP
```

Condition 1 checks that the SELECT list of the Q_n is a subset of the Q_{MV}.SELECT, i.e., the new queries are not allowed to select new columns. Conditions 2 and 3 verify that the FROM and JOIN lists of Q_n matches exactly with Q_{MV}. Condition 4 checks that predicates in the WHERE list of Q_n match exactly the predicates in the Q_{MV} and may only add predicates on the cfree column. Condition 5 checks that all the GROUP-BY list of Q_n (if any) are either already included in the GROUP-BY list of Q_{MV} or is the cfree column.

View query generation. To generate the view query we use Algorithm 4, which rewrites the new query ($\texttt{Q}_n$) against the materialized view ($\texttt{Q}_{MV}$). Line 1 initializes the frontend query ($\texttt{Q}_V$) with an empty query. Line 2 adds all the terms in the SELECT list of $\texttt{Q}_n$ into $\texttt{Q}_V$. Lines 3–4 derive aggregate terms from the materialized view using the `deriveAgg` method which we describe in section 5.3.4. Line 5 adds the name of the materialized view ($\texttt{Q}_{MV}$) in the FROM list of $\texttt{Q}_V$. Lines 6–7 only add predicates on the `cfree` columns to the WHERE clause of $\texttt{Q}_V$. For example, $\texttt{Q}_V$ for Q6 only includes the following two predicates: `l_shipdate >= DATE '1994-01-01'` and `l_shipdate < DATE '1995-01-01'`. Finally, lines 8–10 add all the elements in the GROUP-BY, ORDER-BY, and LIMIT lists to $\texttt{Q}_V$.

For clarity, the above exposition describes how to free a single column, although our algorithm generalizes to freeing multiple columns in a straightforward manner. However, in practice, materialized views for freeing multiple columns are quite large since the materialized view query generates the Cartesian product of the columns—typically, this is more data than the client can handle.

5.3.4 Rewriting Aggregates

In this section we describe how Afterburner rewrites aggregate functions when generating the materialized view ($\texttt{Q}_{MV}$) and view query ($\texttt{Q}_V$). Algorithm 3 and Algorithm 4 use the methods `rewriteAgg` and `deriveAgg` to generate $\texttt{Q}_{MV}$ and $\texttt{Q}_V$, respectively. Our implementation of both methods is rather straightforward; we present the details regarding our implementation for completeness; see Srivastava et al. [52] for more details.

Table 5.2 summarizes the rules Afterburner uses to rewrite aggregate functions. The first column shows the aggregate functions that we support over a place holder term `T`. In order to support a wide range of queries, our system allows `T` to be either a column name or a UDF. UDFs allow users to express aggregates over numeric calculations, for example, in Q6 the UDF `SUM(l_extendedprice * l_discount)` is used to compute revenues. Alternatively, UDFs can be used to define aggregate functions that are based on conditions, for example, the following aggregate function from TPC-H Q12 counts the urgent orders:

```
SUM(CASE
        WHEN o_orderpriority = '1-URGENT'
          OR o_orderpriority = '2-HIGH'
        THEN 1
        ELSE 0
    END) AS high_line_count
```

Aggregate Function	**Rewritten in Q_{MV}**	**Rewritten in Q_V**
`COUNT(T) AS N`	`COUNT(T) AS c`	`SUM(c) AS N`
`SUM(T) AS N`	`SUM(T) AS s`	`SUM(s) AS N`
`AVG(T) AS N`	`COUNT(T) AS c, SUM(T) AS s`	`SUM(c)/SUM(s) AS N`
`MIN(T) AS N`	`MIN(T) AS m`	`MIN(m) AS N`
`MAX(T) AS N`	`MAX(T) AS m`	`MAX(m) AS N`

Table 5.2: Rules for rewriting aggregate functions.

The `rewriteAgg` method uses the first and second columns of Table 5.2 to rewrite the aggregate function. Thus, only rewrites the AVG function to COUNT and SUM that are needed later by `deriveAgg` when generating the Q_V. The implementation of `rewriteAgg` is simple, since the term `T` is pushed down to the backend verbatim. As a final step, `rewriteAgg` removes any redundant aggregate functions that were possibly added during the rewriting to minimize the space required to store Q_{MV}in the frontend. The `deriveAgg` method uses the first and third columns of Table 5.2 to rewrite the aggregate functions, for example, it rewrites `AVG(T)` in terms of `SUM(c)/SUM(s)`.

