The VLDB Journal
https://doi.org/10.1007/s00778-020-00633-6

SPECIAL ISSUE PAPER

DIFF: a relational interface for large-scale data explanation

Firas Abuzaid[1] · Peter Kraft[1] · Sahaana Suri[1] · Edward Gan[1] · Eric Xu[1] · Atul Shenoy[2] · Asvin Ananthanarayan[2] · John Sheu[2] · Erik Meijer[3] · Xi Wu[4] · Jeff Naughton[4] · Peter Bailis[1] · Matei Zaharia[1]

Received: 2 February 2020 / Revised: 16 August 2020 / Accepted: 26 August 2020

Abstract

A range of *explanation engines* assist data analysts by performing feature selection over increasingly high-volume and high-dimensional data, grouping and highlighting commonalities among data points. While useful in diverse tasks such as user behavior analytics, operational event processing, and root-cause analysis, today's explanation engines are designed as stand-alone data processing tools that do not interoperate with traditional, SQL-based analytics workflows; this limits the applicability and extensibility of these engines. In response, we propose the `DIFF` operator, a relational aggregation operator that unifies the core functionality of these engines with declarative relational query processing. We implement both single-node and distributed versions of the `DIFF` operator in MB SQL, an extension of MacroBase, and demonstrate how `DIFF` can provide the same semantics as existing explanation engines while capturing a broad set of production use cases in industry, including at Microsoft and Facebook. Additionally, we illustrate how this declarative approach to data explanation enables new logical and physical query optimizations. We evaluate these optimizations on several real-world production applications and find that `DIFF` in MB SQL can outperform state-of-the-art engines by up to an order of magnitude.

Keywords Data exploration · Explanations · Big data · Data analytics · Databases · Feature selection · Query optimization

1 Introduction

Given the continued rise of high-volume, high-dimensional data sources [9], a range of *explanation engines* (e.g., MacroBase, Scorpion, and Data X-Ray [8,48,56,64,66]) have been proposed to assist data analysts in performing feature selection [31], grouping and highlighting commonalities among data points. For example, a product manager responsible for the adoption of a new mobile application may wish to determine why user engagement declined in the past week. To do so, she must inspect thousands of factors, from the application version to user demographic, device

✉ Firas Abuzaid
fabuzaid@cs.stanford.edu

Peter Kraft
kraftp@stanford.edu

Sahaana Suri
sahaana@stanford.edu

Edward Gan
egan1@stanford.edu

Eric Xu
ericxu0@stanford.edu

Atul Shenoy
atul.shenoy@microsoft.com

Asvin Ananthanarayan
asvina@microsoft.com

John Sheu
jsheu@microsoft.com

Erik Meijer
erikm@fb.com

Xi Wu
wuxi@google.com

Jeff Naughton
naughton@google.com

Peter Bailis
pbailis@cs.stanford.edu

Matei Zaharia
matei@cs.stanford.edu

[1] Stanford DAWN Project, Stanford University, Stanford, CA, USA

[2] Microsoft Inc, Redmond, WA, USA

[3] Facebook Inc, Menlo Park, CA, USA

[4] Google Inc, Mountain View, CA, USA

Published online: 30 September 2020

type, and location metadata, as well as combinations of these features. With conventional business intelligence tools, the product manager must manually perform a tedious set of GROUP BY, UNION, and CUBE queries to identify commonalities across groups of data records corresponding to the declined engagement metrics. Explanation engines automate this process by identifying statistically significant combinations of attributes, or *explanations*, relevant to a particular metric (e.g., records containing `device_make="Apple", os_version="9.0.1", app_version="v50"` are two times more likely to report lower engagement). As a result, explanation engines enable order-of-magnitude efficiency gains in diagnostic and exploration tasks.

Despite this promise, in our experience developing and deploying the MacroBase explanation engine [8] at scale across multiple teams at Microsoft and Facebook, we have encountered two challenges that limit the applicability of explanation engines: interoperability and scalability.

First, analysts often want to search for explanations as part of a larger workflow: An explanation query is typically a subcomponent of a larger pipeline combining extract–transform–load (ETL) processing, OLAP queries, and GUI-based visualization. However, existing explanation engines are designed as stand-alone tools and do not interoperate with other relational tools or workflows. As a result, interactive explanation-based analyses require substantial pre- and post-processing of results. For example, in data warehouses with a snowflake or star schema, analysts must combine fact tables with dimension tables using complex projections, aggregations, and JOINs prior to use in explanation analyses [39]. To construct downstream queries based on the results of an explanation, analysts must manually parse and transform the results to be compatible with additional relational operators.

Second, analysts often require explanation engines that can scale to growing data volumes, while still remaining interactive. For example, a typical explanation analysis might require processing weeks of raw event logs to identify a subtle issue arising from a small subpopulation of users. Since these analyses are usually performed with a human in the loop, a low-latency query response is highly advantageous. In our experience deploying MacroBase at Microsoft and Facebook, we found that existing approaches for data explanation did not scale gracefully to the dozens of high-cardinality columns and hundreds of millions of raw events we encountered. We observed that even a small explanation query over a day's worth of Microsoft's production telemetry data required upward of 10 min to complete.

In response to these two challenges, we introduce the DIFF operator, a declarative relational operator that unifies the core functionality of several explanation engines with traditional relational analytics queries. Furthermore, we show that the DIFF operator can be implemented in a scalable manner.

To address the first challenge, we exploit the observation that many explanation engines and feature selection routines summarize differences between populations with respect to various *difference metrics* or functions designed to quantify particular differences between disparate subgroups in the data (e.g., the prevalence of a variable between two populations). We capture the semantics of these engines via our DIFF operator, which is parameterized by these difference metrics and can generalize to application domains such as user behavior analytics, operational event processing, and root cause analysis. DIFF is semantically equivalent to a parameterized relational query composed of UNION, GROUP BY, and CUBE operators and therefore integrates with current analytics pipelines that utilize a relational data representation.

However, incorporating the DIFF operator into a relational query engine raises two key scalability questions:

1. What *logical layer optimizations* are needed to efficiently execute SQL queries when combining DIFF with other relational algebra operators, especially JOINs?
2. What *physical layer optimizations*—algorithms, indexes, and storage formats—are needed to evaluate DIFF efficiently?

At the logical layer, we present several new optimizations for DIFF. The first optimization is informed by the snowflake and star schemas common in data warehouses, where data are augmented with metadata via JOINs before running explanation queries [39]. A naïve execution of this workflow would fully materialize the JOINs and then evaluate DIFF on their output. We show that if the returned output size after computing the JOINs far exceeds that of the DIFF, it is more efficient to perform the DIFF operation before materializing the JOINs, thereby applying a predicate pushdown-style strategy to DIFF–JOIN queries (similar to learning over JOINs [41]). We introduce an adaptive algorithm that dynamically determines the order of operations, yielding up to 2× speedups on real data.

In our second logical optimization, we show how to leverage functional dependencies present in the input data to prune semantically redundant explanations. For instance, joining with a geographic dimension table may provide state and country information—but for explanation queries, returning both fields is superfluous, as an explanation containing `state = "CA"` is equivalent to one containing `state = "CA", country = "USA"`. We show how pruning candidate explanations using these functional dependencies can yield up to 20% speedups. We also discuss how to extend this idea to support "soft" functional dependencies—unlike "hard" FDs that are guaranteed to hold throughout the entire dataset, soft FDs only hold with high probability.

Our third logical optimization exploits the statistical nature of the *mean shift*, a popular difference metric seen in

many industry use cases for DIFF. We show that, by deriving inequality bounds on and then computing the *variance* of various candidate explanations during execution, we can apply a pruning rule that discards many candidates and accelerates query runtimes. Our experiments show that, for datasets with attributes exhibiting sufficiently low variance, our variance pruning optimization can improve query performance by as much as 5×.

Our fourth proposed logical optimization is informed by a common usage pattern of DIFF observed in industry; we find that DIFF query workloads are often highly repetitive, with successive queries sharing significant overlap in terms of their input parameters. Therefore, we propose a multi-query optimization for DIFF, which takes in a set of a priori DIFF queries over the same input relations but with different parameters and evaluates them as a single super-query, thus reusing work across multiple queries.

At the physical layer, we implement DIFF based on a generalized version of the Apriori algorithm from Frequent Itemset Mining [2]. However, we develop several complementary optimizations, including hardware-efficient encoding of explanations in packed integers, storing columns in a columnar format, and judiciously representing specific columns using bitmaps. By exploiting properties in the input data, such as low-cardinality columns, and developing a cost-based optimizer for selecting an efficient physical representation, our optimized implementation of DIFF delivers speedups of up to 17× compared to alternatives.

To illustrate the performance improvements of these logical and physical optimizations, we implement the DIFF operator in MB SQL, an extension to MacroBase. We develop both a single-node implementation and a distributed implementation in Spark [69], allowing us to scale to hundreds of millions of rows or more of production data. We benchmark our implementation of DIFF with queries derived from several real-world analyses, including workloads from Microsoft and Facebook and show that MB SQL can outperform other explanation query engines, such as MacroBase and RSExplain[56], by up to 10× despite their specialization. Additionally, MB SQL outperforms related dataset mining algorithms from the literature, including optimized Frequent Itemset Mining algorithms found in Spark MLlib [50], by up to 4.5×.

Finally, to better serve industry use cases, we introduce a companion operator, called ANTI DIFF, that returns the complement of the DIFF operator's output. We show that ANTI DIFF can also provide valuable insights in many industrial settings, and we demonstrate that it can take advantage of the same logical and physical optimizations previously mentioned for efficient execution. Furthermore, we present a pruning rule specific to ANTI DIFF, which achieves to a 2× improvement in runtime compared to a naïve implementation.

In summary, we present the following contributions:

- We propose the DIFF operator, a declarative relational aggregation operator that unifies the core functionality of explanation engines with relational query engines.
- We present novel logical optimizations to evaluate the DIFF operator in conjunction with JOINs and functional dependencies; we also present a multi-query optimization technique, as well as our variance pruning optimization. Each of these optimizations can accelerate DIFF queries, in some cases by as much as 5×.
- We introduce an optimized physical implementation of the DIFF operator that combines dictionary encoding, columnar storage, and data-dependent, cost-based bitmap indexes, yielding up to a 17× improvement in performance.
- We introduce a companion operator, called ANTI DIFF, that evaluates the complement of the DIFF operator. We show that ANTI DIFF can take advantage of the same optimizations proposed for DIFF; in addition, we propose a logical optimization specific to ANTI DIFF that yields a 2× speedup.

2 The DIFF operator

The DIFF operator is a relational aggregation operator that provides a declarative interface for explanation queries. In this section, we introduce the DIFF operator's API, sample usage, and semantics and detail how to replicate the behavior of the explanation engines in Sect. 2.3.

2.1 DIFF operator syntax and example workflow

We present syntax for the DIFF operator in Backus–Naur form in Fig. 1. The DIFF operator takes as input two relations— the *test relation* and the *control relation*. Similar to a CUBE query [29], DIFF outputs combinations of attribute–value pairs (e.g., `make="Apple", os="11.0"`), which we refer to as *explanations*, in the form of a single relation, where each row consists of an explanation describing how the test and control relations differ.

DIFF is parameterized by a MAX ORDER argument, which specifies the maximum number of attributes considered per

```
diff_query = table_ref "DIFF" table_ref
             "ON" {attrs}
             "COMPARE BY" {diff_metric([args]) > threshold}
             ["MAX ORDER" integer] ;

diff_metric = "support" | "odds_ratio" | "risk_ratio"
            | "mean_shift" | udf
```

Fig. 1 DIFF syntax in extended Backus–Naur form

Table 1 Generally applicable built-in difference metrics

Difference metric	Description
Support	Fraction of rows with an attribute
Odds ratio	Odds that a row will be in the test relation versus the control relation if it has an attribute versus if it does not
Risk ratio	Probability that a row will be in the test relation versus the control relation if it has an attribute versus if it does not
Mean shift	Percent change in a column's mean for rows containing an attribute in the test versus the control relation

explanation, and one or more difference metric expressions that define the utility of an explanation. These expressions consist of a difference metric that quantifies the difference between explanations and a corresponding threshold; the difference metric is a function that acts on each explanation to define its utility, and explanations that do not satisfy the utility threshold are pruned from the output.

As we demonstrate in Sect. 2.3, different difference metrics allow the `DIFF` operator to encapsulate the functionality of a variety of explanation engines. By default, the `DIFF` operator can make use of four provided difference metrics, which we describe in Table 1. While we found these difference metrics are sufficient for our industrial use cases, the `DIFF` operator supports user-defined difference metrics as well, as we discuss in Sect. 2.4.

Example workflow. To demonstrate how to construct and utilize `DIFF` queries, we consider the case of a mobile application developer who has been notified of increased application crash rates in the last few days. The developer has a relational database of log data from instances of both successful and failed sessions from her application:

timestamp	app_version	device_type	os	crash
08-21-18 00:01	v1	iPhone X	11.0	false
...	...	...	...	...
08-28-18 12:00	v2	Galaxy S9	8.0	true
...	...	...	...	...
09-04-18 23:59	v3	HTC One	8.0	false

With this input, the developer must identify potential explanations or causes for the crashes. To do so, she can make use of the `DIFF` operator by executing the following query:

```
SELECT * FROM
  (SELECT * FROM logs WHERE crash = true)
    crash_logs
DIFF
  (SELECT * FROM logs WHERE crash = false)
    success_logs
ON app_version, device_type, os
COMPARE BY risk_ratio >= 2.0, support >= 0.05
    MAX ORDER 2;
```

Here, the developer first selects her test relation to be the instances when a crash occurred in the logs (`crash_logs`) and the control relation to be instances when a crash did not occur (`success_logs`). In addition, she specifies the dimensions to consider for explanations of the crashes: `app_version, device_type, os`.

The developer must also specify how potential explanations should be ranked and filtered; she can accomplish this by specifying one or more difference metrics and thresholds. In this scenario, she first specifies the *risk ratio*, which quantifies how much more likely a data point matching this explanation is to be in the test relation than in the control relation. By specifying a threshold of 2.0 for the risk ratio, all returned explanations will be at least twice as likely to occur in `crash_logs` than in `success_logs`. Further, the developer only wants to consider explanations that have reasonable coverage (i.e., explain a substantial portion of the crashes). Therefore, she specifies a *support* threshold of 0.05, which guarantees that every returned explanation will occur at least 5% of the time in `crash_logs`. Finally, the developer includes the clause `MAX ORDER 2` to specify that the returned explanations should never contain more than two attributes. Running this `DIFF` query, the developer obtains the following results:

app_version	device_type	os	risk_ratio	support
v1	–	–	10.5	15%
–	iPhone XS	–	9.5	10%
–	–	10.0	8.25	12%
v2	iPhone X	–	7.25	30%
–	Pixel 4	10.2	10.0	35%
v2	–	10.1	7.75	25%

For each explanation, the output includes the explanation's attributes, risk ratio, and support. A `NULL` value (denoted as `"-"` in the output) indicates that the attribute can be any value, similar to the output of a `CUBE` query. Thus, the first explanation—`app_version="v1"`—is 10.5× more likely to be associated with a crash in the logs, and it occurs in 15% of the crashes.

