DIFF: A Relational Interface for Large-Scale Data Explanation

Firas Abuzaid, Peter Kraft, Sahaana Suri, Edward Gan, Eric Xu,
Atul Shenoy[†], Asvin Ananthanarayan[†], John Sheu[†], Erik Meijer[‡],
Xi Wu[§], Jeff Naughton[§], Peter Bailis, Matei Zaharia

Stanford DAWN Project, Microsoft[†], Facebook[‡], Google[§]

{fabuzaid, kraftp, sahaana, egan1, ericxu0, pbailis, mzaharia}@stanford.edu,
{atul.shenoy, asvina, jsheu}@microsoft.com, erikm@fb.com,
{wuxi, naughton}@google.com

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 standalone 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.

PVLDB Reference Format:
F. Abuzaid, P. Kraft, S. Suri, E. Gan, E. Xu, A. Shenoy, A. Ananthanarayan, J. Sheu, E. Meijer, X. Wu, J. Naughton, P. Bailis, M. Zaharia. DIFF: A Relational Interface for Large-Scale Data Explanation. *PVLDB*, 12(4): 419-432, 2018.
DOI: https://doi.org/10.14778/3297753.3297761

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, 49, 58, 65, 67]) 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 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 standalone 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 `JOIN`s prior to use in explanation analyses [40]. 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 explantation 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 upwards of ten minutes to complete.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 12, No. 4
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3297753.3297761

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 `JOIN`s?
2. What *physical layer optimizations*—algorithms, indexes, and storage formats—are needed to evaluate `DIFF` efficiently?

At the logical layer, we present two new optimizations for `DIFF` that are informed by the snowflake and star schemas common in data warehouses, where data is augmented with metadata via `JOIN`s before running explanation queries [40]. A naïve execution of this workflow would fully materialize the `JOIN`s and then evaluate `DIFF` on their output. We show that if the returned output size after computing the `JOIN`s far exceeds that of the `DIFF`, it is more efficient to perform the `DIFF` operation before materializing the `JOIN`s, thereby applying a predicate pushdown-style strategy to `DIFF-JOIN` queries (similar to learning over joins [42]). We introduce an adaptive algorithm that dynamically determines the order of operations, yielding up to 2× speedups on real data. In addition, for input data that contains functional dependencies, we show how to leverage these dependencies for further logical optimization. For instance, joining with a geographic dimension table may provide state and country information—but for explanation queries, returning both fields is redundant, as an explanation containing `state = "CA"` is equivalent to one containing `state = "CA", country = "USA"`. We show how pruning computation using these functional dependencies can yield up to 20% speedups.

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 [70], 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 [58], 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 [51], by up to 4.5×.

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, which can accelerate `DIFF` queries by up to 2×.
- We introduce an optimized physical implementation of the `DIFF` operator that combines dictionary encoding, columnar storage, and data-dependent, cost-based bitmap index, yielding up to a 17× improvement in performance.

2 Industrial Workloads

In this section, we share lessons learned from our two-year experience deploying the MacroBase explanation engine in several industrial settings. We first describe the scalability and interoperability challenges we encountered that led to the creation of the `DIFF` operator, our proposed relational aggregation operator that unifies explanation engines with declarative relational query engines. We then describe successful real-world, large-scale production applications of the `DIFF` operator at Microsoft, Facebook, and Censys.

2.1 Challenges: Scalability and Interoperability

Over the last two years, we have deployed the MacroBase explanation engine across a wide range of production use cases, including datacenter monitoring, application telemetry, performance diagnostics, time series analysis, and A/B testing. Through this process, we discovered two key factors limiting data analysts' ability to apply explanation engines such as MacroBase, Scorpion, Data X-Ray, and RSExplain to their production settings.

First, data analysts require engines capable of efficiently handling their growing industrial data volumes, which are expected to increase by 40% annually [36]. Explanation engines are typically not designed to scale to this degree; with each of these aforementioned engines, a typical explanation query over a 13GB subset of Microsoft production data requires upwards of 10 minutes on a single node. Except for Data X-Ray, these engines do not have distributed implementations, fundamentally limiting their scalability.