5.4 Evaluation

In this section we present an evaluation of interactive SQL analytics performance, taking advantage of Afterburner's split-execution. In this set of experiments, we compare D (Afterburner) against the different deployment alternatives A, B, and C, which we discussed in Section 5.2.2.

In order to create a query mix that captures typical interactive SQL analytics tasks, we once again draw inspiration from the TPC-H benchmark. For evaluation, we considered all the TPC-H queries with columns that can be "freed" based on our approach described in Section 5.3. For each TPC-H query, we generated variant queries as appropriate: for instance, for Q6, we can free the columns `l_shipdate`, `l_discount`, `l_quantity`, which yields Q6a, Q6b, and Q6c, respectively. In total, 12 queries from the TPC-H benchmark are amenable to this treatment, yielding 20 variant queries in total, since for each query, different numbers of columns can be freed.[2] These variant queries are shown in the first column of Table 5.3; the second column shows the column in the query that is freed (e.g., `l_shipdate`) and its cardinality. In principle, Afterburner can free more than one column at a time, which our implementation supports. In

[2]In some cases, the materialized views were too large to store at the client side; these queries were discarded in our evaluation.

practice, however, there may not be sufficient memory at the client to store the materialized views for multiple columns. Therefore, in our evaluation we only considered the case where one column is freed at a time.

We considered a data warehouse as defined by TPC-H at a scale factor of 100 GB, which yields a `lineitem` table with 600 million records. The backend in all scenarios is MonetDB running on a server with dual 8-core Intel Xeon E5-2670 processors (2.6 GHz) with 256 GB of memory on Ubuntu 14.04, configured to take advantage of all available hardware resources. The frontend machine in Deployments C and D is the same as in the desktop describe in the previous sections. In Deployment C, we run MonetDB on the local client machine, and in Deployment D, we run Afterburner in the browser. As a minor detail, in both Deployment B and Deployment C we ran MonetDB with only a single core because the sizes of the materialized views are sufficiently small that single core performance is actually better than multi-core performance—the overhead associated with multi-core query execution is more than the performance gained via parallelism. For Deployment C, latency is measured from the perspective of a browser, i.e., materializing the result set inside the browser. In all our experiments, the client and backend server are both located in the same building, with roundtrip ping times less than half a millisecond. All performance measurements reported below were on a warm cache and we report averages over five trials.

Results of our experiments are shown in Table 5.3. The column marked “A” shows the query latencies under Deployment A, which does not take advantage of materialized views. The “free” clause does not apply; the query latency is simply the query execution time of the original TPC-H query posed against the backend. For ease of comparison, we simply repeat the latency under the record for each query variant. Deployments B, C, and D all take advantage of materialized views: the latency of the query that generates this materialized view is reported under the column “MVQ”. The size of the materialized view (in number of records) is reported under the column “MV size”, and the time it takes to copy the materialized view from the backend to the client is reported in the column “MV copy” (applicable only for Deployments C and D). The latency of the view queries under Deployments B, C, and D are the columns marked “B”, “C”, and “D”.

Continuing with our running example of Q6 from Figure 5.1: from the results table we see that the unmodified query takes around 4.4 s. If we wish to free the `l_shipdate` column, which has a cardinality of 2526, the materialized view query takes around 9.8 s. However, all subsequent queries that involve only changes to predicates on `l_shipdate` only takes $\sim$10 ms to run against the materialized using any of the Deployments B, C, and D. Of course, in the case of Deployments C and D, the materialized view needs to be copied over to the client machine, which takes 247 ms.