The developer in our scenario may find these results uninteresting—they may be known bugs. So, as a follow-up, she changes the `MAX ORDER` clause to be `MAX ORDER 3` and reruns the query:

app_version	device_type	os	risk_ratio	support
v1	–	–	10.5	15%
–	iPhone XS	–	9.5	10%
–	–	10.0	8.25	12%
v2	iPhone X	–	7.25	30%
–	Pixel 4	10.2	10.0	35%
v2	–	10.1	7.75	25%
v3	Galaxy S9	11.0	9.75	20%

The new row `{app_version="v3", device_type="Galaxy S9", os="11.0"}` warrants further study; the developer has not encountered this explanation before. In response, she can issue a second `DIFF` query comparing this week's crashes to last week's:

```
SELECT * FROM
  (SELECT * FROM logs WHERE crash = true and
      timestamp BETWEEN 08-28-18 AND
      09-04-18) this_week
DIFF
  (SELECT * FROM logs WHERE crash = true and
      timestamp BETWEEN 08-21-18 AND
      08-28-18) last_week
ON app_version, device_type, os
COMPARE BY risk_ratio >= 2.0, support >= 0.05
     MAX ORDER 3;
```

which yields the following result:

app_version	device_type	os	risk_ratio	support
v3	Galaxy S9	11.0	20.0	75%

In the most recent week, our explanation from the previous query shows up 20× more often, and 75% of the crashes can be attributed to it. With this `DIFF` query, the developer has confirmed that there is likely a bug in her application causing Galaxy S9 devices running Android OS version 11.0 with app version `v3` to crash.

2.2 Formal definition of the DIFF operator

In this section, we define the `DIFF` operator and its two components: explanations and difference metrics.

Definition 1 Explanation We define an *explanation* of *order* k to be a set of k attribute values:

$$\mathcal{A}_* = \{A_1 = a_1, \ldots, A_k = a_k\} \tag{1}$$

We borrow this definition from prior work on explanation engines, including RSExplain [56], Scorpion [66], and MacroBase [8]. In practice, explanations typically consist of categorical attributes, although our definition can extend to continuous attributes as well, which we discuss in Sect. 2.5.

Definition 2 Difference Metric A difference metric filters candidate explanations based on some measure of severity, prevalence, or relevance; examples include support and risk ratio. We refer to a difference metric and its threshold as a *difference metric clause* γ (e.g., `support >= 0.05`). The output of a difference metric clause is a Boolean indicating whether the explanation $\mathcal{A}_*$ "passed" the difference metric. `DIFF` returns all attribute sets $\mathcal{A}_*$ from R and S that pass all specified difference metrics.

Formally, a difference metric clause γ takes as input two relations R and S and an explanation $\mathcal{A}_*$; it is parameterized by:

- A set $\mathcal{F}$ of d aggregation functions evaluated on R and S
- A comparison function h that takes the outputs of $\mathcal{F}$ on R and S to produce a single measure: $\mathbb{R}^d \times \mathbb{R}^d \times \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}$
- A user-defined minimum threshold, θ

A difference metric is computed by first evaluating the aggregation functions $\mathcal{F}$ over the relations R and S and the attribute set $\mathcal{A}_*$. We evaluate $\mathcal{F}$ first over the entire relation, $\mathcal{F}^R_{\text{global}} = \mathcal{F}(R)$, and then strictly over the rows matching the attributes in $\mathcal{A}_*$: $\mathcal{F}^R_{\text{attrs}} = \mathcal{F}(\sigma_{\mathcal{A}_*}(R))$. Similarly, we apply $\mathcal{F}$ on S, which gives us $\mathcal{F}^S_{\text{global}}$ and $\mathcal{F}^S_{\text{attrs}}$. Each evaluation of $\mathcal{F}$ returns a vector of values in $\mathbb{R}^d$, and $\mathcal{F}^R_{\text{global}}$, $\mathcal{F}^S_{\text{global}}$, $\mathcal{F}^R_{\text{attrs}}$, and $\mathcal{F}^S_{\text{attrs}}$ form the input to h. If h's output is greater than or equal to θ, then the attribute set $\mathcal{A}_*$ has satisfied the difference metric:

$$\gamma = h(\mathcal{F}^R_{\text{global}},\ \mathcal{F}^R_{\text{attrs}},\ \mathcal{F}^S_{\text{global}},\ \mathcal{F}^S_{\text{attrs}}) \geq \theta \tag{2}$$

Using this definition, we can express many possible difference metrics, including those listed in Table 1, as well as custom UDFs. For example, the support difference metric, which is defined over a single relation, would be expressed as:

$$\gamma_{\text{support}} := \begin{cases} \mathcal{F} = \texttt{COUNT(*)} \\ h = \dfrac{\mathcal{F}^R_{\text{attrs}}}{\mathcal{F}^R_{\text{global}}} \end{cases} \tag{3}$$

where θ_* denotes a user-specified minimum support threshold. The risk ratio, meanwhile, would be expressed as:

$$\gamma_{\text{risk\_ratio}} := \begin{cases} \mathcal{F} = \texttt{COUNT(*)} \\ h = \dfrac{\dfrac{\mathcal{F}^R_{\text{attrs}}}{\mathcal{F}^R_{\text{attrs}} + \mathcal{F}^S_{\text{attrs}}}}{\dfrac{\mathcal{F}^R_{\text{global}} - \mathcal{F}^R_{\text{attrs}}}{(\mathcal{F}^R_{\text{global}} - \mathcal{F}^R_{\text{attrs}}) + (\mathcal{F}^S_{\text{global}} - \mathcal{F}^S_{\text{attrs}})}} \end{cases} \tag{4}$$

Finally, the mean shift for a metric column m is the ratio between the mean of m in R for a given explanation $\mathcal{A}$ and the mean of m in S for that same explanation. Using the definition

of the difference metric, we can express it as follows:

$$\gamma_{\text{mean\_shift}} := \begin{cases} \mathcal{F} = \{f = \texttt{SUM(m)},\ g = \texttt{COUNT(*)}\} \\ h = \dfrac{\frac{f^R_{\text{attrs}}}{g^R_{\text{attrs}}}}{\frac{f^S_{\text{attrs}}}{g^S_{\text{attrs}}}} \end{cases} \tag{5}$$

Definition 3 DIFF We now define the DIFF operator Δ, which has the following inputs:

- R, the test relation
- S, the control relation
- Γ, the set of difference metrics
- $\mathcal{A} = \{A_1, \ldots, A_m\}$, the dimensions, which are categorical attributes common to both R and S
- k, the maximum order of dimension combinations

Algebraically, a DIFF query is expressed as follows:

$$\Delta_{\Gamma,\mathcal{A},k}(R, S),$$

or $\Delta(R, S)$ for the sake of brevity.

The DIFF operator applies the difference metrics Γ to every possible explanation with order k or less found in R and S; the explanations can only be derived from $\mathcal{A}$. The difference metrics are evaluated over every explanation $\mathcal{A}_*$—if the explanation satisfies all the difference metrics, it is included in the output of DIFF, along with its values for each of the difference metrics.

A DIFF query can be translated into standard SQL using multiple GROUP BY subqueries, as we illustrate in Appendix A. This translation step is costly, however, both for data analysts and for relational databases: as is shown in Appendix, the equivalent query is often hundreds of lines of SQL that database query planners fail to optimize and execute efficiently. In our experiments, we benchmark DIFF queries against equivalent SQL queries in Postgres and find that our implementation of DIFF is orders of magnitude faster.

With this new relational operator, we introduce two new benefits: (*i*) Users can concisely express their explanation queries *in situ* with existing analytic workflows, rather than rely on specialized explanation engines, and (*ii*) query engines—both on a single node or in the distributed case—can optimize DIFF *across* other relational operators, such as selections, projections, and JOINs. As we discuss in Sects. 4 and 5, integrating DIFF with existing databases requires implementing new logical optimizations at the query planning stage and new physical optimizations at the query execution stage. This integration effort can yield order-of-magnitude speedups, as we illustrate in our evaluation.

2.3 DIFF generality

The difference metric abstraction enables the DIFF operator to encapsulate the semantics of several explanation engines and Frequent Itemset Mining techniques via a single declarative interface. To highlight the generalization power of DIFF, we describe these engines/techniques and show how DIFF can implement them either partially or entirely; we implement several of these generalizations and report the results in our evaluation.

MacroBase MacroBase [8] is an explanation engine that explains important or unusual behavior in data. The MacroBase default pipeline computes risk ratio on explanations across an outlier set and inlier set and returns all explanations that pass a threshold. As the difference metric abstraction arose as a natural evolution of (and replacement for) the MacroBase default pipeline after our experience deploying MacroBase at scale, DIFF can directly express MacroBase functionality using a query similar to the example query in Sect. 2.1. We later on evaluate the performance of such a query compared to a semantically equivalent MacroBase query and find that our implementation of DIFF is over 6× faster.

Data X-Ray Data X-Ray[64] is an explanation engine that diagnoses systematic errors in large-scale datasets. From Definition 10 in [64], we can express Data X-Ray's Diagnosis Cost as a difference metric: Let $\epsilon = \frac{\mathcal{F}^R_{\text{attrs}}}{\mathcal{F}^R_{\text{attrs}} + \mathcal{F}^S_{\text{attrs}}}$, and let α denote the "fixed factor" that users can parameterize to tune a Data X-Ray query. The Diagnosis Cost can then be written as:

$$\gamma_{\text{diagnosis\_cost}} := \begin{cases} \mathcal{F} = \texttt{COUNT(*)} \\ h = \log \frac{1}{\alpha} + \mathcal{F}^R_{\text{attrs}} \log \frac{1}{\epsilon} + \mathcal{F}^S_{\text{attrs}} \log \frac{1}{1-\epsilon} \end{cases}$$

Once the Diagnosis Cost is computed for all attributes, Data X-Ray then tries to find the set of explanations with the least cumulative total cost that explains *all* errors in the data. The Data X-Ray authors show that this reduces to a weighted set cover problem, and they develop an approximate set-cover algorithm to determine what set of explanations to return. Thus, to capture Data X-Ray's full functionality, we evaluate a DIFF query to search for explanations and then post-process the results using a separate weighted set-cover solver. We implement such an engine and find that it obtains the same output and performance as Data X-Ray.

Scorpion Scorpion [66] is an explanation engine that finds explanations for user-identified outliers in a dataset. To rank explanations, Scorpion defines a notion of *influence* in Section 3.2 in [66], which measures, for an aggregate function

f, the delta between f applied to the entire input table R and f applied to all rows *not* covered by the explanation in R. Let g denote the aggregation function `COUNT(*)`, and let λ denote Scorpion's interpolation tuning parameter. Then, the influence can be expressed as the following difference metric:

$$\gamma_{\text{influence}} := \begin{cases} \mathcal{F} = \{f,\ g\} \\ h = \lambda \dfrac{\text{remove}(f^R_{\text{global}}, f^R_{\text{attrs}})}{g^R_{\text{attrs}}} \\ \qquad -(1-\lambda)\dfrac{\text{remove}(f^S_{\text{global}}, f^S_{\text{attrs}})}{g^S_{\text{attrs}}} \end{cases}$$

In this definition, remove refers to the notion of computing an *incrementally removable* aggregate, which the Scorpion authors define in Sect. 5.1 of their paper. An aggregate is incrementally removable if the updated result of removing a subset, s, from the inputs, R, can be computed by only reading s. For example, `SUM` is incrementally removable because `SUM(R - s) = SUM(R) - SUM(s)`. Here, we compute the influence for an explanation by removing the explanation's aggregate f^R_{attrs} from the total aggregate f^R_{global}; by symmetry, we do the same for the aggregates on S.

Unlike `DIFF`, Scorpion explanations can specify sets of values for a specific dimension column and can support more flexible `GROUP BY` aggregations. Nevertheless, the `DIFF` operator provides a powerful way of computing the key influence metric.

RSExplain RSExplain [56] is an explanation engine that provides a framework for finding explanations in database queries. RSExplain analyzes the effect explanations have on *numerical queries* or arithmetic expressions over aggregation queries (e.g., q_1/q_2, where q_1 and q_2 apply the same aggregation f over different input tables). RSExplain measures the *intervention* of an explanation, which is similar to the influence measure used in Scorpion. For a numerical query q_1/q_2 with aggregation f, the intervention difference metric would be written as:

$$\gamma_{\text{intervention}} := \begin{cases} \mathcal{F} = \{f\} \\ h = \dfrac{\text{remove}(f^R_{\text{global}}, f^R_{\text{attrs}})}{\text{remove}(f^S_{\text{global}}, f^S_{\text{attrs}})} \end{cases}$$

Frequent Itemset Mining A classic problem from data mining, Frequent Itemset Mining (FIM) [2], has a straightforward mapping to the `DIFF` operator: We simply construct a `DIFF` query with an empty control relation and whose sole difference metric is support. In our evaluation, we benchmark support-only `DIFF` queries against popular open-source frequent itemset miners, such as SPMF [27] on a single node, and Spark MLlib in the distributed setting . We find that `DIFF` is over 36× faster than SPMF's Apriori, 3.4× faster than SPMF's FPGrowth, and up to 4.5× faster than Spark MLlib's FPGrowth.

Multi-structural databases Multi-structural databases (MSDBs) are a data model that supports efficient analysis of large, complex datasets over multiple numerical and hierarchical dimensions [24,25]. MSDBs store data dimensions as a lattice of topics and define an operator called `DIFFERENTIATE` which returns a set of lattice nodes corresponding to higher-than-expected outlier data point occurrence rates.

`DIFFERENTIATE` approximates an NP-hard optimization problem that rewards small sets which explain a large fraction of the outlier data points. Because `DIFF` operates over relational tables and not MSDBs, we cannot precisely capture the semantics of `DIFFERENTIATE`. However, we can define a `DIFFERENTIATE`-like difference metric by comparing explanation frequency in the outlier set to a background rate and finding sets of relational attributes for which outliers occur significantly more often than they do in general.

2.4 Practical considerations for DIFF

While our definition of `DIFF` is purposefully broad, we observe that there are several practical considerations that `DIFF` query users undertake to maximize the analytical value of their queries. These practices are applicable to a broad range of applications and real-world datasets, especially for industrial workloads. Specifically, a typical `DIFF` query has the following properties:

1. The query always uses support as one of its difference metrics.
2. The maximum order of the query is $k \leq 3$.
3. The query outputs the most concise explanations—i.e., each output tuple should be the *minimal* set of attribute values that satisfy the difference metrics of the query.

The last property—which we refer to as *minimality*—is included so that the `DIFF` query returns valuable results to the user without overwhelming her with extraneous outputs. Obeying this property can change the semantics of `DIFF` and affect its generality (e.g., running `DIFF` with just support would no longer generalize to FIM, since FIM's goal is to find *maximal* itemsets [45]), and `DIFF` implementations can provide a flag to enable or disable it for a given query. If minimality is enabled, then there are two opportunities for optimization: We can (*i*) terminate early and avoid evaluating higher-order explanations when a lower-order subset already satisfies the difference metrics and (*ii*) incrementally prune the search space of candidate explanations as we compute

their difference metric scores, so long as the difference metric is *anti-monotonic*.