Second, data analysts typically search for explanations as part of a larger analytics workflow or environment. As explanation engines are not designed to interoperate with analysts' pipelines, and instead act as standalone data processing systems, making use of them is tedious: analysts must clean and export their data to an engine-compatible format, perform analyses using the engine, and then translate their results back to their relational query engines for further analyses. As a result, rapid exploration and iteration over the results of explanation queries becomes challenging and laborious.

Using these experiences and limitations as inspiration, we propose the `DIFF` operator as a declarative relational operator that serves as an extension and evolution of our initial MacroBase explanation engine. We demonstrate that the `DIFF` operator captures the core semantics of additional explanation engines and data mining techniques, including Data X-Ray, Scorpion, RSExplain, and Frequent Itemset Mining, in Section 3.3. In addition, we provide an efficient implementation of the `DIFF` operator in both the single-node and

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 vs. the control relation if it has an attribute vs. if it does not
Risk Ratio	Probability that a row will be in the test relation vs. the control relation if it has an attribute vs. if it does not
Mean Shift	Percent change in a column's mean for rows containing an attribute in the test vs. the control relation

distributed setting, improving runtime by an order of magnitude compared to existing explanation engines.

2.2 Production Applications

We now describe in detail a subset of the previously described applications that inspired and are now enabled by the DIFF operator.

Microsoft At Microsoft, engineers monitor application behavior via several dashboards of aggregated telemetry events in conjunction with preset alerts for anomalous behavior. If unusual behavior is detected, the engineers must perform manual root cause analyses, requiring inspection of logged events via these dashboards and relational query processing. This manual procedure is often tedious and time-consuming, especially as the underlying issue may not be immediately visible in existing dashboards. Moreover, not all performance bottlenecks will be caught by the alerts, and identifying these false negatives can be even more challenging.

Engineers can use existing explanation engines to perform this root cause analysis, but constantly exporting data from their dashboarding tools is labor-intensive, which leads to difficulties integrating explanation functionality with legacy services. Further, existing engines are incapable of scaling to the full size of their data of interest (hundreds of millions to billions of rows, or more). By incorporating the DIFF operator into their production dashboarding tools, engineers have instead been able to automatically discover the likely reasons for a range of common abnormal application behaviors, such as high tail latency [20] or low resource utilization [30]. Additionally, DIFF queries that compare large volumes of normal and abnormal raw application events can also reveal bottlenecks that preset alerts fail to catch.

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 this data has 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 can use the DIFF operator to automate these analyses, allowing them to find potential security vulnerabilities as they evolve. For example, a researcher can execute a DIFF query over scans from different time ranges (e.g., week-over-week or month-over-month), which reveals trends that are difficult to uncover through typical declarative relational analyses, such as bursts of activities on particular ports amongst a set of IP addresses, or a sharp drop in certain device types across several Autonomous Systems.

```
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
```

Figure 1: DIFF syntax in extended Backus-Naur Form

3 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 Section 3.3.

3.1 DIFF Operator Syntax and Example Workflow

We present syntax for the DIFF operator in Backus-Naur form in Figure 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 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 Section 3.3, 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 sufficient for our industrial use cases, the DIFF operator supports user-defined difference metrics as well.