The performance differences of the view queries for Q6 (comparing Deployments B, C, and D) are for the most part negligible, but performance aside, these deployments manifest all the

	FREE **column (cardinality)**	**A (ms)**	**MVQ (ms)**	**MV size (records)**	**MV copy (ms)**	**B (ms)**	**C (ms)**	**D (ms)**	**Breakeven (D vs. A)**
Q1a	l_shipdate (2526)	34,330	119,843	3,817	277	19	16	13	4
Q2a	p_size (50)	2,556	27,047	2,365,583	155,826	82	28	43	72
Q2b	p_type (150)	2,556	4,405	236,211	16,380	331	128	44	9
Q3a	c_mktsegment (5)	17,170	19,051	5,662,337	107,526	198	132	1,147	8
Q3b	o_orderdate (2406)	17,170	36,142	16,553,365	268,052	233	174	1,280	18
Q4a	o_orderdate (2526)	6,924	119,843	12,030	277	11	9	12	18
Q5a	r_name (5)	7,712	8,890	25	203	14	10	11	2
Q5b	o_orderdate (2406)	7,712	10,125	12,030	517	10	8	13	2
Q6a	l_shipdate (2526)	4,362	9,815	2,526	247	13	12	7	3
Q6b	l_discount (11)	4,362	4,590	11	375	12	9	6	2
Q6c	l_quantity (50)	4,362	8,100	50	420	11	10	6	2
Q7a	l_shipdate (2526)	8,388	8,390	5,052	649	11	10	11	2
Q12a	l_shipmode (7)	5,420	5,760	7	28	12	11	13	2
Q12b	l_receiptdate (2554)	5,420	7,219	4,985	163	10	13	12	2
Q14a	l_shipdate (2526)	4,052	29,298	2,526	364	9	10	7	8
Q17a	p_brand (25)	20,374	142,773	25	468	14	11	6	8
Q17b	p_container (40)	20,374	120,131	40	678	9	11	7	6
Q20a	n_name (25)	8,546	10,000	447,508	10,699	565	243	22	3
Q21a	o_orderstatus (3)	23,540	30,202	96,037	2,692	68	21	66	2
Q21b	n_name (25)	23,540	76,754	999,953	16,221	129	67	94	4

Table 5.3: Experimental results comparing Deployments A, B, C, and D on interactive SQL data exploration scenarios derived from the TPC-H benchmark. Each variant query describes a different "freed" column.

trade-offs discussed in Section 5.2.2. The performance of Deployment B is affected by backend query load (e.g., concurrent queries by multiple data scientists) as well as variability in network latencies. In our case the backend is less than half a millisecond away, but in an enterprise cloud deployment, latencies could be much longer. For instance, the roundtrip ping time between our client and an arbitrary instance on Amazon's EC2 service running in the US East region is around 25 ms. To be fair, though, link latencies will also affect the time it takes to copy the materialize view over to the client (in the cases of Deployments C and D), but the advantages of local interactions remain in eliminating all subsequent need for interactions with the backend.

Comparing Deployments C and D, with Afterburner (Deployment D), we have eliminated the need to have a local RDBMS installation—complete with all the headaches of maintaining a local software stack. With Afterburner, we achieve breakeven in three queries: if the data scientist issues three queries using the same query template as part of interactive data exploration, we make up for the fact that the initial materialized view query takes longer (even after accounting for the cost of transferring over the materialized view). This is shown in the final column in Table 5.3.

Looking at results across all queries in Table 5.3, we see that in most cases Afterburner (Deployment D) achieves performance parity with MonetDB in both Deployments B and C. Where there are performance differences, they are for a most part negligible. However, there are a few special cases to note: for Q3, Afterburner queries take substantially longer to execute in the browser. In this case, the materialized view is quite large and the query involves a top-k. MonetDB is able to optimize this into a scan, whereas Afterburner inefficiently sorts all records before taking the top k. The performance difference in this case is caused by deficiencies in query optimizations, not an inherent limitation of our approach.

Other results worth discussing are performance differences between Deployments B and C for some queries—since they are both running MonetDB and the differences are much larger than can be explained solely by hardware (the backend vs. the client). This is most evident for Q20a, where the view query for Deployment B takes more than twice as long as the view query for Deployment C. We attribute these differences to the cost of transferring the result set over: in this case, the result set contains 18k records. The latency in Deployment C includes the overhead of loading the results set into the browser.

In summary, it would be fair to characterize Deployment D as achieving performance parity with Deployment C. This is consistent with results from the previous section, where we examined end-to-end SQL analytics within the browser in comparison with MonetDB. Therefore, with Afterburner we can eliminate the need for client-side deployment of an RDBMS (with all its associated administrative and maintenance headaches) without compromising performance.