In general, a difference metric with threshold θ is anti-monotone if, whenever a set of attribute values $\mathcal{A}^*$ fails to exceed θ, so, too, will any superset of $\mathcal{A}^*$. The most common example of an anti-monotonic difference metric is support: The frequency of $\mathcal{A}^*$ in a table R will always be greater than or equal to any superset of $\mathcal{A}^*$.

When employing user-defined difference metrics, defining these metrics to also obey the anti-monotonicity property is highly desirable. While any function (or set of functions) can be supported as a difference metric (so long as they obey its definition), an anti-monotone difference metric enables the DIFF operator to prune candidate explanations and substantially improve query performance. If an explanation does not satisfy a user-defined anti-monotone difference metric, then all possible supersets of that explanation can also be discarded, thus reducing the search space of possible explanations. We discuss this optimization in more detail in Sect. 5.1.

2.5 Supporting continuous attributes

To support continuous attributes, the `DIFF` operator requires a means of discretizing each continuous attribute into distinct, non-overlapping ranges, effectively creating a set of categorical values to represent the continuous attribute. As with difference metrics, the `DIFF` operator itself is not wedded to any specific discretization strategy; instead, this choice is left up to the user.

This design decision is motivated by the observation that different continuous attributes will likely require different discretization strategies. For example, one may choose equi-width partitions for one attribute (as systems like DBSherlock do [68]) and equi-depth partitions for another (which are more expensive to compute). For attributes that exhibit an exponential distribution, one may choose to apply a log-uniform discretization. The user can also tune the number of discrete partitions for a given attribute, which can also affect the resulting explanations found.

This raises an interesting question: When evaluating explanation queries on datasets with continuous attributes, does there exist a data-driven method for determining the best possible discretization strategy for each continuous attribute? Further research is required to answer this question, and we leave this to the reader as future work.

3 ANTI DIFF: a complement to DIFF

While the `DIFF` operator is helpful for finding explanations, analysts may also be interested in finding data points that are *not* covered by explanations. More precisely, for a test relation R and control relation S, users may want to find tuples $r \in R$ whose attribute values $\mathcal{A}_*$ do not appear in $\Delta(R, S)$. To address this use case, we also introduce the `ANTI DIFF` operator, which mirrors the `DIFF` SQL syntax defined in Fig. 1. Intuitively, the `ANTI DIFF` operator is the inverse of `DIFF`; that is, an `ANTI DIFF` query will return all explanations that would not appear in the output of `DIFF`. More formally, the k-order `ANTI DIFF` of R and S will return the set of explanations with order k such that each explanation $\mathcal{A}_*$ does not satisfy at least one difference metric $\gamma \in \Gamma$.

Like `DIFF`, `ANTI DIFF` is also a relational operator, and it, too, can be translated to SQL. In fact, any `ANTI DIFF` query can be expressed as a `DIFF` query: The `ANTI DIFF` of test relation R and control relation S is equivalent to the `DIFF` of R and S subtracted from the `CUBE` of R over the dimensions $\mathcal{A}$.

This translation, however, is not an effective approach for the implementation of `ANTI DIFF`, as computing the `CUBE` can be costly for large datasets. In Sect. 4.2, we discuss how to employ logical optimizations to efficiently evaluate `ANTI DIFF` queries.

3.1 Example workflow and formal definition

While the `DIFF` operator can be used to find explanations for unusual trends in large datasets, the `ANTI DIFF` operator, by contrast, is helpful for finding features that correlate highly with expected or "normal" trends. If the test relation R is now treated instead as a "gold standard" set of features with corresponding metrics, then the `ANTI DIFF` query aims to answer the following question: Which other sets of features in the data are also highly correlated with that gold standard?

Such queries are useful in a variety of scenarios, such as time series analysis: Data analysts often wish to explain trends that *persist* between different time windows, not just those that differ. Another common use case for `ANTI DIFF` is to find distinct feature sets that surprisingly exhibit high similarity, which is especially valuable in cases when these feature sets prove elusive to find during manual analyses.

Example workflow. To demonstrate how to construct and utilize `ANTI DIFF` queries, an online content creator has published many videos worldwide. Her videos are measured by the impressions they collect, such as the time spent watching each video, and she keeps track of these analytics in the following table:

video_id	user_age	user_gender	user_country	watch_time
gu75ku9	30-40	male	Nigeria	0.23
...	...	...	...	...
gu75ku9	25-29	not shared	Jordan	0.45
...	...	...	...	...
x6g8f4y	18-24	female	Bolivia	1.0

(Note that the watch_time column indicates the *fraction* of time spent watching—a value of 1.0 indicates that the user watched the entire video.)

With these data, the content creator wants to gain a better understanding of her viewership's engagement. She recently released a new video (video_id = "x6g8f4y"), and she wants to track a core demographic, which represents her primary audience: young adult women in Latin and South America. She knows that, in her previous videos, engagement was high among this cohort, and she wants to confirm that this is still the case. Therefore, she runs the following ANTI DIFF query:

```
SELECT * FROM
  (SELECT * FROM view_data WHERE video = "
      x6g8f4y") new_video
ANTI DIFF
  (SELECT * FROM view_data WHERE video != "
      x6g8f4y") old_videos
ON user_age, user_gender, user_country
COMPARE BY mean_shift(watch_time) >= 1.5,
    support >= 0.1
MAX ORDER 3;
```

and receives the following output:

user_age	user_gender	user_country	mean_shift	support
18–24	–	-	1.05	40%
–	female	–	1.15	55%
–	–	Bolivia	1.10	5%
...	...	...	...	...
18–24	female	–	1.25	30%
...	...	...	...	...
18–24	female	Bolivia	2.05	3%
...	...	...	...	...

Because she uses the ANTI DIFF, she receives any explanation that did not satisfy at least one of the two difference metrics. Note the presence of the column {user_age="18-24", user_gender="female", user_country="Bolivia"}: Although its mean_shift is greater than 1.5, it is included in the output due to its support not meeting the 10% threshold specified in the query.

These results seem to confirm that her latest video was once again popular among her target audience. But, to double-check, she issues the same query but filters for all 3-order explanations (by adding a simple WHERE clause):

user_age	user_gender	user_country	mean_shift	support
18-24	female	Bolivia	2.05	3%
18-24	female	Costa Rica	1.05	8%
...	...	..	...	8%
24-29	female	Argentina	1.15	6%

This result confirms that her latest video is once again performing well across her target demographic. By using the ANTI DIFF, the online content creator can track various cohorts and examine their engagement levels with a single query.

Definition 4 ANTI DIFF

We now define the ANTI DIFF operator, which we denote as ∇. Like DIFF, the ANTI DIFF operator takes the same inputs: test and control relations R and S, a set of difference metrics Γ, a set of dimensions $\mathcal{A}$, and a maximum order k.
Algebraically, a ANTI DIFF query is expressed as follows:

$$\nabla_{\Gamma,\mathcal{A},k}(R, S),$$

or $\nabla(R, S)$ for the sake of brevity.

Like DIFF, the ANTI DIFF operator applies the difference metrics Γ to every possible explanation with order k or less found in R and S; the explanations can only be derived from $\mathcal{A}$. The difference metrics are evaluated over every explanation $\mathcal{A}_*$—if the explanation *does not satisfy any* of the difference metrics, it is included in the output of ANTI DIFF, along with its values for each of the difference metrics.

4 Logical optimizations for DIFF

For many workloads, data analysts need to combine relational fact tables with dimension tables, which are stored in a large data warehouse and organized using a snowflake or star schema. Because the DIFF operator is defined over relational data, we can design logical optimizations that take advantage of this setting and co-optimize across other expensive relational operators. These optimizations are not possible in existing explanation engines, which do not provide an algebraic abstraction for explanation finding. In this section, we discuss several *logical* optimizations:

1. a predicate-pushdown-based adaptive algorithm for evaluating DIFF in conjunction with JOINs;
2. an algorithm for ANTI DIFF that incrementally applies difference metrics and prunes the search space of possible explanations, thus obviating the need to evaluate the CUBE of R;
3. a technique that leverages both hard and soft functional dependencies to accelerate DIFF query evaluation when possible;
4. a statistical predicate pushdown optimization for queries combining the support and mean shift difference metrics, based on variance decompositions; and
5. a technique to apply multi-query optimization for consecutive DIFF queries applied to the same input relations.

Throughout this section—along with the subsequent section on physical optimizations—we focus on optimizations

for DIFF that make the assumptions in Sect. 2.4. Unless otherwise stated, each optimization further assumes a DIFF query with exactly two difference metrics: risk ratio and support.

4.1 DIFF–JOIN predicate pushdown

Suppose we have relations R, S, and T, with a common attribute a. In T, a is a primary key column, and in R and S, a is a foreign key column. T has additional columns $\mathcal{T} = \{t_1, \ldots, t_n\}$.

A common query in this setting is to evaluate the DIFF on `R NATURAL JOIN T` and `S NATURAL JOIN T`; we refer to this as a DIFF–JOIN query. Here, T effectively augments the space of features that the DIFF operator considers to include $\mathcal{T}$. This occurs frequently in real-world workflows: When finding explanations, many analysts wish to augment their datasets with additional metadata (e.g., hardware specifications, weather metadata) by executing primary key-foreign key JOINs [39]. For example, a production engineer who wants to explain a sudden increase in crash rate across a cluster may want to augment the crash logs from each server with its hardware specification and kernel version.

More formally, we wish to evaluate $\Delta_{\Gamma,\mathcal{A},k}(R \bowtie_a T, S \bowtie_a T)$, the k-order DIFF over $R \bowtie_a T$ and $S \bowtie_a T$. The naïve approach to evaluate this query would be to first evaluate each JOIN and then subsequently evaluate the DIFF on the two intermediate outputs. This can be costly, however—the JOINs may be expensive to evaluate [1,52,54,65], potentially more expensive than DIFF. Moreover, if the outputs of the JOINs contain few attribute value combinations that satisfy the difference metrics, then fully evaluating the JOINs becomes effectively a wasted step.

The challenge of efficiently evaluating DIFF in conjunction with one more JOINs is a specialized scenario of the multi-operator query optimization problem: A small estimation error in the size of one or more intermediate outputs can transitively yield a very large estimation error for the cost of the entire query plan [37]. This theoretical fact inspired extensive work in adaptive query processing [21], including systems such as Eddies [5] and RIO [7]. Here, we take a similar approach and design an adaptive algorithm for evaluating the DIFF–JOIN that avoids the pitfalls of expensive intermediate outputs.

Our adaptive algorithm is summarized in Algorithm 1. We start by evaluating DIFF on the foreign key columns in R and S (line 2), but without enforcing the support difference metric.

Evaluating the DIFF on the foreign keys gives us a set of candidate foreign keys K—these keys will map to candidate values in T. This is a form of predicate pushdown applied using the risk ratio: Rather than JOIN all tuples in T with R and S, we use the foreign keys to prune the tuples that do not need to be considered for evaluating the DIFF

Algorithm 1 DIFF-JOIN Predicate Pushdown, support and risk ratio

1: **procedure** DIFF- JOIN(R, S, T, k, $\mathcal{A}$, θ_{rr}, θ_{supp})
2: $K \leftarrow \Delta_{\Gamma=\theta_{\text{rr}},\mathcal{A},k}(\pi_a R, \pi_a S)$ ▷ DIFF, risk ratio only
3: **if** $|K| > threshold$ **then**
4: **return** $\Delta_{\Gamma=\{\theta_{\text{supp}},\theta_{\text{rr}}\},\mathcal{A},k}(R \bowtie T, S \bowtie T)$
5: $V \leftarrow K \rtimes T$
6: **for** $t \in T$ **do** ▷ each tuple
7: **for** $t_i \in t$ **do** ▷ each value
8: **if** $t_i \in V$ and t_i.**pk** $\notin K$ **then**
9: $V \leftarrow V \cup t$
return $\Delta_{\Gamma=\{\theta_{\text{supp}},\theta_{\text{rr}}\},\mathcal{A},k}(R \bowtie V, S \bowtie V)$

of $R \bowtie T$ and $S \bowtie T$. We cannot apply the same predicate pushdown using support, since multiple foreign keys can map to the same attribute in T, allowing low-support foreign keys to contribute to a high-support attribute. However, predicate pushdown via the risk ratio is mathematically possible: Suppose we have two foreign keys x and y, which both map to the same value v in T. The risk ratio of v—denoted $\text{rr}(v)$—is thus a weighted average of $\text{rr}(x)$ and $\text{rr}(y)$. This means that, if $\text{rr}(v)$ exceeds the threshold θ_{rr}, then either $\text{rr}(x)$ or $\text{rr}(y)$ must also exceed θ_{rr}. Therefore, the risk ratio difference metric can be applied on the column a, since at least one foreign key for a corresponding value in T will always be found.

We continue by semi-joining K with T, which yields our preliminary set of candidate values, V (line 5). However, the semi-join does not give us the complete set of possible candidates—because multiple foreign keys can map to the same value, there may be additional tuples in T that should be included in V. Thus, we loop over T again; if any attribute value in a tuple t is already present in V, but t's primary key is not found in K, then we add t to V (lines 6-9). We conclude by evaluating the DIFF on $R \bowtie V$ and $S \bowtie V$.

The technique of pushing down the difference metrics to the foreign keys does not always guarantee a speedup—only when K is relatively small. Thus, on line 3, we compare the size of K against a pre-determined threshold. (In our experiments, we found that $threshold = 5000$ yielded the best performance.) If $|K|$ exceeds the threshold, then we abort the algorithm and evaluate the DIFF–JOIN query using the naïve approach. As we show in our evaluation, this adaptive strategy can yield up to 2× speedups on real-world queries over normalized datasets.

4.2 ANTI DIFF pruning

As discussed in Sect. 3, the ANTI DIFF of relations R and S can be translated into a `CUBE(R) - DIFF(R, S)`. Such an implementation, however, is potentially inefficient due to the materialization costs of the CUBE subquery.

Instead, we can leverage a key design decision (which we discuss in Sect. 5) in our implementation of the DIFF operator. In MB SQL, DIFF uses the Apriori algorithm to

explore the space of explanations in a bottom-up manner—Apriori will first examine single-attribute explanations before exploring higher-order ones. This choice, combined with the minimality property (discussed in Sect. 2.4), allows us to prune higher-order explanations based on the anti-monotone difference metrics of one or more of their lower-order subsets. We exploit this strategy in our implementation of `DIFF` to reduce query runtimes whenever possible.

This same insight can be applied when evaluating `ANTI DIFF` queries as well; rather than prune a candidate explanation when it fails to satisfy an anti-monotone difference metric (the behavior when executing `DIFF`), in the case of `ANTI DIFF`, we add it to the set of results to be returned. Likewise, when a candidate explanation has satisfied all of the difference metrics, we do the opposite of `DIFF` and prune it from the search space of explanations. For candidate explanations that are in neither case—they do not yet satisfy all the difference metrics, but their supersets could—we also include them in the output of `ANTI DIFF`. In totality, this optimization effectively computes `CUBE(R) - DIFF(R, S)` without materializing all possible explanations.[1] In Sect. 7.4.4, we show that our implementation of this technique provides a 2× speedup over `CUBE(R) - DIFF(R, S)` for `ANTI DIFF` queries over each of the datasets in our evaluation.