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. She can make use of the DIFF operator using 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 3;
```

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 3 to specify that the returned explanations should never contain more than three attributes. Running this DIFF query, the developer obtains the following results:

app_version	device_type	os	risk_ratio	support
v1	-	-	10.5	15%
v2	iPhone X	-	7.25	30%
v3	Galaxy S9	11.0	9.75	20%

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 or ROLLUP 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 the first two results uninteresting—they may be known bugs. However, the third explanation warrants further study. 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.

3.2 Formal Definition of the DIFF Operator

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

Definition 3.1. Explanation
We define an *explanation* of *order* k to be a set of k attribute values:

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

We borrow this definition from prior work on explanation engines, including RSExplain [58], Scorpion [67], and MacroBase [8]. In practice, explanations typically consist of categorical attributes, although our definition can extend to continous data ranges as well.

Definition 3.2. 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 if 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 $\mathcal{A}_*$ and 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: $\mathbb{R}^d \times \mathbb{R}^d \times \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$
- 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, $\mathcal{F}^R_{\text{global}} = \mathcal{F}(R)$, 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:

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

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:

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

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

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

Definition 3.3. 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

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 expalantion $\mathcal{A}_*$—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 to standard SQL using GROUP BYs and UNIONs, as we illustrate in our experiments that benchmark DIFF queries against equivalent SQL queries in Postgres. This translation step is costly, however, both for data analysts and for relational databases: the equivalent query is often hundreds of lines of SQL that database query planners fail to optimize and execute efficiently. 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 Sections 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.

3.3 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 Section 3.1. 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 [65] is an explanation engine that diagnoses systematic errors in large-scale datasets. From Definition 10 in [65], we can express Data X-Ray's Diagnosis Cost as a difference metric: let $\epsilon = \dfrac{\mathcal{F}^R_{\text{attrs}}}{\mathcal{F}^R_{\text{attrs}} + \mathcal{F}^S_{\text{attrs}}}$, 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, 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 [67] 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 [67], 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 Section 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 [58] 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 data sets 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 MS-DBs, 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.

3.4 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 the *minimality* property—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 [46]), 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 $\mathcal{A}^*$. We illustrate how to leverage the anti-monotonicity of difference metrics for faster DIFF evaluation in Section 5.1.

3.5 ANTI DIFF: An extension 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 Figure 1. If ∇ denotes the ANTI DIFF operator, then we define the k-order ANTI DIFF of R and S to be:

$$_{\Gamma}\nabla_{\mathcal{A},k}(R, S) = \{r \in R \mid \pi_A(r) \notin \pi_A(\Delta_{\Gamma,\mathcal{A},k}(R, S))\} \quad (5)$$

For the scope of this paper, we limit our discussion of the ANTI DIFF query to this section, and leave any discussion on efficient implementations of it as future work.