Chapter 6

Conclusion

We conclude this dissertation with a summary and a discussion about directions for future work.

6.1 Summary

In this dissertation, we identified and focused on two main challenges that face non-technical users who want to analyze data without enough computer science skills. The main challenges that we focused are: having to manage data analysis infrastructure is hard for non-technical users and the large number of options during data exploration tasks requires guidance. Chapter 3 introduced Afterburner, our in-browser SQL engine, which solves the first problem by making SQL as accessible as a webpage. Chapter 4 evaluated the efficiency and effectiveness of Explanation Tables which is an information-theoretic technique that guides data exploration tasks. In addition, we have introduced optimizations that allow for generating Explanation Tables inside the browser.

In Chapter 1, we have given an example of how a data scientist explored an open data set such as `Salary` using an RDBMS. She starts by identifying an attribute of interest, for instance, the binary attribute of having a high salary (`HighSalary`), which has a value of one if the public servant was granted high salary and set to zero otherwise. One common theme in analytics is to ask questions related to the data set using its dimension attributes. For instance, how does the dimension attribute values affect having a high salary, i.e., how does the ministry, position title, position class, etc., or combination of these attributes affect the binary outcome attribute? She can rely on exploring the data cube of the data set, i.e., she can load the data in an RDBMS and explore the result of different GROUP-BY combinations.

Recall, that this scenario poses two challenges for Bob (the user without a technical background from our example use case). The first challenge is that he does know how to install and manage an RDBMS. In Chapter 3, we have introduced Afterburner, our in-browser SQL engine, which solves this problem. Thus, he can load the data into a data cube exploration tool such as the one we outlined earlier. Combined with Afterburner, the tool can run entirely in the web browser, i.e., exploring the data becomes as simple as loading a web page. Coupled with an intuitive user interface, we have demonstrated that the user can navigate the data cube of the `Salary` data set without knowing SQL. To enable in-browser SQL analytics, we have outlined our proposed optimizations in order to enable SQL analytics using JavaScript which is known to be slower since the language was designed to priorities flexibility and security rather than performance. According to our knowledge, Afterburner is the first in-browser implementation of an SQL engine with performance rivaling native SQL RDBMS for modestly-sized data sets (e.g., millions of records) and complex queries (e.g., multiple joins). Afterburner's design depends on two main optimizations: using JavaScript typed arrays for columnar storage and code generation targeting asm.js. In addition, Afterburner depends on other optimizations that minimize its memory footprint, such as minimal materialization of intermediate query results and reusing typed arrays for query processing.

Afterburner alone does not solve the entire "data exploration for everyone" problem since users need guidance. This is exactly the second challenge we focus in this dissertation. To address this challenge, in Chapter 4, we have evaluated efficiency and effectiveness of Explanation Tables, an information-theoretic technique that picks the most informative patterns in the data cube for the user. The main intuition is that the most informative patterns are likely to be the most interesting starting points for data cube exploration. The problem of picking informative patterns in the data cube is challenging since patterns usually overlap; thus Explanation Tables applies the maximum entropy principle to patterns of the data cube. However, the size of the data cube is large and many of the existing data cube pruning techniques cannot be applied to the problem. Thus, we evaluated Flashlight, which prunes its candidate pattern space using a sample. A straightforward implementation of Flashlight is not efficient because it requires computing a complex join. Thus, we evaluated an optimized strategy for computing Flashlight, which generates Explanation Tables that are comparable to an exhaustive baseline in quality; nevertheless, requires a small fraction of the running time. In order to provide guidance for exploration tasks inside the web browser, we have adapted Flashlight to use Afterburner's in-memory storage to allow for creating explanation tables inside the browser. Our evaluations show the viability of guiding data exploration tasks inside the browser without any external dependencies.

Finally, in Chapter 5, we propose a split-execution technique that accelerates interactive SQL analytics for larger databases. This technique is based on computing a materialized view at the backend and copying it to the browser so that subsequent SQL data exploration queries can use the view and run fully locally. Our evaluations show that our proposal results in lower latency for SQL data exploration without adding much administrative burden at the frontend or backend.