4.3 Leveraging hard functional dependencies

As previously described, extensive data collection and augmentation are commonplace in the monitoring and analytics workloads we consider. Datasets are commonly augmented with additional metadata, such as hardware specifications or geographic information, that can yield richer explanations. This, however, can lead to redundancies or dependencies in the data. In this section, we focus specifically on how *functional dependencies* can be used to optimize `DIFF` queries.

Given a relation R, a set of attributes $X \subseteq R$ functionally determines another set $Y \subseteq R$ if Y is a function of X. In other words, X functionally determines Y if knowing that a row contains some attribute $x \in X$ means the row must also contain another particular attribute $y \in Y$. This relationship is referred to as a hard functional dependency (FD) and is denoted as $X \rightarrow Y$. Examples of commonly found FDs include location-based FDs (Zip Code → City) and FDs arising from user-defined functions or derived features (Raw Temperature → Discretized Temperature). As the output of the `DIFF` operator is a set of user-facing explanations, returning results which contain multiple functionally dependent attributes is both computationally inefficient and distracting to the end user. Thus, we present a logical optimization that leverages FDs.

There are two classes of functional dependencies which we optimize differently; an example of each is shown in the following tables:

Device	Zip Code	City
iPhone	94016	SF
-	94119	SF
Galaxy S9	94134	SF

Device	Country	ISO Code
iPhone	France	–
iPhone	–	FR
Galaxy S9	India	–
Galaxy S9	–	IN

In the first class, we have attributes X Y where $X \rightarrow Y$ but not $Y \rightarrow X$. For example, zip code functionally determines city, but the reverse is not true. If we ignore this sort of functional dependency, we may end up with uninteresting results like those in the top table. These results are redundant *within each explanation*: the City column is redundant with the Zip Code column. We know that if `iPhone-94016` is an explanation, `iPhone-94016-SF` is as well. Likewise, if `iPhone-94016` is not an explanation, then `iPhone-94016-SF` must not be either. Therefore, the `DIFF` operator should not consider these combinations of columns.

In the second class of functional dependencies, there exist attributes X and Y where both $X \rightarrow Y$ and $Y \rightarrow X$. This means that X and Y are perfectly redundant with one another. For instance, in the second table, Country → ISO Code, and ISO Code → Country. Naïvely running `DIFF` over this dataset may return results as in the bottom table.

Here, the results are redundant across *different* explanations. Given the first and third explanations, we can derive the second and fourth, and vice versa. We do not need to run `DIFF` over both Country and ISO Code, because they provide identical information.

Depending on what types of functional dependencies are observed, the `DIFF` operator employs the following logical optimizations: (*i*) If $X \rightarrow Y$, do not consider or evaluate explanations containing both X and Y; (*ii*) $X \rightarrow Y$ and $Y \rightarrow X$, do not evaluate or consider explanations containing X. (Or alternatively, do not evaluate or consider explanations containing Y.) We evaluate the runtime speedups provided by each of these optimizations in our evaluation.

[1] To keep `ANTI DIFF` consistent with `CUBE(R) - DIFF(R, S)`, we also prune all explanations with no support in R.

4.4 Leveraging soft functional dependencies

We described two classes of hard functional dependencies and how they can be used to improve performance. In this section, we consider "soft" functional dependencies, which can also be used in a manner similar to hard FDs.

Soft functional dependencies are a generalization of the hard functional dependencies described in Sect. 4.3 [36]. Given a relation R, and two sets of attributes $X, Y \subseteq R$, we denoted a hard FD as $X \rightarrow Y$ if X functionally determines Y—that is, if the value of X determines the value of Y. On the other hand, in a soft FD, X only determines the value of Y with high likelihood. For instance, in the following example, the City column is a soft FD with the Country column. While Amsterdam refers to the capital of the Netherlands with high probability, it may also refer to the city in New York.

Device	Country	City
iPhone	USA	Amsterdam
-	Netherlands	Amsterdam
Galaxy S9	Netherlands	Amsterdam

Unlike with hard FDs, omitting the Country column in returned explanations may not be desired. However, we have encountered scenarios in which soft FDs can be treated similar to hard FDs, allowing the same optimizations to be applied. In particular, this occurs in the presence of noisy or erroneous data collection or random jitter induced in device measurements. Thus, we allow the user to specify when soft FDs should be treated the same as hard FDs. We quantify the performance improvement gained by allowing for soft FDs in Sect. 7.4.5.

4.5 Automatic FD detection

Thus far, we assumed FDs are provided in advance by the user. In this section, we describe how to efficiently identify FDs in datasets automatically.

To automatically detect functional dependencies, we use a portion of the CORDS algorithm [36]. Automatically identifying hard FDs can be performed by counting the cardinality of each column. If $|X| = |X, Y|$, then X functionally determines Y. If $|X|$ is close to $|X, Y|$, then we can denote this as being a soft FD relationship. More formally, a soft FD can be quantified by its strength, $\frac{|X|}{|X,Y|}$, where a strength of 1 indicates a hard FD. In addition to FDs, CORDS also computes the statistical correlation between columns via Chi-squared test. The strength of the FDs and correlations to be used by `DIFF` is a user-specified parameter.

As computing these values can be computationally expensive, CORDS instead uses data samples to approximate the strength of the FDs and correlations. [36] shows that a fixed size data sample is sufficient to capture these relationships, thus allowing the method to scale to large datasets (as we demonstrate in Sect. 7.4.5).

4.6 Variance pruning

As discussed earlier in Sect. 4.2, the choice of Apriori allows us to evaluate and apply difference metrics to explanations as the search space is being explored; thus, pruning candidate higher-order explanations allows us to apply statistical predicate pushdown techniques for specific difference metric combinations.

For example, for `DIFF` queries with mean shift and support difference metrics, we can use metric variance to filter out potential explanations and reduce query runtimes. Intuitively, if an explanation matches a set of records whose current means and variance make it impossible for any subset to exceed both the support and mean shift thresholds, then we can proactively avoid considering any higher-order extensions of this explanation.

The mean shift is given by

$$\gamma_{\text{ms}} = \text{mean}(R_{\mathcal{A}_*})/\text{mean}(S_{\mathcal{A}_*})$$

for records in the test relation R and control relation S that match an explanation with attributes $\mathcal{A}_*$ (i.e., $R_{\mathcal{A}_*}$ is shorthand for $\sigma_{\mathcal{A}_*}(R)$).

This means that if one can bound both $\text{mean}(R_{X_*})$ and $\text{mean}(S_{X_*})$ for X_*, a higher-order extension of $\mathcal{A}_*$ (i.e., $X_* \supseteq \mathcal{A}_*$), then one can avoid searching over any X_* that could not possibly exceed a given mean shift threshold.

By tracking the variance in addition to the mean, one can bound the total variance of records matching an explanation using a variance decomposition. Let $R_{\mathcal{A}_*}$ denote a random variable corresponding to sampling a metric x uniformly from $\mathcal{A}_*$, and let R_{X_*} denote the event that $x \in R_{X_*}$. Consider the variance of a metric $\text{Var}(R_{\mathcal{A}_*})$; using a classical decomposition, we can bound the variance by conditioning on the event that a randomly sampled metric x lies in R_{X_*}:

$$\begin{aligned}\text{Var}(R_{\mathcal{A}_*}) &= E[\text{Var}(R_{\mathcal{A}_*}|R_{X_*})] + \text{Var}[E[R_{\mathcal{A}_*}|R_{X_*}]] \\ &\geq \text{Var}[E[R_{\mathcal{A}_*}|R_{X_*}]]\end{aligned}$$

Now, note that $E[R_{\mathcal{A}_*}|R_{X_*}]$ is a random variable equal to either

1. $\text{mean}(R_{X_*})$, with probability $\frac{\text{count}(R_{X_*})}{\text{count}(R_{\mathcal{A}_*})}$
2. or $\text{mean}(R_{A_*} \backslash R_{X_*})$, with probability $1 - \frac{\text{count}(R_{X_*})}{\text{count}(R_{\mathcal{A}_*})}$

and with expected value $\text{mean}(R_{A_*})$. Thus, we have that

$$(\text{mean}(R_{X_*}) - \text{mean}(R_{\mathcal{A}_*}))^2 \cdot \frac{\text{count}(R_{X_*})}{\text{count}(R_{\mathcal{A}_*})} \leq \text{Var}(R_{\mathcal{A}_*})$$

Given a support threshold θ_s, we know that $\frac{\text{count}(R_{X_*})}{\text{count}(R)} \geq \theta_s$ for any explanation with valid support. We can combine these two inequalities to derive upper and lower bounds on $\text{mean}(R_{X_*})$, $\text{mean}(S_{X_*})$, and $\gamma_{\text{ms}}(X_*)$, thereby enabling us to prune any superset of of the explanation $\mathcal{A}_*$ that does not meet the mean shift threshold.

In our implementation, we therefore must compute an additional aggregation function to calculate the variance $\text{Var}(R_{X_*})$ over various explanations. This can be done cheaply by aggregating the sum of squares of values in a dataset and using the identity

$$\text{Var}(R_{X_*}) = \left(\frac{1}{|R_{X_*}|}\sum_{y \in R_{X_*}} y^2\right) - \text{mean}(R_{X_*})^2.$$

In practice, we found that the speedup from additional pruning outweighs the cost of evaluating the variance and checking these inequalities.

4.7 Multi-query optimization

DIFF users often wish to query the same dataset with several different difference metric thresholds. For example, if a developer is using DIFF with support and risk ratio as metrics, they may wish to query with high support and low risk ratio thresholds and then again with low support and high risk ratio thresholds, to determine how to best tune these metrics to discover the most interesting insights about their data. DIFF supports such query patterns with its *multi-query* optimization, which maximizes the performance of multiple queries run sequentially.

DIFF's multi-query optimization assumes a user is sequentially running a set of queries on the same dataset with the same difference metrics but different thresholds. It minimizes the time spent running the queries. To do this, it constructs a single super-query whose output is a superset of the union of the outputs of the original queries. DIFF then runs each of the original queries as a filter over the output of the super-query.

We will now explain how to construct the super-query. Assume a user wishes to run k sequential DIFF queries. The queries all share the same m anti-monotonic difference metrics with thresholds $\theta_{a1} \ldots \theta_{am}$ and the same n non-anti-monotonic difference metrics with thresholds $\theta_1 \ldots \theta_n$. The super-query must choose values of each θ_a and θ to maximize performance while still returning all explanations returned by any of the original queries.

For each anti-monotonic difference metric, the super-query will use the smallest of the original query thresholds: $\min_i \theta_{ai}$. This is because DIFF prunes candidate explanations using the anti-monotonicity of these difference metrics, pruning a candidate explanation and all its supersets if it scores below the threshold of any anti-monotonic difference metric. Any candidate explanation that cannot pass the smallest of the original thresholds would not have been returned by any of the original queries.

Handling non-anti-monotonic difference metrics is more complicated because DIFF prunes via minimality, pruning all supersets of a candidate explanation that passes all difference metric thresholds. However, if a candidate explanation fails to pass the threshold of a non-anti-monotonic difference metric, one of its supersets may still pass the threshold. Therefore, if the super-query used minimal query thresholds for non-anti-monotonic difference metrics, it may prune higher-order candidate explanations that were part of the outputs of original queries with higher thresholds on those difference metrics. To get around this, the super-query modifies the minimality rule to only prune supersets of candidate explanations that score above the *largest* of the original query thresholds for non-anti-monotonic difference metrics.

The output of the super-query is a superset of union of the outputs of the original queries. Therefore, to answer one of the original queries, DIFF filters the super-query's output with the original query's difference metrics. This is guaranteed to return the same result as the original query. As we show in Sect. 7, the multi-query optimization improves performance by up to 3× when running four queries sequentially.

5 Physical optimizations for DIFF

In this section, we discuss the core algorithm underlying DIFF, a generalization of the Apriori algorithm [2] from the Frequent Itemset Mining (FIM) literature. Based on the assumptions from Sect. 2.4, we apply several physical optimizations, including novel techniques that exploit specific properties of our datasets and relational model to deliver speedups of up to 17×.

5.1 Algorithms

The DIFF operator uses the Apriori itemset mining algorithm [2] as its core subroutine for finding explanations (i.e., itemsets of attributes). Apriori was developed to efficiently find all itemsets in a dataset that exceed a support threshold. We chose Apriori instead of other alternatives, such as FPGrowth, because it is simple and perfectly parallel, making it easy to distribute and scale.

Our Apriori implementation is a generalization of the original Apriori introduced in [2]. We build a map from itemsets

of attributes to sets of aggregates. For each explanation order k, we iterate through all itemsets of attributes of size k in all rows of the dataset. Upon encountering an itemset, we check whether all subsets of order $k-1$ pass all anti-monotonic difference metrics. If they did, we update each of its aggregates. After iterating through all rows and itemsets for a particular k, we evaluate all difference metrics on the sets of aggregates associated with itemsets of size k. If an itemset passes all difference metrics, we return it to the user. If it only passes the anti-monotonic difference metrics, we consider the itemset during the subsequent $k+1$ pass over the data. However, if it fails any of the anti-monotonic difference metrics, we can prune the itemset from further consideration in high orders.

While Apriori gives us a scalable algorithm to find explanations, it performs poorly when applied naïvely to high-dimensional, relational data of varying cardinalities. In particular, the many reads and writes to the itemset-aggregate map become a bottleneck at large scales. We now discuss our optimizations that address performance.

5.2 Packed integers and column ordering

To improve the performance of the itemset-aggregate map at query time, we encode on the fly each unique value in the dataset whose frequency exceeds the support threshold as a 21-bit integer. This is done by building a frequency map per column and then discarding entries from each map that do not meet the support threshold. With this encoding, all explanations can be represented using a single 64-bit integer, for k up to and including 3. This allows us to index our map with single packed integers instead of with arrays of integers, improving overall runtimes by up to 1.7×. This optimization is possible because the total number of unique values in our datasets never exceeds 2^{21} even with a support threshold of 0. If the total number of unique values does exceed 2^{21}, we do not perform this optimization and instead store itemsets as arrays of integers.

To improve the map's read performance, we borrow from prior research [44,63] and use a columnar storage format for our data layout strategy. Because most reads are for a handful of high-frequency itemsets, this improves cache performance by avoiding cache misses on those itemsets, improving runtimes by up to 1.9×.