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 two *logical* optimizations: the first is a predicate-pushdown-based adaptive algorithm for evaluating DIFF in conjunction with JOINs; it can provide up to $2\times$ speedup on real-world queries over normalized datasets. The second is a technique that leverages functional dependencies to accelerate DIFF query evaluation when possible; it can provide up to 20% speedups on datasets with one more functional dependencies.

Throughout this section—along with the subsequent section on physical optimizations—we focus on optimizations for DIFF that make the assumptions in Section 3.4. In addition, the optimizations discussed in Section 4.1 further assume that DIFF uses 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 [40]. 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, then subsequently evaluate the DIFF on the two intermediate outputs. This can be costly, however—the JOINs may be expensive to evaluate [1, 53, 55, 66], 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 [38]. 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

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: $V \leftarrow K \ltimes T$
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 $\Delta_{\Gamma=\{\theta_{\text{supp}},\theta_{\text{rr}}\},\mathcal{A},k}(R \bowtie V, S \bowtie V)$

DIFF 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.

4.2 Leveraging Functional Dependencies

As previously described, extensive data collection and augmentation is 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 functional dependency (FD) and is denoted $X \rightarrow Y$. Examples of commonly found FDs include location-based FDs (Zip Code $\rightarrow$ City) and FDs arising from user-defined functions or derived features (Raw Temperature $\rightarrow$ 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 left 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 $\rightarrow$ ISO Code, and ISO Code $\rightarrow$ Country. Naïvely running DIFF over this dataset may return results as in the right 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.

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 Section 3.4, 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 if 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.

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 becomes 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-aggregates 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, 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 [45, 64] 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, 71]. 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 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, particularly 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 tradeoff 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 if bitmaps speed up processing prior to mining itemsets from a set of columns. This is possible because our data originates 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 Figure 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 $c_1 ... c_N$, we should mine itemsets from those columns using bitmaps if the product $c_1 * c_2 ... * c_N < r$. Here, r is a parameter derived empirically from experiments similar to those in Figure 2. It is the ratio $\frac{t_{nb}}{t_b/(c_1 * c_2 ... c * N)}$ where t_{nb} and t_b are runtimes mining itemsets of order N from N columns with cardinalities $c_1 ... c_N$. 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×.

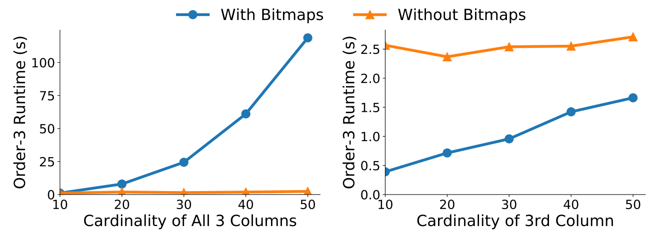

Figure 2: Bitmap vs. non-bitmap performance mining 3-order itemsets. Left: all columns share same cardinality. Right: two columns have fixed cardinality and the third varies.

6 Implementation

We implement the `DIFF` operator and the previously described optimizations in MB SQL, our extension to MacroBase. We develop both single-node and distributed implementations of our relational query engine; in this section, we focus on the distributed setting, and describe how we integrate the `DIFF` operator 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 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 Section 3.4. 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

To evaluate the scalability and generality of the DIFF operator, we implement DIFF in MB SQL (an extension of MacroBase) on a single node and in Apache Spark[1]. We evaluate the DIFF operator across a variety of datasets and queries in both of these settings.