Our work capitalizes on arguably the most widespread software environment, which is the web browser. We have shown experimentally that SQL analytics and information-based data guidance for modestly sized databases are as accessible as a webpage; thus, can be used by everyone. Our argument is that the impact of open data will significantly increase when the data is analyzed by more people. We also think that analytics democratization will benefit the society greatly. In the next section, we discuss directions for future work that we believe is next in importance towards analytics democratization.

6.2 Future Work

6.2.1 Analytics for Everyone

We see our work as a first step towards analytics democracy, and recognize the following to be arguably the best next natural extensions to our work:

- Firstly, relational data usually requires preparation steps such as identifying data types or correcting records that have obvious errors. We would like to study ways that would allow everyone to participate in such data preparation tasks. For example, facilitate ways were users can collaboratively annotate or clean open data. In order to ensure the transparency, integrity, and availability of the data we would like to build a distributed public ledger that does not depend on a single authority to store the data and govern its preparation.
- Secondly, today only very few organizations can store and analyze very large data sets, since it requires plenty of resources. We would like to study ways that would allow everyone to pool their client resources easily in order to analyze very large data sets. For example, a data journalist can plead his audience to donate their client resources for an analytics task that would benefit the society. Recent advancements in distributed computing allow people to serve society by donating CPU time. However, to our knowledge, nothing of that kind exists in data analytics. This is because of challenges related to moving and storing large amounts of data.

6.2.2 In-Browser Analytics

We would like to extend Afterburner in the following directions:

- First, we have demonstrated the feasibility of in-browser SQL physical operators using Afterburner. We would like to study the performance of different SQL physical operators, since it can uncover which operators lead to better query latencies under different conditions, such as database statistics, available resources, etc. This would allow for architecting better query planners that can pick the best operators for in-browser SQL processing.
- Second, Afterburner code generation targets asm.js. WebAssembly[1] is an emerging standard for high-performance in-browser applications. We would like to extend Afterburner such that it targets WebAssembly.
- Third, we would like to extend Afterburner to exploit parallelism.

6.2.3 Guiding Data Exploration Tasks

We would like to extend our studies of data exploration techniques in the following directions:

- First, Explanation Tables pick the most informative patterns from the data cube which are likely to be relevant to users. Our evaluations show using real data sets that Explanation Tables generate more informative summaries compared to other techniques. However, we would like to compare Explanation Tables with related techniques through user studies to assess its ability to guide users through data exploration tasks.
- Second, extend Explanation Table's information-theoretic framework so that it can pick patterns based on more than one outcome attribute. For example, highlight parts of the data cube with occasional correlations between two outcome attributes.
- Third, our evaluations to Explanation Table's information-theoretic framework are restricted to binary outcome attributes and categorical dimensional attributes. We would like to extend our evaluations to numeric outcome attributes and numeric dimensional attributes. For numeric attributes, we can evaluate the recent extensions to Explanation Tables [19].

6.2.4 Split-Execution

We would like to extend Afterburner's split-execution framework in the following directions:

[1] `http://webassembly.org/`

- First, split-execution in Afterburner depends on a user hint which is then translated into a materialized view. We would like to investigate ways where users do not have to provide hints, i.e., predict which materialized views based on the users' interactions.
- Second, split-execution in Afterburner depends on materialized views that compute exact results. We would like to look into incorporating approximate query answering techniques.
- Third, add a cost based model to Afterburner, which can estimate the breakeven factor to users before freeing an attribute.

References

[1] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. The Aqua approximate query answering system. In *SIGMOD*, pages 574–576, 1999.

[2] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. BlinkDB: Queries with bounded errors and bounded response times on very large data. In *EuroSys*, pages 29–42, 2013.

[3] Sameer Agarwal, Aurojit Panda, Barzan Mozafari, Anand P. Iyer, Samuel Madden, and Ion Stoica. Blink and it's done: Interactive queries on very large data. *PVLDB*, 5(12):1902–1905, 2012.

[4] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *VLDB*, pages 496–505, 2000.

[5] Stefaan Verhulst Ali Clare and Andrew Young. Enhanced transparency and accountability in development cooperation. 2016.

[6] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational data processing in Spark. In *SIGMOD*, pages 1383–1394, 2015.

[7] Peter Bailis, Edward Gan, Samuel Madden, Deepak Narayanan, Kexin Rong, and Sahaana Suri. MacroBase: Prioritizing attention in fast data. In *SIGMOD*, pages 541–556, 2017.