5.3 Bitmaps

We can further optimize `DIFF` by leveraging bitmap indexes, a strategy used in MAFIA [15] and other optimized Apriori implementations [6,26,70]. We encode each column as a collection of bitmaps, one for each unique value in the column. Each bitmap's ith bit is 1 if the column contained the value in its ith position and 0 otherwise. To compute the frequency

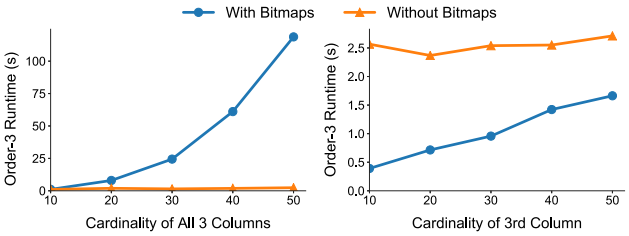

Fig. 2 Bitmap versus non-bitmap performance mining 3-order itemsets. Left: all columns share same cardinality. Right: two columns have fixed cardinality and the third varies

of an itemset, we count the number of ones in the bitwise `AND` of the bitmaps corresponding to the itemset.

However, the cost of using bitmaps—both compressed (e.g., Roaring Bitmaps [16]) and uncompressed—in the context of Apriori can be prohibitive for high-cardinality datasets, which prior work does not consider. While each individual `AND` is fast, the number of potential bitmaps is proportional to the number of distinct values in the dataset. Additionally, the number of `AND` operations is proportional to $\binom{n}{3}$, where n is the number of distinct elements in a given column. This trade-off between intersection speed and number of operations is true for compressed bitmaps as well: `AND`s are faster with Roaring Bitmaps only when the bitmaps are sufficiently sparse, which only holds true for very large n. In our evaluation, we find that using bitmaps for the `CMS` dataset with support 0.001 would require computing over 4M `AND`s of bitmaps with 15M bits each.

To combat these issues, we develop a per-column cost model to determine whether bitmaps speed up processing prior to mining itemsets from a set of columns. This is possible because our data originate in relational tables, so we know the cardinality of our columns in advance. The runtime of `DIFF` without bitmaps on a set of columns is independent of column cardinalities. However, the runtime of `DIFF` with bitmaps is proportional to the product of the cardinalities of each column. We demonstrate this in Fig. 2. On the left, we run `DIFF` on three synthetic columns with varying cardinality and find that bitmap runtime increases with the product of the cardinalities of the three columns, while non-bitmap runtime does not change. On the right, we fix the cardinalities of two columns and vary the third and find that bitmap runtime increases linearly with the varied cardinality, while non-bitmap runtime again does not change.

Given these characteristics of the runtime, a simple cost model presents itself naturally. Given a set of columns of cardinalities $c_1 \ldots c_N$, we should mine itemsets from those columns using bitmaps if the product $c_1 * c_2 \ldots * c_N < r$. Here, r is a parameter derived empirically from experiments similar to those in Fig. 2. It is the ratio $\frac{t_{nb}}{t_b/(c_1 * c_2 \ldots c * N)}$ where t_{nb} and t_b are runtimes mining itemsets of order N from N columns with cardinalities $c_1 \ldots c_N$. We find that for a given

machine, r does not change significantly between datasets. Overall, we show in our evaluation that this selective use of bitmap indexes improves performance by up to 5×.

6 Implementation

To implement the `DIFF` and `ANTI DIFF` operators, we develop both single-node and distributed implementations (in Apache Spark) of our relational query engine, the latter of which is implemented in Apache Spark.[2] In this section, we focus on the distributed setting and describe how we integrate our operators in Spark SQL [4]. We evaluate the distributed scalability of `DIFF` in our evaluation.

6.1 MB SQL in Spark

For our distributed implementation, we integrate MB SQL with Spark SQL, which provides a reliable and optimized implementation of all standard SQL operators and stores structured data as relational DataFrames. We extend the Catalyst query optimizer—which allows developers to specify custom rule-based optimizations—to support our logical optimizations. For standard SQL queries, MB SQL defers execution to standard Spark SQL and Catalyst optimizations, while all MacroBase-specific queries, including the `DIFF` operator, are (*i*) optimized using our custom Catalyst rules and (*ii*) translated to equivalent Spark operators (e.g., `map`, `filter`, `reduce`, `groupBy`) that execute our optimized Apriori algorithm. In total, integrating the `DIFF` operator with Spark SQL requires ~1600 lines of Java code.

Pruning optimization for distributed setting A major bottleneck in the distributed Apriori algorithm is the reduce stage when merging per-node itemset-aggregate maps. Each node's map contains the number of occurrences for every single itemset, which can grow exponentially with order. Therefore, naïvely merging these maps across nodes can incur significant communication costs. For example, for `MS-Telemetry A`, the reduction of the itemset-aggregate maps is typically an order of magnitude more expensive than other stages of the computation.

To overcome this bottleneck, we prune each map locally *before* reducing, using the anti-monotonic pruning rules introduced in Sect. 2.4. Naïvely applying our pruning rules to each local map may incorrectly remove entries that satisfy the pruning rules on one node but not another. Therefore, we use a two-pass approach: In the first pass, we prune the local entries but preserve a copy of the original map. We reduce the keys of the pruned map into a set of all entries that pass our pruning rules on *any* node. Then, in the second pass, we use this set to prune the original maps and finally combine the pruned originals to get our result.

7 Evaluation

In this section, we evaluate our implementation of the `DIFF` and `ANTI DIFF` operators on a variety of datasets, queries, and settings. We show that both our single-node and distributed implementations can efficiently scale to large datasets, and we demonstrate that our logical and physical optimizations substantially improve query performance.

7.1 Experimental setup

Single-node benchmarks were run on an Intel Xeon E5-2690 v4 CPU with 512 GB of memory. Distributed benchmarks were run on Spark v2.2.1 using a GCP cluster comprised of `n1-highmem-4` instances, with each worker equipped with 4 vCPUs from a 2.2GHz Intel E5 v4 (Broadwell) processor and 26 GB of memory.

7.2 Datasets

We benchmark the `DIFF` operator on the real-world datasets summarized in Tables 2 and 3. Unless otherwise specified, all queries are executed over all columns in the dataset and use as difference metrics support with a threshold of 0.01, risk ratio with a threshold of 2.0, and `MAX ORDER 3`. We measure and report the end-to-end query runtime, which includes the time to apply our integer packing, column ordering, and bitmap optimizations.

Telemetry at Microsoft With Microsoft's permission, we use two of their private datasets: `MS-Telemetry A` (60 GB, 175M rows, 13 columns) and `MS-Telemetry B` (26 GB, 37M rows, 15 columns). These consist of application telemetry data collected from their internal dashboarding system. In our benchmarks, we evaluate Microsoft's production `DIFF` queries on both datasets.

Censys Internet port scans We requested and obtained data from Censys, an Internet security research project that has now been commercialized [23]. Censys data are widely used in the Internet security research community and are freely accessible to researchers via a handshake agreement.[3] The dataset (75 GB, 400M rows, 17 columns) consists of port scans across the Internet from two separate days, 3 months apart, where each record represents a distinct IP address.

[2] Our implementation is open source and available at https://github.com/stanford-futuredata/macrobase.

[3] https://support.censys.io/hc/en-us/articles/360038761891-Research-Access-to-Censys-Data.

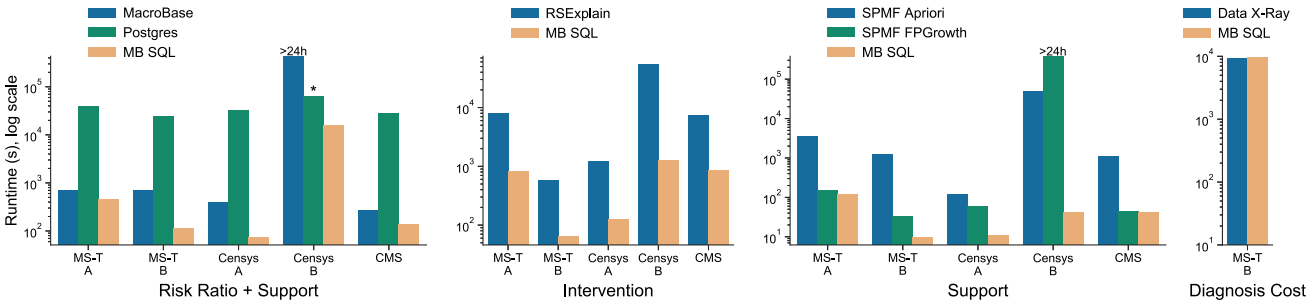

Fig. 3 Runtime comparison of MB SQL performance on `DIFF` queries with various difference metrics against the equivalent query in other explanation engines. Unless denoted with an asterisk, all queries were executed with `MAX ORDER 3`. `>24 h` indicates that the query did not complete in 24 h

For our single-node experiments, we generate two smaller versions of this dataset: `Censys A` (3.6 GB, 20M rows, 17 columns) and `Censys B` (2.6 GB, 8M rows, 102 columns). In our benchmarks, we evaluate `DIFF` queries comparing the port scans across the 2 days.

Center for Medicare Studies (CMS): The 7.7 GB (15M row) Center for Medicare Studies dataset, which is publicly available,[4], lists registered payments made by pharmaceutical and biotech companies to doctors. In our benchmarks, we evaluate a `DIFF` query comparing changes in payments made between 2 years (2013 and 2015).

We also benchmarked the scalability of the `DIFF` operator on a day's worth of anonymous scrolling performance data from a single table used by a production service at Facebook. In our benchmarks, we evaluate a `DIFF` query comparing the top 0.1% (p999) of events for a target metric against the remaining 99.9%. To simulate a production environment, we ran our benchmarks on a cluster located in a Facebook datacenter. Each worker in the cluster was equipped with 56 vCores, 228-GB RAM, and a 10 Gbps Ethernet connection.

7.3 End-to-end benchmarks

In this section, we evaluate the end-to-end performance of `DIFF`. We compare `DIFF`'s performance to other explanation engines as well as to other related systems such as frequent itemset miners, finding that performance is at worst equivalent and up to 9× faster on queries they support. We then evaluate distributed `DIFF`'s and find that it scales to hundreds of millions of rows of data and hundreds of cores.

7.3.1 Generality

In this section, we benchmark `DIFF` against the core subroutines of three other explanation engines: Data X-Ray [64], RSExplain [56], and the original MacroBase [8], matching their semantics using `DIFF` as described in Sect. 2. We also compare `DIFF` against Apriori and FPGrowth from SPMF [27] as well as SQL-equivalent `DIFF` queries described in Sect. 2.2 on Postgres. Results are shown in Fig. 3. All queries were executed with `MAX ORDER 3` except for the Postgres query on Censys B (denoted with an asterisk), which we ran with `ORDER 2` due to Postgres's limit on the number of `GROUP BY` columns.

Original MacroBase We first compare the performance of `DIFF` to that of the original MacroBase implementation [8]. We used support and risk ratio and measured end-to-end runtimes. We found that `DIFF` ranged from 1.6× faster on `MS-Telemetry A` to around 6× faster on `MS-Telemetry B` and `Censys A` than original MacroBase. In the much larger `Censys-B` dataset, `DIFF` finished in 4.5 h, while MacroBase could not finish in 24 h. The differences in performance here come from our physical optimizations.

Data X-Ray To compare against Data X-Ray, we create a difference metric from a Data X-ray cost metric, disable minimality, and feed the `DIFF` results into Data X-Ray's own set-cover algorithm taken from their implementation.[5] We benchmark Data X-Ray on `MS-Telemetry B` because it is our only dataset supporting a query—explaining systematic failures—that fits Data X-Ray's intended use cases. We attempted to run the benchmark on the entire dataset; however, we repeatedly encountered `OutOfMemory` errors from Data X-Ray's set-cover algorithm during our experiments. Therefore, we report the experimental results on a subset of `MS-Telemetry B` (1M rows). We do not observe any speedup, as the runtime of the set-cover solver dominated the overall runtime; with `DIFF`, we obtain effectively matching results. (Our performance is 2% worse.)

[4] https://www.cms.gov/OpenPayments/Explore-the-Data/Data-Overview.html.

[5] https://bitbucket.org/xlwang/dataxray-source-code.

Table 2 Datasets used for our single-node benchmarks

Dataset	File size (CSV) (GB)	# rows	# columns	# 3-order combos
Censys A	3.6	20M	17	19.5M
Censys B	2.6	8M	102	814.9M
CMS	7.7	15M	16	63.8M
MS-telemetry A	17	50M	13	73.4M
MS-telemetry B	13	19M	15	1.3B

Table 3 Datasets used for our distributed benchmarks

Dataset	File size (CSV) (GB)	# rows	# columns	# 3-order combos
Censys	75	400M	17	38M
MS-telemetry A	60	175M	13	132M

RSExplain To compare against RSExplain, we implement RSExplain's intervention metric as a difference metric and disable minimality. To compute a numerical query subject to the constraints described in Sect. 4.1 of the original paper, we calculate results for each individual query separately and then combine them per explanation. We evaluate the performance of this in queries on our single-node datasets, comparing the ratio of events in the later time period versus the earlier in `Censys A` and `CMS`, the ratio of high-latency to low-latency events in `MS-Telemetry A`, and the ratio of successful to total events in `MS-Telemetry B`. To reduce runtimes, we remove a handful (≤ 2) of columns with large numbers of unique values from each dataset. We found that the `DIFF` implementation was consistently between 8× and 10× faster at calculating intervention than the originally described algorithm.

Frequent Itemset Mining Though `DIFF` is semantically more general than Frequent Itemset Mining, we compare `DIFF`'s performance with popular FIM implementations. Specifically, we compare the runtime of the summarization step of `DIFF` to the runtimes of the Apriori and FPGrowth implementations in the popular Java data mining library SPMF[27]. When we run `DIFF` with only one difference metric, support, and disable the minimality property, `DIFF` is semantically equivalent to a frequent itemset miner. `DIFF` ranges from 11× faster on `Censys A` to 36× faster on `MB-Telemetry-B` than SPMF Apriori and from 1.1× faster on `MS-Telemetry A` to 3× faster on `MS-Telemetry B` and `Censys A` than SPMF FPGrowth at Frequent Itemset Mining. These speedups are due to the physical optimizations we discuss in Sect. 5.

Postgres We benchmark `DIFF` against a semantically equivalent SQL query in Postgres v10.3. We translate the support and risk ratio to Postgres UDFs and benchmark `DIFF` queries on the `CMS` and `Censys A` datasets. We find that `DIFF` is orders of magnitude faster than the equivalent SQL query.

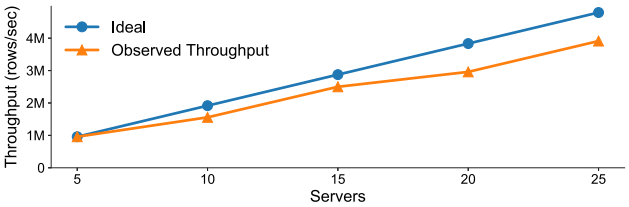

Fig. 4 Distributed scalability analysis of MB SQL in Spark. On a public GCP cluster, we evaluate a `DIFF` query with support of 0.001 and risk ratio of 2.0 on `Censys`. We observe a near-linear scale-up as we increase the number of servers

7.3.2 Distributed

We evaluate the scalability and performance of our distributed implementation of `DIFF`, described in Sect. 6, on our largest datasets.