7.1 Experimental Setup

Single-node benchmarks were run on an Intel Xeon E5-2690 v4 CPU with 512GB 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 26GB 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 (60GB, 175M rows, 13 columns) and MS-Telemetry B (26GB, 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: The publicly available[2] Censys [23] dataset (75GB, 400M rows, 17 columns), used in the Internet security community, consists of port scans across the Internet from two separate days, three months apart, where each record represents a distinct IP address. For our single-node experiments, we generate two smaller versions of this dataset: Censys A (3.6GB, 20M rows, 17 columns) and Censys B (2.6GB, 8M rows, 102 columns). These datasets are publicly available to researchers at . In our benchmarks, we evaluate DIFF queries comparing the port scans across the two days.

Center for Medicare Studies (CMS): The 7.7GB (15M row) Center for Medicare Studies dataset, which is publicly available[3], 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 two 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.

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

[2]https://support.censys.io/getting-started/research-access-to-censys-data

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

Table 2: Datasets used for our single-node benchmarks.

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

Table 3: Datasets used for our distributed benchmarks.

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

7.3.1 Generality

In this section, we benchmark DIFF against the core subroutines of three other explanation engines: Data X-Ray [65], RSExplain [58], and the original MacroBase [8], matching their semantics using DIFF as described in Section 3. We also compare DIFF against Apriori and FPGrowth from SPMF [27] as well as SQL-equivalent DIFF queries described in Section 3.2 on Postgres. Results are shown in Figure 3.

Original MacroBase We first compare the performance of DIFF to that of the original MacroBase [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 hours while MacroBase could not finish in 24 hours. 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[4]. 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 that of the cost calculations and so we obtain effectively matching results (our performance is 2% worse).

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 Section 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) columns with large numbers of unique values from each dataset. We found that the DIFF implementation was consistently between 8×-10× faster at calculating intervention than the originally described data-cube-based 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

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

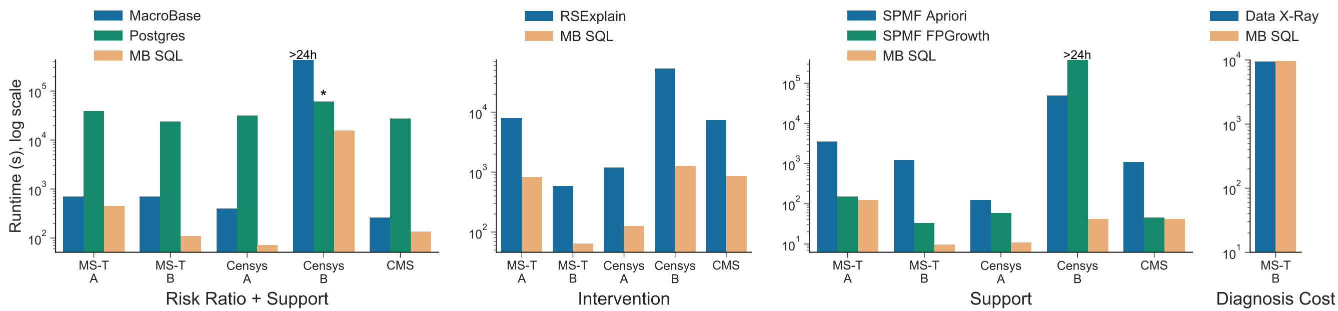

Figure 3: Runtime comparison of MB SQL performance on `DIFF` queries with various difference metrics against the equivalent query in other explanation engines. All queries were executed with `MAX ORDER 3`, except for the Postgres query on Censys B (denoted with an asterisk), which we ran with `ORDER 2` due to Postgres's limit on the number of `GROUP BY` columns. `>24h` indicates that the query did not complete in 24 hours.

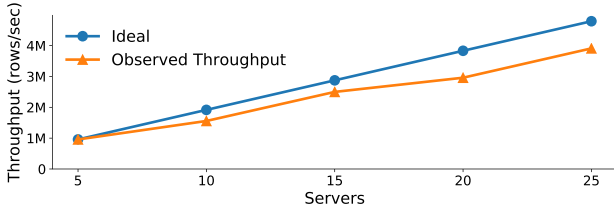

Figure 4: Distributed scalability analysis of MB SQL in Spark. On a public GCP cluster, we evaluate a `DIFF` query with support of 0.001 and risk ratio of 2.0 on `Censys`. We observe a near-linear scale-up as we increase the number of servers.

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 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 Section 5.

Postgres For these experiments, 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.

7.3.2 Distributed

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

Censys In Figure 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 three 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 4 cores.

At 50 executors, MB SQL in Spark evaluated our `DIFF` query in approximately 2,000 seconds. With fewer executors allocated, MB SQL's performance was slowed by significant memory pressure: more than 10% of the overall compute 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 1,000 seconds.

