FaDE: More Than a Million What-ifs Per Second

Haneen Mohammed
Columbia University
ham2156@columbia.edu

Alexander Yao
Columbia University
awy2108@columbia.edu

Charlie Summers
Columbia University
cgs2161@columbia.edu

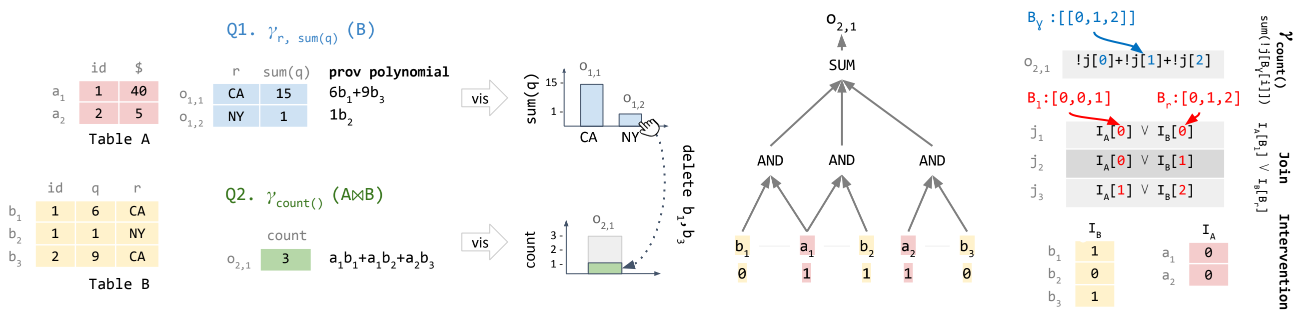
Hongbin Zhong
Georgia Tech
rj986215159@outlook.com

Gromit Yeuk-Yin Chan
Adobe Research
ychan@adobe.com

Subrata Mitra
Adobe Research
sumitra@adobe.com

Lampros Flokas
Celonis Inc.
l.flokas@celonis.de

Eugene Wu
Columbia University
ewu@cs.columbia.edu



(a) Input relations (b) Query outputs & provenance polynomial annotations (c) Intervention-based application (d) Circuit representation of Q2 provenance polynomials (e) FaDE generates vectorized intervention logic for Q2

Figure 1: Provenance-based intervention for a cross-filter visualization. (a) Input relations are annotated with ids, and (b) query results are annotated with provenance polynomials that describe how tuples are joined (×) and aggregated (+). (c) In the visualized results, clicking on NY updates the lower chart by re-running Q_2 without b_1,b_3. (d) Existing methods set the existence bits $b_1 = b_3 = 0$ in the provenance circuit and recursively compute the circuit—often slower than re-running the query. (e) FaDE models provenance as dense arrays (B_l, B_r, B_γ) and interventions (I_A, I_B) as bit vectors over operator inputs where 1 means intervene (e.g., delete) rather than 'exist'. Join uses provenance to $\vee$ the appropriate interventions ($I_A[B_l] \vee I_B[B_r]$), and group-by counts the non-intervened join results. On TPC-H SF=1, FaDE evaluates $>1M$ interventions per second.

ABSTRACT

What-if queries are the building blocks for many explanation and analytics applications—sensitivity analysis, hypothetical reasoning, data cleaning, probabilistic databases—that explore how a query's output changes due to input data changes. Their response time is bounded by intervention evaluation latency, which can be in the minute or hours for complex queries and large datasets. FaDE is a compilation engine that uses provenance to evaluate hypothetical deletion and scaling interventions at low latency and high throughput. FaDE forgoes conventional provenance representations as symbolic expressions and leverages their underlying relational structure. This accelerates intervention evaluation on average by 1000× against IVM and 10,000× against prior provenance-based approaches. In addition, FaDE develops a suite of optimizations (e.g., compilation, parallelization, incremental evaluation, sparse representations) that collectively raise evaluation throughput to >1 million interventions per sec—a rate that can brute-force existing applications within $1s$.