[8] World Bank. The world bank policy on access to information. 2010.

[9] Mangesh Bendre, Bofan Sun, Ding Zhang, Xinyan Zhou, Kevin Chen-Chuan Chang, and Aditya G. Parameswaran. DATASPREAD: unifying databases and spreadsheets. *PVLDB*, 8(12):2000–2003, 2015.

[10] Adam Berger, Vincent J. Della Pietra, and Stephen A. Della Pietra. A maximum entropy approach to natural language processing. *Computational Linguistics*, 22(1):39–71, 1996.

[11] Kevin Beyer and Raghu Ramakrishnan. Bottom-Up computation of sparse and Iceberg CUBE. *SIGMOD*, pages 359–370, 1999.

[12] Ivan T. Bowman and Kenneth Salem. Semantic prefetching of correlated query sequences. In *ICDE*, pages 1284–1288, 2007.

[13] Nicolas Bruno and Surajit Chaudhuri. Automatic physical database tuning: A relaxation-based approach. In *SIGMOD*, pages 227–238, 2005.

[14] Shaul Dar, Michael J. Franklin, Björn T. Jónsson, Divesh Srivastava, and Michael Tan. Semantic data caching and replacement. In *VLDB*, pages 330–341, 1996.

[15] Thomas G. Dietterich. An experimental comparison of three methods for constructing ensembles of Decision Trees: Bagging, boosting, and randomization. *Machine Learning*, 40(2):139–157, 2000.

[16] Kareem El Gebaly, Parag Agrawal, Lukasz Golab, Flip Korn, and Divesh Srivastava. Interpretable and informative explanations of outcomes. *PVLDB*, 8(1):61–72, 2014.

[17] Kareem El Gebaly, Lukasz Golab, and Jimmy Lin. Portable in-browser data cube exploration. *IDEA*, pages 35–39, 2017.

[18] Kareem El Gebaly and Jimmy Lin. In-browser interactive SQL analytics with Afterburner. In *SIGMOD*, 2017.

[19] Guoyao Feng, Lukasz Golab, and Divesh Srivastava. Scalable informative rule mining. In *ICDE*, pages 437–448, 2017.

[20] Michael J. Franklin, Björn Thór Jónsson, and Donald Kossmann. Performance tradeoffs for client-server query processing. In *SIGMOD*, pages 149–160, 1996.

[21] Alex Galakatos, Andrew Crotty, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. Revisiting reuse for approximate query processing. *PVLDB*, 10(10):1142–1153, 2017.

[22] Georgios Giannikis, Darko Makreshanski, Gustavo Alonso, and Donald Kossmann. Shared workload optimization. *PVLDB*, 7(6):429–440, 2014.

[23] Lukasz Golab, Howard Karloff, Flip Korn, Divesh Srivastava, and Bei Yu. On generating near-optimal tableaux for conditional functional dependencies. *PVLDB*, 1(1):376–390, 2008.

[24] Lukasz Golab, Flip Korn, and Divesh Srivastava. Efficient and effective analysis of data quality using pattern tableaux. *IEEE Data Eng. Bull.*, 34(3):26–33, 2011.

[25] Jonathan Goldstein and Per-A'oke Larson. Optimizing queries using materialized views: A practical, scalable solution. In *SIGMOD*, pages 331–342, 2001.

[26] Ashish Gupta, Venky Harinarayan, and Dallan Quass. Aggregate-Query processing in data warehousing environments. In *VLDB*, pages 358–369, 1995.

[27] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: An update. *SIGKDD Explorations*, 11(1):10–18, 2009.

[28] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. QPipe: A simultaneously pipelined relational query engine. In *SIGMOD*, pages 383–394, 2005.

[29] Carlos Iglesias and Kristen Robinson. Open data barometer - third edition. 2016.

[30] Lilong Jiang, Michael Mandel, and Arnab Nandi. GestureQuery: A multitouch database query interface. *PVLDB*, 6(12):1342–1345, 2013.

[31] Zhongjun Jin, Michael R. Anderson, Michael J. Cafarella, and H. V. Jagadish. Foofah: Transforming data by example. In *SIGMOD*, pages 683–698, 2017.