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 the physical optimizations described in Section 5 with the results shown in Figure 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 Section 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 Section 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. In total, our optimized implementation is 2.5-17× faster than our unoptimized implementation.

7.4.2 Distributed Factor Analysis

In Figure 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 [47] 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.

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.

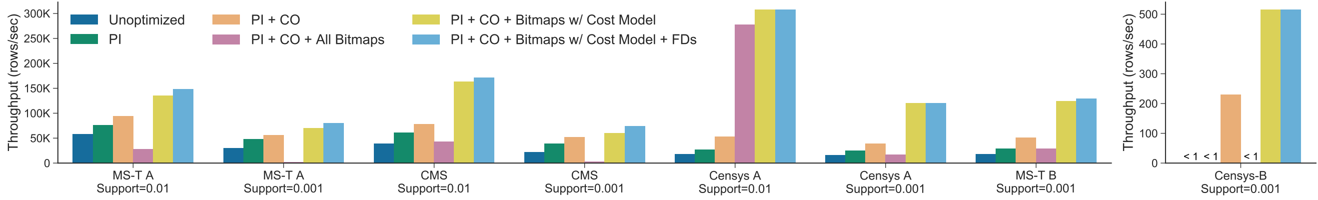

Figure 5: Factor analysis of our optimizations, conducted on a single machine and core. We successively add all optimizations (PI: Packed Integers, CO: Column Ordered, FDs: Functional Dependencies) discussed in Sections 4 and 5.

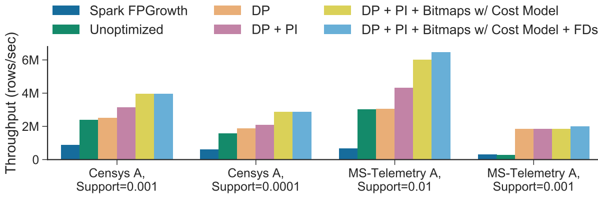

Figure 6: Factor analysis of distributed optimizations, conducted on 25 four-core machines. We successively add all optimizations (DP: Distributed Pruning, PI: Packed Integers, FDs: Functional Dependencies) discussed in Sections 4, 5, and 6 and report corresponding throughput. We also compare against Spark's FPGrowth.

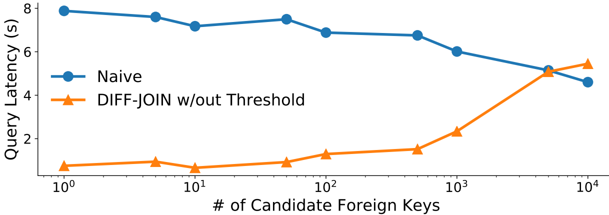

Figure 7: Runtime of the DIFF-JOIN predicate pushdown algorithm with *threshold* disabled vs. the naïve approach as $|K|$ is varied. $|R|$ = $|S|$ = 1M rows, and $|T|$ = 100K rows with 4 columns. The DIFF query is run with risk ratio = 10.0 and support = 10^{-4}.

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 $2\times$.

In Figure 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 4 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.

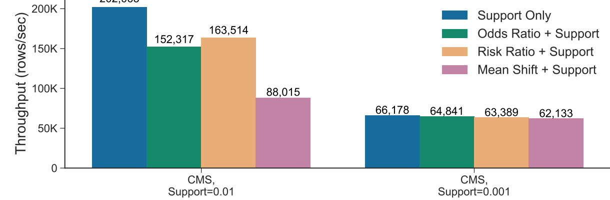

Figure 8: Throughput of different difference metrics on CMS, with a fixed ratio of 2.0 (except for Support Only).

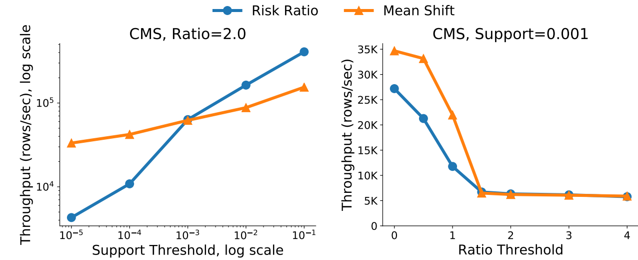

Figure 9: Throughput of two DIFF queries with varying support (left) and ratio (right) thresholds.

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 Figure 8. 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 Figure 9. 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 Figure 9, we find that decreasing either the risk ratio or the mean shift ratio decreases runtime. This is attributable to the minimality rule in Section 3.4. 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 ratios, fewer itemsets are pruned by minimality, so more must be considered.

8 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 the DIFF operator 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. [58] developed a formal approach for explanation queries over multiple relations, but they require computing an entire data cube to evaluate queries. Wu and Madden [67] 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 [39,57,69] or diagnose data errors [65]; 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 [31,35,41,48,60]. Various feature selection techniques, such as compressive sensing [12], correlation-based tests [32], and tree-based approaches [56] 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 [59] 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 [46]; 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 [6,15,26,71]. This previous work, however, mostly considers lists of transactions instead of relational tables and thus has no notion of column cardinality; we show in Section 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 [63], 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 [64], dictionary encoding [50], and bitmap indexes [17,54]. 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 [68] and Impala [14].

In addition, our proposed optimizations draw from research in adaptive query processing [5,7,21]. We show in Section 4 how to optimize DIFF-JOIN queries using our adaptive algorithm, which builds upon extensive work on optimizing JOINs [52,53,55,61,62]. Our algorithm also shares similarity with recent work examining the cost of materializing JOINs in machine learning workloads [19,44], including learning over JOINs [42]. 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 [37], 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.

9 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. Because the DIFF query is semantically equivalent to a standard relational query composed of UNION, GROUP BY and CUBE operators, it integrates 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, we are able to scale to large industrial workloads across Microsoft and Facebook. We are continuing to develop the DIFF operator with our collaborators, including Microsoft, Facebook, Censys, and Google, and hope to provide additional improvements to further boost data analyst productivity.