Censys In Fig. 4, we run a `DIFF` using support and risk ratio on our largest public dataset, 400 million rows of Censys data, on a varying number of 4-core nodes. This query compares two Internet port scans (200M rows each) made 3 months apart to analyze how the Internet has changed in that time period. We find that this query scales well with increasing cluster sizes.

Facebook To measure the performance of MB SQL on even larger datasets, we ran a similar scalability experiment on a day's worth of anonymous scrolling performance data from a service at Facebook using one of their production clusters. Workers in the cluster are not reserved for individual Spark jobs, but are instead allocated as containers using a resource manager. We therefore benchmark the `DIFF` query for this service at the granularity of a Spark executor. Each executor receives 32 GB of RAM and four cores.

At 50 executors, MB SQL in Spark evaluated our `DIFF` query in approximately 2000 s. With fewer executors allocated, MB SQL's performance was slowed by significant memory pressure: More than 10% of the overall compute

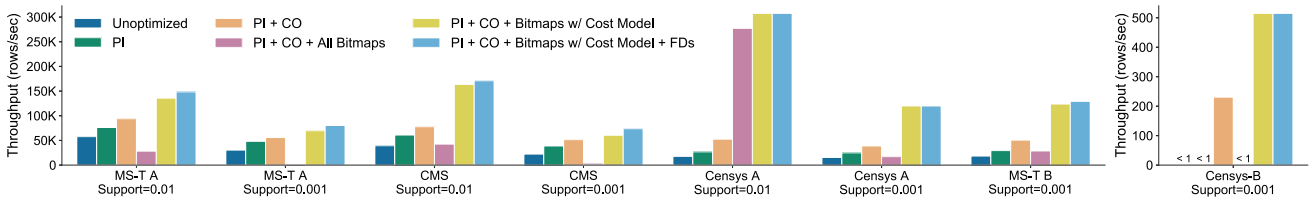

Fig. 5 Factor analysis of our optimizations, conducted on a single machine and core. We successively add all optimizations (PI: packed integers, CO: column ordered, FDs: functional dependencies) discussed in Sects. 4 and 5

time was spent on garbage collection, since the data itself took a significant fraction of the allocated memory. At 300 executors, which relieved the memory pressure, MB SQL in Spark evaluated the query in 1000 s.

7.4 Analysis of optimizations

In this section, we analyze the effectiveness of our physical and logical optimizations. We show that they improve query performance by up to 17× on a single node and 7× in a cluster.

7.4.1 Single-node factor analysis

We conduct a factor analysis to evaluate the effectiveness of our proposed physical optimizations (Sect. 5), with the results shown in Fig. 5. Our efficient encoding scheme (`Packed Integers`) and layout scheme (`Column Ordered`) produce substantial gains in performance, improving it by up to 1.7× and 1.9×, respectively. Applying bitmaps to all columns (`All Bitmaps`) improves performance by up to 5× on datasets with low-cardinality columns, such as `Censys A` with a high-support threshold, but performs poorly when columns have high cardinalities. To decide when bitmaps are appropriate, we use the cost model in Sect. 5 (`Bitmap w/ Cost Model`), which produces speedups on all datasets and queries.

We also evaluate the performance of our functional dependency optimization described in Sect. 4 (`FDs`) and find that it produces speedups of up to 1.2× in all datasets except `Censys A` and `Censys B`, which had no FDs. As previously discussed, we achieve these speedups by leveraging the functional dependencies to prune redundant explanations: For the `CMS` dataset, 14.5% of the explanations were pruned; for `MS-Telemetry-A`, 26.7% were pruned; and for `MS-Telemetry-B`, 5.4% were pruned. In total, our optimized implementation is 2.5–17× faster than our unoptimized implementation.

7.4.2 Distributed factor analysis

In Fig. 6, we conduct a factor analysis to study the effects of our optimizations in the distributed setting. Because, to our knowledge, no other explanation engines have open-source distributed implementations to compare to, we benchmark against a popular distributed frequent itemset miner, the parallel FPGrowth algorithm first described in [46] and implemented as part of Spark's `MLlib` library. We run our experiments on all 400 million rows of `Censys` on a cluster of 25 four-core machines and report throughput.

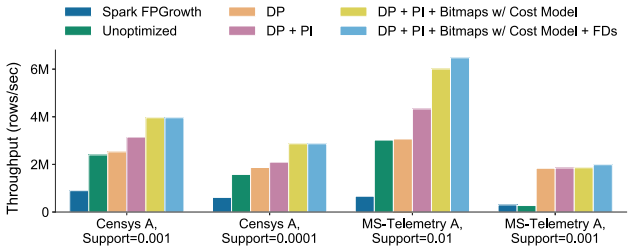

Fig. 6 Factor analysis of distributed optimizations, conducted on 25 four-core machines. We successively add all optimizations (DP: distributed pruning, PI: packed integers, FDs: functional dependencies) discussed in Sects. 4, 5, and 6 and report corresponding throughput

We find that MB SQL's `DIFF` consistently outperforms Spark's FPGrowth by 2.5× to 4.5×. Even *unoptimized* `DIFF` outperforms Spark FPGrowth, because our benchmark `DIFF` queries return low-order explanations ($k \leq 3$), while FPGrowth is designed for finding higher-order explanations common in Frequent Itemset Mining. Analyzing the optimizations individually, we find that our efficient encoding and bitmap schemes produce similar speedups as on a single core. FDs produce a 10% speedup on `MS-Telemetry A`. (No functional dependencies were found in `Censys`.) Our distributed pruning optimization produces speedups of up to 6× on datasets with high-cardinality columns.

7.4.3 DIFF–JOIN

In this section, we evaluate the performance of our `DIFF–JOIN` logical optimization. First, we apply the `DIFF–JOIN` predicate pushdown algorithm on a normalized version of `MS-Telemetry B` that requires a `NATURAL JOIN` between two fact tables R and S and a single dimension table T. We benchmark our optimization against the naïve approach and find that it improves the query response time by 2×.

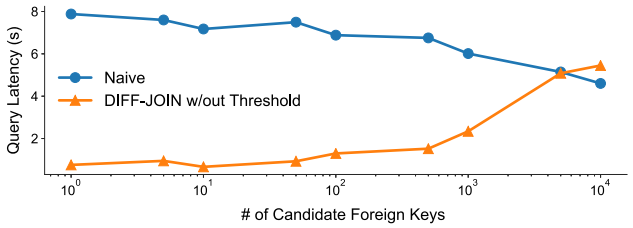

Fig. 7 Runtime of the DIFF–JOIN predicate pushdown algorithm with *threshold* disabled versus the naïve approach as $|K|$ is varied. $|R| = |S| = 1$M rows, and $|T| = 100$K rows with 4 columns. The DIFF query is run with risk ratio = 10.0 and support = 10^{-4}

In Fig. 7, we conduct a synthetic experiment to illustrate the relationship between $|K|$, the number of candidate foreign keys, and our DIFF–JOIN optimization. We set $|R|$ and $|S|$ to be 1M rows and $|T|$ to be 100K rows with four attributes. In R, we set a subset of foreign keys to occur with a relative frequency compared to S, ensuring that this subset becomes K. Then, we measure the runtime of both our algorithm and the naïve approach on a DIFF query with risk ratio and support thresholds of 10.0 and 10^{-4}, respectively.

At $|K| = 5000$, the runtimes of both are roughly equivalent, confirming our setting of *threshold*. As $|K|$ increases, we find that the runtime of the DIFF–JOIN predicate pushdown algorithm increases as well—a larger $|K|$ leads to a larger V, the set of candidate values, which, in turn, leads to a more expensive JOIN between R and V, and S and V. For the naïve approach, a larger K leads to shorter overall runtime due to less time spent in the Apriori stage. The larger set of candidate keys leads to many single-order and fewer higher-order attribute combinations, and the Apriori algorithm spends less time exploring higher-order itemsets.

7.4.4 ANTI DIFF

To demonstrate the effectiveness of our inverted pruning rules for ANTI DIFF, we take each of the Risk Ratio + Support DIFF queries from Fig. 3 and benchmark their ANTI DIFF equivalents on all six datasets. For each dataset, we execute the ANTI DIFF operator using two approaches: (*i*) ANTI DIFF(R, S) = CUBE(R) - DIFF(R, S) and (*ii*) using the pruning optimizations described in Sect. 4.2. Figure 8 shows the observed query throughput for both implementations. Across the board, we see that our pruning technique achieves 1.9-2.2× speedups compared to CUBE(R) - DIFF(R, S). While the runtimes of DIFF and ANTI DIFF are comparable, the CUBE subquery adds additional overhead, making it uncompetitive against our optimized approach. (Note that the execution of CUBE also uses the same bitmap-based optimizations used by DIFF, thus substantially reducing its potential overhead.)

7.4.5 Functional dependencies

To evaluate the performance improvement from including soft FDs and automatic FD detection, we construct synthetic datasets consisting of several FDs of varying strength. All datasets consist of a binary metric column to separate the two populations to run against DIFF and 40 attribute columns for explanation. All columns for each dataset are constructed the same way, with the datasets consisting of 1M, 10M, and 100M rows. Five columns are constructed to return explanations (correlated with the metric column). Ten columns are not correlated with the metric column and thus will not appear in explanations. The remainder are columns that are FDs between one of the five explanation columns, at varying levels of strength. These FD columns are created such that they will be recovered by running a method for automatic FD detection, as described in Sect. 4.5, at varying strength threshold values.

In Fig. 9, we compare the runtime of DIFF with and without FDs. We display the runtime of DIFF run over the 10M datapoint dataset as we vary the soft FD detection threshold. The choice of detection threshold is decreased in order to successively recover the FDs of varying strength constructed as described above. We find that once the threshold is reduced

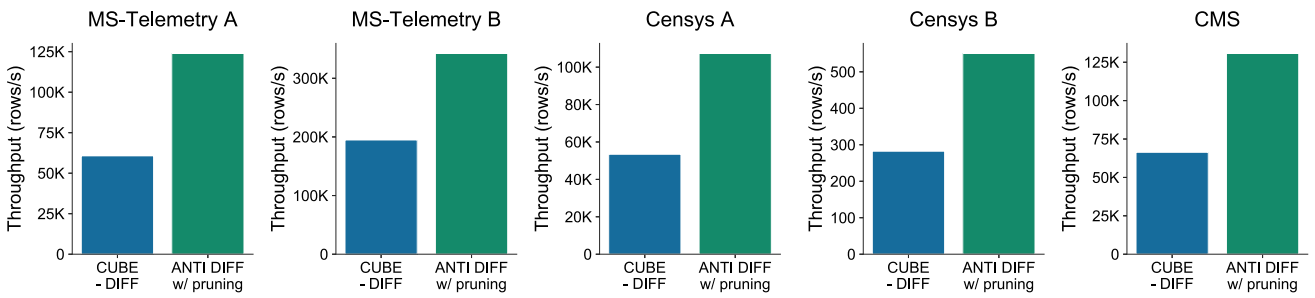

Fig. 8 Throughput comparison between the naïve implementation of ANTI DIFF(CUBE(R, S) - DIFF(R)) and our optimized implementation, which leverages the inverted set of pruning rules. Results averaged over five separate trials

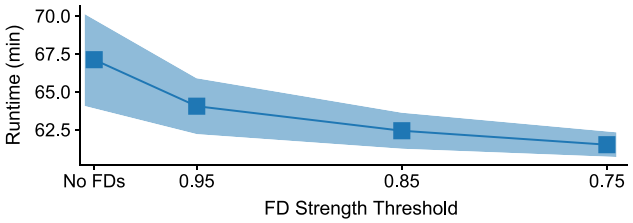

Fig. 9 Runtime improvements from using soft functional dependencies. As the FD strength threshold is lowered to capture more FDs, the overall `DIFF` runtime decreases

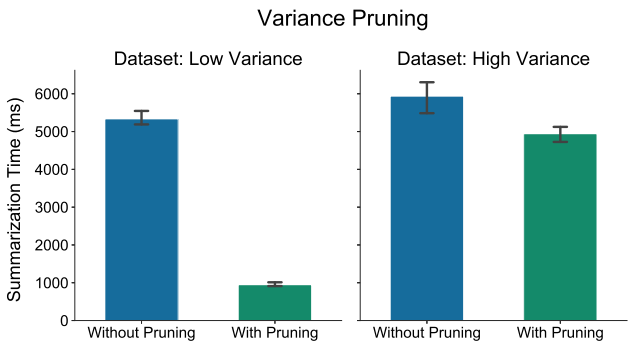

Fig. 10 Variance pruning significantly improves mean shift query performance on datasets with low variance, while providing much smaller improvements on datasets with high variance

to identify all induced FDs, we obtain up to a 1.1× speedup, corresponding to a 5.6-min decrease in wall-clock runtime.

As the FD detection algorithm operates on data samples, we also verify that the runtime overhead induced by FD detection does not grow proportional to dataset size. We run the FD detection algorithm over our datasets of varying sizes (1M, 10M, 100M) and confirm that the runtime remains constant. On average across five trials, we found that the 1M dataset required 7.5 s, the 10M dataset required 7.41 s, and the 100M dataset required 7.27 s.

7.4.6 Variance pruning

As described in Sect. 4.6, additional pruning is possible when users issue `DIFF` queries with both mean shift and support quality metrics. We evaluate the effectiveness of these pruning optimizations on a synthetic benchmark dataset with R and S consisting of 1 million records each with 15 dimensions $d_1 \ldots d_{15}$, each uniformly distributed over ten possible attribute values. The benchmark has a real-valued metric H which is normally distributed $\mathcal{N}(10, r)$ unless the record has $d_0 = 1$ in which case H is normally distributed $\mathcal{N}(50, 5r)$ for a real-valued parameter r that describes the internal variance of the data. In our experiments, $r = 1$ is used for a dataset with low variance and $r = 3$ is used for a dataset with high variance. We execute a `DIFF` query to identify the explanation $\mathcal{A}_* = \{d_0 = 1\}$ using a support threshold of $\theta_{\text{supp}} = 0.05$ and a mean shift threshold of $\theta_{\text{ms}} = 4.0$ and present the execution times for identifying explanations in Fig. 10. On datasets with low variance, our derived inequality yields stronger bounds on the means of the candidate explanations, and variance pruning is able to achieve up to a 5× speedup.

7.4.7 Multi-query optimization

In this section, we evaluate the performance of MB SQL's multi-query optimization. On each dataset, we run four queries sequentially with different difference metric thresholds, with and without the multi-query optimization; the results are shown in Fig. 11. All four queries used support and risk ratio as difference metrics. In each case, one query had a high support and low risk ratio threshold, one had a low support and high risk ratio threshold, one had both low, and one had both high. We find that the optimization improves performance by 2–3×. The performance of the multi-query is always similar to the performance of the slowest original query; the magnitude of the overall speedup depends on the differences between the original queries' performances.

7.5 Comparison of difference metrics and parameters

To evaluate the relative performance of MB SQL's `DIFF` running with different difference metrics, we compare their runtimes on `CMS`. The results are shown in Fig. 12. The `Support Only` query, equivalent to classical FIM, is fastest due to its simplicity and amenability to bitmap optimizations. Combining support with risk ratio (`Risk Ratio`) or odds ratio (`Odds Ratio`) yielded a slightly slower query. Combining support with the mean shift metric is even slower, since the mean shift cannot take advantage of our bitmap optimizations.