PVLDB Reference Format:
Haneen Mohammed, Alexander Yao, Charlie Summers, Hongbin Zhong, Gromit Yeuk-Yin Chan, Subrata Mitra, Lampros Flokas, and Eugene Wu. FaDE: More Than a Million What-ifs Per Second. PVLDB, 18(4): 943 - 955, 2024.
doi:10.14778/3717755.3717757

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/cudbg/whatifs.

1 INTRODUCTION

What-if queries are the building block for a wide range of applications, such as sensitivity analysis, deletion-based query explanations, data cleaning, debugging, interactive visualization, data integration, probabilistic databases, and query-by-example [4, 10, 13, 14, 20, 24, 27, 30, 33, 36]. These applications measure changes to a query $Q(D)$'s output when deleting subsets of the query's input relations (*"deletion interventions"*) or transforming attribute values

Proceedings of the VLDB Endowment, Vol. 18, No. 4 ISSN 2150-8097.
doi:10.14778/3717755.3717757

used to compute aggregated metrics (*"scaling interventions"*). They primarily differ in the type and set of interventions to evaluate, and their main bottleneck is the cost to quickly re-evaluate Q under each intervention. Let us examine three exemplar applications:

- **What-if Analysis and Hypothetical Reasoning** evaluates user-specified interventions on Q. For instance, how would profits change if click rates increased by 80% in all California cities?
- **Explanation Engines & Why Analysis** help explain unexpected trends in an aggregation query $Q(D)$'s results. A common form of explanation [4, 33, 36] identifies a conjunctive predicate p such that removing the input that matches the predicate $Q(\sigma_{\neg p}D)$ has a desired result on the query output. To do so, explanation engines search the space of all N^{th}-order equality predicates for those that maximize a loss function over $Q(\sigma_{\neg p}D)$.
- **Possible Worlds and Incomplete Databases** Conditional tables (c-tables) [23] encode incomplete databases where tuples are annotated with propositional formulas over random variables. Probabilistic databases extend c-tables with a probability space over random variable assignments, and a query output's probability can be estimated using monte-carlo methods by re-evaluating Q over sampled database instances.

Many applications require searching over a large space of candidate interventions—tens of thousands or even millions—and are bound by the throughput of intervention evaluation. General approaches such as incremental view maintenance (IVM) are designed for arbitrary database interventions and suffer from large memory overheads and slow evaluation when the intervention is large [5].

Thus, prior applications identify problem-specific assumptions to heuristically prune the search space and/or perform efficient precomputation. For instance, Caravan [14, 16] uses apriori knowledge of the desired hypothetical scenarios (parameterized interventions) and pre-computes provenance circuits that include special variables that turn on/off combinations of hypotheticals.

Explanation engines suffer from similar constraints: Explanation-Ready Databases (ERDB) [33] requires the developer to pre-define explanation templates in order to pre-compute tables to accelerate IVM, Scorpion [36] is limited to incrementally updatable aggregation functions such as `SUM` and `COUNT`, and DIFF [3, 4] heuristically uses minimum support (number of tuples that match the predicate) to prune the search space. Even ignoring the offline costs, they still take seconds or minutes to run.

What if it is possible to evaluate interventions at a sufficiently high throughput that such heuristics and assumptions are not necessary? For instance, monte-carlo methods for probabilistic queries typically draw 100-1000 samples [25]. Similarly, given a set of 10 attributes each with 50 unique values, there are only 112,000 second-order conjunctive predicates to evaluate. A system that can evaluate $1M$ interventions per second would greatly accelerate existing search heuristics, support ad-hoc questions, and solve many practical problems within a second using brute force.

The Promise of Provenance. A promising method that matches the needs of these applications is to use data provenance to accelerate query re-evaluation. Most data provenance systems [21, 34] today are built on the provenance semiring formalism. Each output tuple is annotated with a symbolic polynomial expression that describes how input tuples (variables) were joined (×) or unioned (+) during query processing. Re-evaluating the query result under different interventions is equivalent to re-evaluating each polynomial under different variable assignments for Select-Project-Join-Aggregate-Union (SPJAU) queries .