10 Acknowledgments

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.

11 References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases: the logical level*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[2] R. Agarwal, R. Srikant, et al. Fast algorithms for mining association rules. In *VLDB*, pages 487–499, 1994.

[3] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou. Understanding the mirai botnet. In *USENIX Security*, 2017.

[4] M. Armbrust et al. Spark sql: Relational data processing in spark. In *SIGMOD*, pages 1383–1394. ACM, 2015.

[5] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD*, volume 29, pages 261–272. ACM, 2000.

[6] J. Ayres et al. Sequential pattern mining using a bitmap representation. In *KDD*, pages 429–435. ACM, 2002.

[7] S. Babu, P. Bizarro, and D. DeWitt. Proactive re-optimization. In *SIGMOD*, pages 107–118. ACM, 2005.

[8] P. Bailis et al. Macrobase: Prioritizing attention in fast data. In *SIGMOD*, pages 541–556. ACM, 2017.

[9] P. Bailis et al. Prioritizing attention in fast data: Principles and promise. *CIDR Google Scholar*, 2017.

[10] E. Baralis, T. Cerquitelli, and S. Chiusano. Index support for frequent itemset mining in a relational dbms. In *ICDE*, pages 754–765. IEEE, 2005.

[11] E. Baralis, T. Cerquitelli, and S. Chiusano. Imine: Index support for item set mining. *IEEE Transactions on Knowledge and Data Engineering*, 21(4):493–506, 2009.

[12] R. G. Baraniuk. Compressive sensing [lecture notes]. *IEEE signal processing magazine*, 24(4):118–121, 2007.

[13] Y. Benjamini and D. Yekutieli. The control of the false discovery rate in multiple testing under dependency. *Annals of statistics*, pages 1165–1188, 2001.

[14] M. Bittorf et al. Impala: A modern, open-source sql engine for hadoop. In *CIDR*, 2015.

[15] D. Burdick, M. Calimlim, and J. Gehrke. Mafia: A maximal frequent itemset algorithm for transactional databases. In *ICDE*, pages 443–452. IEEE, 2001.

[16] S. Chambi et al. Better bitmap performance with roaring bitmaps. *Software: practice and experience*, 46(5):709–719, 2016.

[17] S. Chambi et al. Optimizing druid with roaring bitmaps. In *IDEAS*, pages 77–86. ACM, 2016.

[18] S. Chaudhuri. An overview of query optimization in relational systems. In *PODS*, pages 34–43. ACM, 1998.

[19] L. Chen et al. Towards linear algebra over normalized data. *PVLDB*, 10(11):1214–1225, 2017.

[20] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56:74–80, 2013.

[21] A. Deshpande et al. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.

[22] Z. Durumeric et al. The matter of heartbleed. In *IMC*, pages 475–488. ACM, 2014.

[23] Z. Durumeric et al. A search engine backed by Internet-wide scanning. In *SIGSAC*, pages 542–553. ACM, 2015.

[24] R. Fagin et al. Efficient implementation of large-scale multi-structural databases. In *VLDB*, pages 958–969. VLDB Endowment, 2005.

[25] R. Fagin et al. Multi-structural databases. In *PODS*, pages 184–195. ACM, 2005.

[26] W. Fang et al. Frequent itemset mining on graphics processors. In *DaMoN*, pages 34–42. ACM, 2009.

[27] P. Fournier-Viger et al. The spmf open-source data mining library version 2. In *Joint European conference on machine learning and knowledge discovery in databases*, pages 36–40. Springer, 2016.

[28] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE*, pages 209–218. IEEE, 1993.

[29] J. Gray et al. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data mining and knowledge discovery*, 1(1):29–53, 1997.

[30] A. Greenberg et al. The cost of a cloud: research problems in data center networks. *ACM SIGCOMM computer communication review*, 39(1):68–73, 2008.

[31] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *Journal of machine learning research*, 3(Mar):1157–1182, 2003.

[32] M. A. Hall. Correlation-based feature selection of discrete and numeric class machine learning. 2000.

[33] J. M. Hellerstein et al. Architecture of a database system. *Foundations and Trends® in Databases*, 1(2):141–259, 2007.

[34] J. M. Hellerstein and M. Stonebraker. *Readings in database systems*. 2005.

[35] S. C. Hoi et al. Online feature selection for mining big data. In *BigMine*, pages 93–100. ACM, 2012.

[36] IDC. The digital universe of opportunities: Rich data and the increasing value of the internet of things, 2014. `http://www.emc.com/leadership/digital-universe/`.