To evaluate how support and ratio thresholds affect the performance of our `DIFF` implementation, we picked two representative queries (`Risk Ratio` and `Mean Shift`) and ran them with varying support and ratio. The results are shown in Fig. 13. In the left chart, we confirm that decreasing support increases runtime. At low supports, `Mean Shift` outperforms `Risk Ratio` because the `Mean Shift` pruning rules are more efficient. (`Mean Shift` requires itemsets to be supported among both test and control rows, but `Risk Ratio` only among test rows.) At higher supports, `Risk Ratio` becomes faster as it can take advantage of bitmap optimizations. Examining the right chart in Fig. 13, we find that decreasing either the risk ratio or the mean shift ratio decreases runtime. This is attributable to the minimality rule in Sect. 2.4. At low ratios, most low-order itemsets that pass the support threshold also pass the ratio threshold, so their higher-order supersets never need to be considered. With high

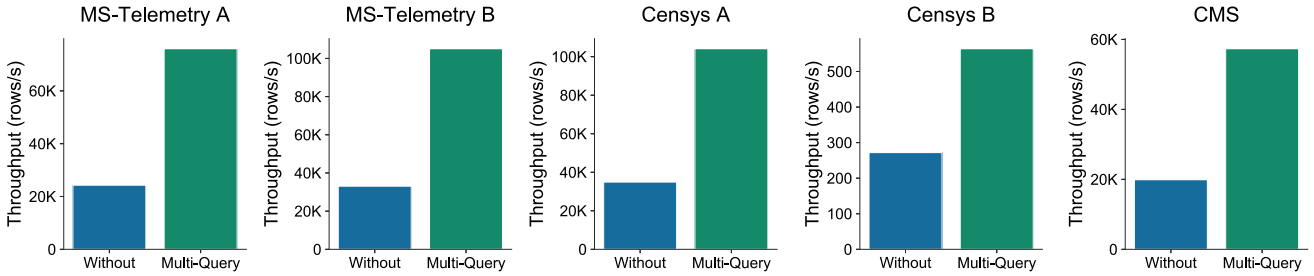

Fig. 11 Throughput comparison of four queries run sequentially on the same dataset with different difference metric thresholds, with and without multi-query optimization

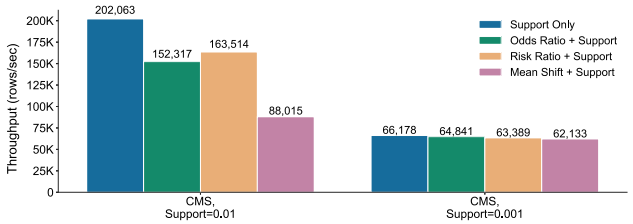

Fig. 12 Throughput of different difference metrics on `CMS`, with a fixed ratio of 2.0 (except for `Support Only`)

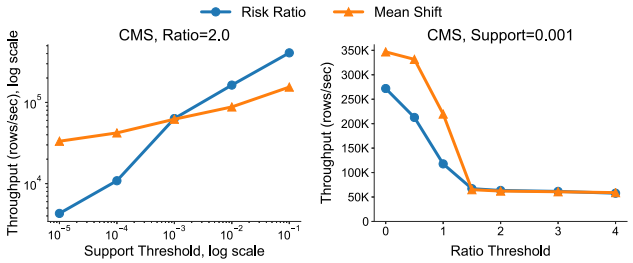

Fig. 13 Throughput of two `DIFF` queries with varying support (left) and ratio (right) thresholds

ratios, fewer itemsets are pruned by minimality, so more must be considered.

8 Industrial workloads

In this section, we share lessons learned from our 2-year experience deploying the MacroBase explanation engine in a large-scale industrial setting at Microsoft. We also discuss early adoption of the `DIFF` operator in a deployment at Facebook, as well as early lessons learned from an ongoing pilot study with the Censys security research project.

8.1 Microsoft deployment

The `DIFF` operator was refined as part of a collaboration with cloud infrastructure teams at Microsoft. Engineers on these teams monitor application metrics by looking at aggregated dashboards as well as automated alerts. One common need, as seen with MacroBase [8], is to explain anomalous events such as latency or failure rate spikes [20,30]. Digging into these anomalies required manual inspection of the raw event logs behind the dashboards and running handcrafted SQL group by queries to identify potential correlations. Moreover, going from dashboard to handcrafted SQL and executing multiple expensive group by queries imposed significant overheads on engineer productivity. These requirements motivated the need for an operator that was flexible enough to capture multiple anomaly types visible on the dashboard and scalable enough to provide interactive results on large data streams.

Table 4 Test and control relations in use at Microsoft

Split type	Description
Movers	Before-and-after comparisons across user-specified time windows. The analyst determines which customer cohorts exhibited the most change between two given days, weeks, months, etc.
Outliers	Outliers vs inliers on a user-specified metric. The analyst determines which features are most highly correlated with the outlying behavior observed for the metric
Population	Comparing a user-specified subset of the input data against the entire dataset. The analyst determines which features are unique or atypical to a particular subpopulation in the data

With help from our collaborators, we integrated the `DIFF` operator so that it is available to other engineers at Microsoft primarily through a dashboard interface, which can run a structured subset of `DIFF` queries on different subsets of the data. Engineers at Microsoft have found three types of test/-control pairings to be most commonly useful, so options for executing `DIFF` on these test/control datasets are available as part of the "Transforms" section of the dashboard. These three types of test/control splits are summarized in Table 4.

In Fig. 14, we illustrate how a *movers* `DIFF` query allows users to specify time ranges in a dashboard and identify potential explanations for an observed query count spike on January 15. In this case, the `DIFF` operator was able to identify a number of request origin categories that were highly

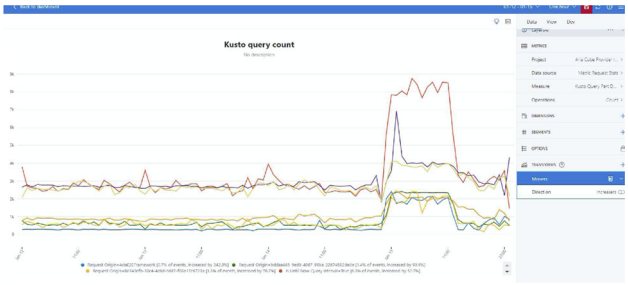

Fig. 14 A dashboard interface for issuing DIFF queries at Microsoft allows users to search for attribute combinations correlated with changes across test and control relations

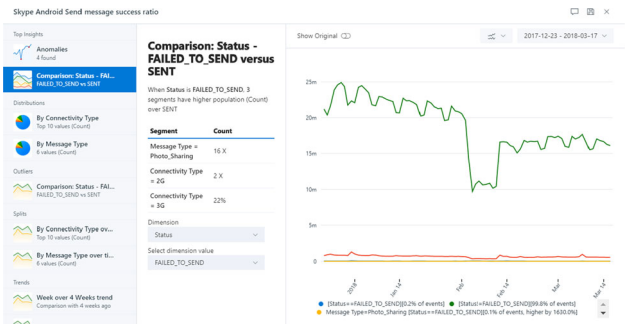

Fig. 15 An automated query runner at Microsoft—termed "smart analytics"—runs multiple DIFF queries automatically and shows a ranked list of the most interesting query results

correlated with an increase in requests on a user-specified time window.

In order to make DIFF queries more accessible to neophyte users, we have found that automatically running multiple DIFF queries on a dataset with varying parameters and options and then exposing the most significant results can also provide meaningful results. We illustrate one such usage of automatic DIFF queries in Fig. 15: Here, a population comparison DIFF query is used to discover that failed messages are highly correlated with both photo-sharing as well as mobile client network type.

All of these DIFF queries are integrated with the dashboard service at Microsoft via a REST API. Our implementation of the DIFF operator is exposed as a Kubernetes Web service which accepts requests specifying the query parameters as well as the dataset to operate over, which can be provided either inline or as an external http path and is often pre-aggregated by an external query service. Results are then returned as JSON to the dashboard service for display.

This integration has seen significant usage. In Fig. 16, we show counts for the number of requests to the DIFF executor service at Microsoft. Since Microsoft was part of the motivation for expanding the DIFF operator to support a flexible range of queries, we were pleased to see that multiple query types are all seeing use beyond the original outlier explanations introduced in MacroBase [8].

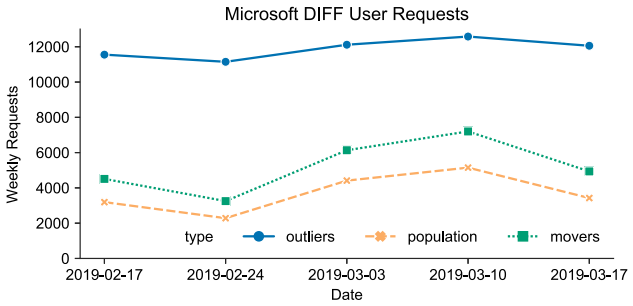

Fig. 16 Weekly DIFF requests from a dashboard service at Microsoft. All three DIFF query types supported by the dashboard see consistent week over week usage

8.2 Additional applications

In addition to the production deployment at Microsoft, we have seen success applying the DIFF operator in a variety of additional settings.

Facebook At Facebook, teams often evaluate the reliability and performance of different application and service features; if a feature performs unexpectedly on a given target metric, analysts must quickly find the root cause of the deviation. To do this, an analyst typically hypothesizes several dimensions that could be highly correlated with the metric's unforeseen performance; each hypothesis is then manually validated one at a time by executing multiple relational queries.

Existing explanation engines cannot be efficiently applied in this scenario, as they would only be capable of operating on small data subsets, and data movement is heavily discouraged, especially given that declarative relational workflows are already commonplace. With the DIFF operator, analysts can instead automate this procedure without leaving their workspace. In a matter of minutes, the analyst can execute a DIFF query evaluating an entire day of experiment dimensions, which directly reveals the combination of factors that are most responsible for the deviation.

Censys Censys is a search engine over Internet-wide port scan data [23], enabling security researchers to monitor how Internet devices, networks, and infrastructure change over time.

At Censys, researchers have performed thousands of Internet-wide scans consisting of trillions of probes, and these data have played a central role in analyzing and understanding some of the most significant Internet-scale vulnerabilities, such as Heartbleed [22] and the Mirai botnet [3]. Uncovering these vulnerabilities is often time-consuming—teams of researchers spend months analyzing Censys data to understand the genesis of the vulnerability.

Due to the high volume of these Internet-wide scans, distributed operators are required for scalable analyses—hence,

existing explanation engines are insufficient. In our pilot project, Censys researchers have used the DIFF operator to automate these analyses, allowing them to find potential security vulnerabilities as they evolve. For example, researchers can execute DIFF queries over scans from different time ranges (e.g., week over week or month over month), which reveal trends that are difficult to uncover through typical declarative relational analyses, such as bursts of activities on particular ports among a set of IP addresses, or a sharp drop in certain device types across several autonomous systems.

The high *velocity* of the Censys data has also tested the scalability of our implementation, even with our logical and physical optimizations. It has highlighted the potential role that sampling could play in quickly uncovering insights to users with DIFF: An intelligent sampling strategy could mitigate the ingest scalability challenges that Censys poses. It is important to stress, however, that straightforward approaches (e.g., uniform sampling) are not sufficient, particularly when attempting to explain anomalies or outliers at the scale of the Internet. Designing more clever and representative sampling techniques is an interesting area for future work.

9 Related work

Explanation query engines Many researchers have explored extending the functionality of databases to understand causality and answer explanation queries, starting with Sarawagi and Sathe's i^3 system. Unlike our proposed DIFF operator, Sarawagi and Sathe's DIFF finds differences between two different *cells* in an OLAP cube, rather than two relations (or alternatively, two data cubes). Subsequently, Fagin et al. [24,25] introduce a theoretical framework for answering explanation queries in their proposed data model, a multi-structural database. They propose a DIFFERENTIATE operator that, unlike DIFF, requires solving an NP-hard optimization problem for an exact answer.

We implemented DIFF as a relational interface for MacroBase; our difference metrics abstraction generalizes the support and risk ratio semantics introduced by Bailis et al. [8]. Others have proposed general frameworks for explanation queries: Roy et al. [56] developed a formal approach for explanation queries over multiple relations, but they require computing an entire data cube to evaluate queries. Wu and Madden [66] introduce a framework for finding explanations for aggregate queries on a single relation based on the notion of influence, which we can express using our DIFF operator. Finally, many specialized explanation query engines have also been proposed to explain performance bottlenecks [38,57,68] or diagnose data errors [64]; the DIFF operator allows us to express core subroutines of each of these systems using a single interface without sacrificing performance.

Feature selection Finding explanations for trends in large-scale datasets can be cast as a feature selection problem, an important task in machine learning [31,35,40,47,59]. Various feature selection techniques, such as compressive sensing [12], correlation-based tests [32], and tree-based approaches [55], are used to select a subset of relevant features (i.e., variables, predictors) to construct a machine learning model. Through our difference metric interface, the DIFF operator presents a generalizable approach for efficiently applying one or more correlation-based feature selection techniques (e.g., Chi-squared tests) and retrieving relevant features.

Multiple hypothesis testing Because a DIFF query can produce many explanations, it is potentially vulnerable to false positives (Type I errors). We can correct for this by calculating p values for our difference metrics, as Bailis et al. [8] do for risk ratio. We can then compare these p values to our desired confidence thresholds, applying corrections such as the Bonferroni correction [58] or the Benjamini–Hochberg procedure [13] to account for the number of explanations returned. We can then set our support threshold high enough that any explanation that passes it must be significant. In our experiments, our support thresholds were high enough given the relatively small number of explanations returned and relatively large number of rows in our datasets to ensure statistical significance.

Frequent Itemset Mining Our work draws inspiration from the Frequent Itemset Mining (FIM) literature [45]; specifically, the DIFF operator uses a variant of the Apriori algorithm [2] to explore different dimension combinations as itemsets, which are potential explanations that answer a given DIFF query. A substantial amount of prior work optimizes Apriori performance, such as applying bitmap indexes for faster candidate generation [6,15,26,70]. This previous work, however, mostly considers lists of transactions instead of relational tables and thus has no notion of column cardinality; we show in Sect. 7 that cardinality-aware bitmap indexes lead to substantial improvements for DIFF query evaluation. Additionally, prior work does not consider opportunities to optimize over relational data: Our experiments illustrate that we can exploit functional dependencies to prune the search space of Apriori and accelerate DIFF query performance. Proposals for custom FIM indexes in relational databases—such as I-trees [10] and IMine [11]—apply to FPGrowth [62], not Apriori.