[32] Manas Joglekar, Hector Garcia-Molina, and Aditya G. Parameswaran. Interactive data exploration with Smart Drill-Down. In *ICDE*, pages 906–917, 2016.

[33] Minsuk Kahng, Shamkant B. Navathe, John T. Stasko, and Duen Horng (Polo) Chau. Interactive browsing and navigation in relational databases. *PVLDB*, 9(12):1017–1028, 2016.

[34] Manos Karpathiotakis, Ioannis Alagiannis, and Anastasia Ailamaki. Fast queries over heterogeneous data through engine customization. *PVLDB*, 9(12):972–983, 2016.

[35] Meraj Ahmed Khan, Larry Xu, Arnab Nandi, and Joseph M. Hellerstein. Data Tweening: Incremental visualization of data transforms. *PVLDB*, 10(6):661–672, 2017.

[36] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. Building efficient query engines in a high-level language. *PVLDB*, 7(10):853–864, 2014.

[37] Nick Koudas, Chen Li, Anthony K. H. Tung, and Rares Vernica. Relaxing join and selection queries. In *VLDB*, pages 199–210, 2006.

[38] Konstantinos Krikellas, Stratis Viglas, and Marcelo Cintra. Generating code for holistic query evaluation. In *ICDE*, pages 613–624, 2010.

[39] Laks V. S. Lakshmanan, Raymond T. Ng, Christine Xing Wang, Xiaodong Zhou, and Theodore Johnson. The generalized MDL approach for summarization. In *VLDB*, pages 766–777, 2002.

[40] Per-Åke Larson and Jingren Zhou. View matching for outer-join views. In *VLDB*, pages 445–456, 2005.

[41] Daniel Lathrop and Laurel Ruma. *Open Government: Collaboration, Transparency, and Participation in Practice*. O'Reilly Media, Inc., 1st edition, 2010.

[42] Fei Li and H. V. Jagadish. Understanding natural language queries over relational databases. *SIGMOD Record*, 45(1):6–13, 2016.

[43] Jimmy Lin. Building a self-contained search engine in the browser. In *ICTIR*, pages 309–312, 2015.

[44] Arnab Nandi, Cong Yu, Philip Bohannon, and Raghu Ramakrishnan. Data cube materialization and mining over MapReduce. *TKDE*, 24(10):1747–1759, 2012.

[45] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[46] Yongjoo Park, Ahmad Shahab Tajik, Michael J. Cafarella, and Barzan Mozafari. Database Learning: Toward a database that becomes smarter every time. In *SIGMOD*, pages 587–602, 2017.

[47] Steven Ruggles, J. Trent Alexander, Katie Genadek, Ronald Goeken, Matthew Schroeder, and Matthew Sobek. Integrated public use microdata series: Version 5.0, 2010.

[48] Sunita Sarawagi. Explaining differences in multidimensional aggregates. In *VLDB*, pages 42–53, 1999.

[49] Sunita Sarawagi. User-Adaptive exploration of multidimensional data. In *VLDB*, pages 307–316, 2000.

[50] Sunita Sarawagi. User-Cognizant multidimensional analysis. *VLDB Journal*, 10(2-3):224–239, 2001.

[51] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. How to architect a query compiler. In *SIGMOD*, pages 1907–1922, 2016.

[52] Divesh Srivastava, Shaul Dar, H. V. Jagadish, and Alon Y. Levy. Answering queries with aggregation using views. In *VLDB*, pages 318–329, 1996.

[53] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy M. Lohman, and Alan Skelley. DB2 Advisor: An optimizer smart enough to recommend its own indexes. In *ICDE*, pages 101–110, 2000.

[54] Stefaan Verhulst and Andrew Young. Key findings of the open data impact case studies. 2016.

[55] Dong Xin, Jiawei Han, Xiaolei Li, and Benjamin W. Wah. Star-Cubing: Computing Iceberg cubes by top-down and bottom-up integration. In *VLDB*, pages 476–487, 2003.

[56] Markos Zaharioudakis, Roberta Cochrane, George Lapis, Hamid Pirahesh, and Monica Urata. Answering complex SQL queries using automatic summary tables. In *VLDB*, pages 105–116, 2000.