[37] I. F. Ilyas et al. Cords: automatic discovery of correlations and soft functional dependencies. In *SIGMOD*, pages 647–658. ACM, 2004.

[38] Y. E. Ioannidis and S. Christodoulakis. *On the propagation of errors in the size of join results*, volume 20. ACM, 1991.

[39] N. Khoussainova, M. Balazinska, and D. Suciu. Perfxplain: Debugging mapreduce job performance. *PVLDB*, 5(7):598–609, 2012.

[40] R. Kimball and M. Ross. *The data warehouse toolkit: the complete guide to dimensional modeling*. John Wiley & Sons, 2011.

[41] P. Konda et al. Feature selection in enterprise analytics: a demonstration using an r-based data analytics system. *PVLDB*, 6(12):1306–1309, 2013.

[42] A. Kumar. *Learning Over Joins*. PhD thesis, The University of Wisconsin-Madison, 2016.

[43] A. Kumar et al. To join or not to join?: Thinking twice about joins before feature selection. In *SIGMOD*, pages 19–34. ACM, 2016.

[44] A. Kumar, J. Naughton, and J. M. Patel. Learning generalized linear models over normalized data. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1969–1984. ACM, 2015.

[45] A. Lamb et al. The vertica analytic database: C-store 7 years later. *VLDB*, 5(12):1790–1801, 2012.

[46] J. Leskovec et al. *Mining of Massive Datasets*. Cambridge university press, 2014.

[47] H. Li et al. Pfp: parallel fp-growth for query recommendation. In *RecSys*, pages 107–114. ACM, 2008.

[48] J. Li et al. Feature selection: A data perspective. *ACM Computing Surveys (CSUR)*, 50(6):94, 2017.

[49] A. Meliou, S. Roy, and D. Suciu. Causality and explanations in databases. *PVLDB*, 7(13):1715–1716, 2014.

[50] S. Melnik et al. Dremel: interactive analysis of web-scale datasets. *PVLDB*, 3(1-2):330–339, 2010.

[51] X. Meng et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[52] T. Neumann and B. Radke. Adaptive optimization of very large join queries. 2018.

[53] H. Q. Ngo et al. Worst-case optimal join algorithms. *Journal of the ACM (JACM)*, 65(3):16, 2018.

[54] P. O'Neil and D. Quass. Improved query performance with variant indexes. In *SIGMOD*, volume 26, pages 38–49. ACM, 1997.

[55] A. Pagh and R. Pagh. Scalable computation of acyclic joins. In *PODS*, pages 225–232. ACM, 2006.

[56] E. Rounds. A combined nonparametric approach to feature selection and binary decision tree design. *Pattern Recognition*, 12(5):313–317, 1980.

[57] S. Roy et al. Perfaugur: Robust diagnostics for performance anomalies in cloud services. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 1167–1178. IEEE, 2015.

[58] S. Roy and D. Suciu. A formal approach to finding explanations for database queries. In *SIGMOD*, pages 1579–1590. ACM, 2014.

[59] G. Rupert Jr et al. *Simultaneous statistical inference*. Springer Science & Business Media, 2012.

[60] Y. Saeys, I. Inza, and P. Larrañaga. A review of feature selection techniques in bioinformatics. *bioinformatics*, 23(19):2507–2517, 2007.

[61] S. Schuh, X. Chen, and J. Dittrich. An experimental comparison of thirteen relational equi-joins in main memory. In *SIGMOD*, pages 1961–1976. ACM, 2016.

[62] P. G. Selinger et al. Access path selection in a relational database management system. In *Readings in Artificial Intelligence and Databases*, pages 511–522. Elsevier, 1988.

[63] X. Shang, K.-U. Sattler, and I. Geist. Sql based frequent pattern mining with fp-growth. In *Applications of Declarative Programming and Knowledge Management*, pages 32–46. Springer, 2005.

[64] M. Stonebraker et al. C-store: a column-oriented dbms. In *VLDB*, pages 553–564. VLDB Endowment, 2005.

[65] X. Wang et al. Data x-ray: A diagnostic tool for data errors. In *SIGMOD*, pages 1231–1245. ACM, 2015.

[66] D. E. Willard. Applications of range query theory to relational data base join and selection operations. *journal of computer and system sciences*, 52(1):157–169, 1996.

[67] E. Wu and S. Madden. Scorpion: Explaining away outliers in aggregate queries. *PVLDB*, 6(8):553–564, 2013.

[68] F. Yang et al. Druid: A real-time analytical data store. In *SIGMOD*, pages 157–168. ACM, 2014.

[69] D. Y. Yoon, N. Niu, and B. Mozafari. Dbsherlock: A performance diagnostic tool for transactional databases. In *SIGMOD*, pages 1599–1614. ACM, 2016.

[70] M. Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 2–2. USENIX Association, 2012.

[71] F. Zhang, Y. Zhang, and J. Bakos. Gpapriori: Gpu-accelerated frequent itemset mining. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 590–594. IEEE, 2011.