OLAP query optimization Query optimization has long been a research focus for the database community [18,28,33,34]. In this paper, we present novel logical optimizations and an efficient physical implementation for DIFF, a new relational operator. Our physical optimizations leverage previous

techniques used to accelerate OLAP workloads, including columnar storage [63], dictionary encoding [49], and bitmap indexes [17,53]. Our implementation of DIFF requires a data-dependent application of these techniques that take into account the cardinality of individual attributes. With these improvements, the DIFF operator can be incorporated into existing OLAP warehouses, such as Druid [67] and Impala [14].

In addition, our proposed optimizations draw from research in adaptive query processing [5,7,21]. We show in Sect. 4 how to optimize DIFF–JOIN queries using our adaptive algorithm, which builds upon extensive work on optimizing JOINs [51,52,54,60,61]. Our algorithm also shares similarity with recent work examining the cost of materializing JOINs in machine learning workloads [19,42], including learning over JOINs [41]. Kumar et al. [43] study the impact of avoiding primary key-foreign key (KFK) JOINs during feature selection; they develop a set of information-theoretic decision rules to inform users when a KFK JOIN can be safely avoided without leading to a lower test accuracy for the downstream machine learning model. In our work, we assume that the JOIN is beneficial for the downstream model, and we design an adaptive algorithm for evaluating the JOIN efficiently in a data-dependent manner.

Lastly, our logical optimizations borrow from previous work on functional dependencies (FDs), such as CORDS [36], which mines datasets for FDs. In MB SQL, we do not focus on functional dependency discovery—we assume that they are provided by the user. Our contribution is a modified version of the Apriori algorithm that takes advantage of functional dependencies to prune the search space during candidate generation.

10 Conclusion

To combat the interoperability and scalability challenges common in large-scale data explanation tasks, we presented the DIFF operator, a declarative operator that unifies explanation and feature selection queries with relational analytics workloads. We also present ANTI DIFF, a companion operator that evaluates the complement of DIFF. Because both DIFF and ANTI DIFF are semantically equivalent to a standard relational query composed of UNION, GROUP BY , and CUBE operators, they integrate with current analytics pipelines, providing a solution for improved interoperability. Further, by providing logical and physical optimizations that take advantage of DIFF's relational model and usage patterns, we are able to scale to large industrial workloads across Microsoft and Facebook. We are continuing to develop the DIFF and ANTI DIFF operators with our collaborators, including Microsoft, Facebook, Censys, and Google, and hope to provide additional improvements to further boost data analyst productivity.

Acknowledgements We thank Kexin Rong, Hector Garcia-Molina, our colleagues in the Stanford DAWN Project, and the anonymous VLDB reviewers for their detailed feedback on earlier drafts of this work. This research was supported in part by affiliate members and other supporters of the Stanford DAWN project—Ant Financial, Facebook, Google, Intel, Microsoft, NEC, SAP, Teradata, and VMware—as well as Toyota Research Institute, Keysight Technologies, Hitachi, Northrop Grumman, Amazon Web Services, Juniper Networks, NetApp, and the NSF under CAREER grant CNS-1651570. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

A Translating DIFF to standard SQL

We present a sample DIFF query, borrowed from the Example Workflow in Sect. 2.1, and its translation into standard SQL.

```
SELECT * FROM
  (SELECT * FROM logs WHERE crash = true)
      crash_logs
DIFF
  (SELECT * FROM logs WHERE crash = false)
      success_logs
ON app_version, device_type, os
COMPARE BY risk_ratio >= 2.0, support >= 0.05
     MAX ORDER 3;
```

This query is equivalent to the following Postgres-compatible SQL query:

```
SELECT app_version,
       device_type,
       os,
       support(test_attr_count,
           test_global_count),
       risk_ratio(test_attr_count,
           control_attr_count,
           test_global_count,
           control_global_count)
  FROM (SELECT app_version,
               device_type,
               os,
               COUNT(*) as test_attr_count,
               (SELECT COUNT(*) FROM logs WHERE
                    crash = true) as
                   test_global_count
FROM logs WHERE crash = true
GROUP BY GROUPING SETS (
        (app_version),
        (device_type),
        (os),
        (app_version, device_type),
        (app_version, os),
        (device_type, os),
        (app_version, device_type, os)
    )
) t1 NATURAL JOIN
  (SELECT app_version,
      device_type,
      os,
      COUNT(*) as control_attr_count,
      (SELECT COUNT(*) FROM logs WHERE crash =
          false) as control_global_count
FROM logs WHERE crash = false
GROUP BY GROUPING SETS (
        (app_version),
        (device_type),
        (os),
```

```
        (app_version, device_type),
        (app_version, os),
        (device_type, os),
        (app_version, device_type, os)
    )
) t2
  WHERE support(test_attr_count,
      test_global_count) >= 0.05 AND
        risk_ratio(test_attr_count,
                   control_attr_count,
                   test_global_count,
                   control_global_count) >=
                       2.0;
```

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases: The Logical Level. Addison-Wesley Longman Publishing Co. Inc, Boston (1995)
2. Agarwal, R., Srikant, R., et al.: Fast algorithms for mining association rules. In: VLDB, pp. 487–499 (1994)
3. Antonakakis, M., April, T., Bailey, M., Bernhard, M., Bursztein, E., Cochran, J., Durumeric, Z., Halderman, J.A., Invernizzi, L., Kallitsis, M., Kumar, D., Lever, C., Ma, Z., Mason, J., Menscher, D., Seaman, C., Sullivan, N., Thomas, K., Zhou, Y.: Understanding the mirai botnet. In: USENIX Security (2017). https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis
4. Armbrust, M., et al.: Spark sql: relational data processing in spark. In: SIGMOD, pp. 1383–1394. ACM (2015)
5. Avnur, R., Hellerstein, J.M.: Eddies: continuously adaptive query processing. In: SIGMOD, vol. 29, pp. 261–272. ACM (2000)
6. Ayres, J., et al.: Sequential pattern mining using a bitmap representation. In: KDD, pp. 429–435. ACM (2002)
7. Babu, S., Bizarro, P., DeWitt, D.: Proactive re-optimization. In: SIGMOD, pp. 107–118. ACM (2005)
8. Bailis, P., Gan, E., Madden, S., Narayanan, D., Rong, K., Suri, S.: Macrobase: prioritizing attention in fast data. In: SIGMOD, pp. 541–556. ACM (2017)
9. Bailis, P., et al.: Prioritizing attention in fast data: principles and promise. In: CIDR. Google Scholar (2017)
10. Baralis, E., Cerquitelli, T., Chiusano, S.: Index support for frequent itemset mining in a relational dbms. In: ICDE, pp. 754–765. IEEE (2005)
11. Baralis, E., Cerquitelli, T., Chiusano, S.: Imine: index support for item set mining. IEEE Trans. Knowl. Data Eng. **21**(4), 493–506 (2009)
12. Baraniuk, R.G.: Compressive sensing [lecture notes]. IEEE Signal Process. Mag. **24**(4), 118–121 (2007)
13. Benjamini, Y., Yekutieli, D.: The control of the false discovery rate in multiple testing under dependency. Ann. Stat. **29**, 1165–1188 (2001)
14. Bittorf, M., et al.: Impala: a modern, open-source SQL engine for hadoop. In: CIDR (2015)
15. Burdick, D., Calimlim, M., Gehrke, J.: Mafia: a maximal frequent itemset algorithm for transactional databases. In: ICDE, pp. 443–452. IEEE (2001)
16. Chambi, S., et al.: Better bitmap performance with roaring bitmaps. Softw. Pract. Exp. **46**(5), 709–719 (2016)
17. Chambi, S., et al.: Optimizing druid with roaring bitmaps. In: IDEAS, pp. 77–86. ACM (2016)
18. Chaudhuri, S.: An overview of query optimization in relational systems. In: PODS, pp. 34–43. ACM (1998)
19. Chen, L., et al.: Towards linear algebra over normalized data. PVLDB **10**(11), 1214–1225 (2017)
20. Dean, J., Barroso, L.A.: The tail at scale. Commun. ACM **56**, 74–80 (2013)
21. Deshpande, A., et al.: Adaptive query processing. Found. Trends Databases **1**(1), 1–140 (2007)
22. Durumeric, Z., et al.: The matter of heartbleed. In: IMC, pp. 475–488. ACM (2014)
23. Durumeric, Z., et al.: A search engine backed by Internet-wide scanning. In: SIGSAC, pp. 542–553. ACM (2015)
24. Fagin, R., et al.: Efficient implementation of large-scale multi-structural databases. In: VLDB, pp. 958–969. VLDB Endowment (2005)
25. Fagin, R., et al.: Multi-structural databases. In: PODS, pp. 184–195. ACM (2005)
26. Fang, W., et al.: Frequent itemset mining on graphics processors. In: DaMoN, pp. 34–42. ACM (2009)
27. Fournier-Viger, P., et al.: The SPMF open-source data mining library version 2. In: Joint European conference on machine learning and knowledge discovery in databases, pp. 36–40. Springer (2016)
28. Graefe, G., McKenna, W.J.: The volcano optimizer generator: extensibility and efficient search. In: ICDE, pp. 209–218. IEEE (1993)
29. Gray, J., et al.: Data cube: a relational aggregation operator generalizing group-by, cross-tab, and sub-totals. Data Min. Knowl. Discov. **1**(1), 29–53 (1997)
30. Greenberg, A., et al.: The cost of a cloud: research problems in data center networks. ACM SIGCOMM Comput. Commun. Rev. **39**(1), 68–73 (2008)
31. Guyon, I., Elisseeff, A.: An introduction to variable and feature selection. J. Mach. Learn. Res. **3**(Mar), 1157–1182 (2003)
32. Hall, M.A.: Correlation-based feature selection of discrete and numeric class machine learning. Working Paper Series (2000)
33. Hellerstein, J.M., Stonebraker, M.: Readings in database systems. MIT press (2005)
34. Hellerstein, J.M., et al.: Architecture of a database system. Found. Trends® Databases **1**(2), 141–259 (2007)
35. Hoi, S.C., et al.: Online feature selection for mining big data. In: BigMine, pp. 93–100. ACM (2012)
36. Ilyas, I.F., et al.: Cords: automatic discovery of correlations and soft functional dependencies. In: SIGMOD, pp. 647–658. ACM (2004)
37. Ioannidis, Y.E., Christodoulakis, S.: On the Propagation of Errors in the Size of Join Results, vol. 20. ACM, New York (1991)
38. Khoussainova, N., Balazinska, M., Suciu, D.: Perfxplain: debugging mapreduce job performance. PVLDB **5**(7), 598–609 (2012)
39. Kimball, R., Ross, M.: The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling. Wiley, Hoboken (2011)
40. Konda, P., et al.: Feature selection in enterprise analytics: a demonstration using an r-based data analytics system. PVLDB **6**(12), 1306–1309 (2013)
41. Kumar, A.: Learning over joins. Ph.D. thesis, The University of Wisconsin-Madison (2016)
42. Kumar, A., Naughton, J., Patel, J.M.: Learning generalized linear models over normalized data. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp. 1969–1984. ACM (2015)
43. Kumar, A., et al.: To join or not to join?: thinking twice about joins before feature selection. In: SIGMOD, pp. 19–34. ACM (2016)
44. Lamb, A., et al.: The vertica analytic database: C-store 7 years later. VLDB **5**(12), 1790–1801 (2012)
45. Leskovec, J., et al.: Mining of Massive Datasets. Cambridge University Press, Cambridge (2014)
46. Li, H., et al.: Pfp: parallel fp-growth for query recommendation. In: RecSys, pp. 107–114. ACM (2008)
47. Li, J., et al.: Feature selection: a data perspective. ACM Comput. Surv. (CSUR) **50**(6), 94 (2017)

48. Meliou, A., Roy, S., Suciu, D.: Causality and explanations in databases. PVLDB **7**(13), 1715–1716 (2014)
49. Melnik, S., et al.: Dremel: interactive analysis of web-scale datasets. PVLDB **3**(1–2), 330–339 (2010)
50. Meng, X., et al.: Mllib: machine learning in apache spark. J. Mach. Learn. Res. **17**(1), 1235–1241 (2016)
51. Neumann, T., Radke, B.: Adaptive optimization of very large join queries. In: SIGMOD, pp. 677–692. ACM (2018)
52. Ngo, H.Q., et al.: Worst-case optimal join algorithms. J. ACM: JACM **65**(3), 16 (2018)
53. O'Neil, P., Quass, D.: Improved query performance with variant indexes. In: SIGMOD, vol. 26, pp. 38–49. ACM (1997)
54. Pagh, A., Pagh, R.: Scalable computation of acyclic joins. In: PODS, pp. 225–232. ACM (2006)
55. Rounds, E.: A combined nonparametric approach to feature selection and binary decision tree design. Pattern Recogn. **12**(5), 313–317 (1980)
56. Roy, S., Suciu, D.: A formal approach to finding explanations for database queries. In: SIGMOD, pp. 1579–1590. ACM (2014)
57. Roy, S., et al.: Perfaugur: robust diagnostics for performance anomalies in cloud services. In: 2015 IEEE 31st International Conference on Data Engineering (ICDE), pp. 1167–1178. IEEE (2015)
58. Rupert Jr., G., et al.: Simultaneous Statistical Inference. Springer, Berlin (2012)
59. Saeys, Y., Inza, I., Larrañaga, P.: A review of feature selection techniques in bioinformatics. Bioinformatics **23**(19), 2507–2517 (2007)
60. Schuh, S., Chen, X., Dittrich, J.: An experimental comparison of thirteen relational equi-joins in main memory. In: SIGMOD, pp. 1961–1976. ACM (2016)
61. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access path selection in a relational database management system. In: Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, pp. 23–34 (1979)
62. Shang, X., Sattler, KU., Geist, I.: SQL based frequent pattern mining with FP-growth. In: Seipel, D., Hanus, M., Geske, U., Bartenstein, O. (eds.) Applications of Declarative Programming and Knowledge Management. INAP 2004, WLP 2004. Lecture Notes in Computer Science, vol. 3392. Springer, Berlin, Heidelberg (2005). https://doi.org/10.1007/11415763_3
63. Stonebraker, M., et al.: C-store: a column-oriented dbms. In: VLDB, pp. 553–564. VLDB Endowment (2005)
64. Wang, X., et al.: Data x-ray: a diagnostic tool for data errors. In: SIGMOD, pp. 1231–1245. ACM (2015)
65. Willard, D.E.: Applications of range query theory to relational data base join and selection operations. J. Comput. Syst. Sci. **52**(1), 157–169 (1996)
66. Wu, E., Madden, S.: Scorpion: explaining away outliers in aggregate queries. PVLDB **6**(8), 553–564 (2013)
67. Yang, F., et al.: Druid: A real-time analytical data store. In: SIGMOD, pp. 157–168. ACM (2014)
68. Yoon, D.Y., Niu, N., Mozafari, B.: Dbsherlock: a performance diagnostic tool for transactional databases. In: SIGMOD, pp. 1599–1614. ACM (2016)
69. Zaharia, M., et al.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: NSDI, pp. 2–2. USENIX Association (2012)
70. Zhang, F., Zhang, Y., Bakos, J.: Gpapriori: Gpu-accelerated frequent itemset mining. In: 2011 IEEE International Conference on Cluster Computing (CLUSTER), pp. 590–594. IEEE (2011)