Example 1. *Output tuples in Figure 1(b) are annotated with a provenance polynomial $o_{j,i}$ that records how the output tuple was computed. Setting $b_2 = 0$ deletes B's second tuple, which updates $o_{1,1} = 1$ and $o_{1,2} = 0$. This indicates that only $o_{1,1}$ remains. Similarly, setting $b_3 = 0$ updates the output aggregates of $o_{1,1}$ to 6.*

In theory, provenance-based re-execution addresses the drawbacks of IVM because 1) it performs strictly less work since all data dependencies and join matches are known apriori, 2) it does not need to maintain large intermediate relations and reduces memory pressure, and 3) the re-execution cost is independent of the intervention size. In practice, ProvSQL [34] is the only system that implements provenance-based IVM for deletion and scaling interventions. Unfortunately, it is slower than general IVM systems like DBToaster [5] and even re-running the query from scratch due to recursive evaluation of the circuit-based representation. This makes non-sequential memory accesses, which wastes memory bandwidth and CPU cache. Each node also executes a different operation, which reduces code locality.

Example 2. *Figure 1(d) depicts the standard circuit-based representation [15] of the provenance polynomials for $o_{2,1}$ The circuit is stored as a directed acyclic graph (DAG) and evaluated by recursively traversing it in a top down fashion.*

Our key insight is that every output tuple shares the same circuit structure—all circuit nodes that correspond to the same logical operator apply the same logical/arithmetic operations and access the same data. We can improve instruction and data locality by representing provenance and re-executing the polynomials on a per-operator basis. This can be analogized as the difference between row- and column-oriented query execution.

Example 3. *Figure 1(e) depicts per-operator representations for the join and group-by operators of Q2, along with the intervention evaluation code. In contrast to the circuit representation, the provenance is represented as integer arrays: $B_l[i]$ ($B_r[i]$) specifies the input offset in A (B) for the i^{th} join result, and $B_\gamma[i]$ specifies the indexes of the join results that contribute to the i^{th} group (and consequently query output $o_{2,1}$). FaDE takes as input interventions I_A, I_B as bit masks over the input relations, − 1 to interven (delete/scale).*

In this work we propose FaDE, which builds on recent advances in provenance-capturing DBMSes to evaluate what-if queries at low-latency and high-throughput by using provenance-based intervention evaluation. Recent DBMSes like SmokedDuck [27] are able to capture per-operator provenance as dense 2D (for group-by) or 1D (other operators) integer arrays during query execution with very low runtime overhead—as little as 0% for some TPC-H queries.

FaDE observes that these provenance arrays, when used for intervention evaluation, are amenable to fast scans, parallelizaton, and hardware vectorization. FaDE first translates the initial query Q's execution plan into a corresponding intervention plan composed of pre-defined operators as well as operators generated and compiled at run time. Given matrices that encode the desired interventions, the intervention plan quickly evaluates them using tight loops.

Although the high level intuition is simple, achieving high throughput for practical applications is difficult—a straight-forward implementation processes a mere 20 interventions/second. This is because every step—both intervention generation and evaluation—of the process must be high throughput in order to avoid bottlenecks. This leads to several interlocking challenges.

First, fast intervention evaluation is sensitive to the provenance representation as it impacts hardware prefetching, caching, and memory utilization. For instance, using backward provenance for group-by requires inefficient scattered memory accesses over the input tuples. In addition, fast provenance capture techniques evaluate operators bottom-up during query execution, and can over-generate provenance for intermediates that do not contribute to any output tuples [27, 31]. This directly increases the amount of wasted work during intervention evaluation.

Second, intervention representation is important because dense bit-matrix representation is quadratic in size with respect to the input database size and number of interventions. For example, 100,000 interventions on an input table with 1M records would be 12.5GB. A more efficient representation is necessary to avoid materializing and processing over gigabytes of interventions for applications like explanation engines, which search large intervention spaces.

Third, the appropriate intervention implementations, as well as the appropriate level of parallelization, batching, and vectorization, vary greatly depending on the base query, the set of interventions, available resources, and the complexity of the group-by aggregation functions. There are too many factors for a single approach to adequately support, and an effective automated approach is needed to choose the best intervention plan on a per-query basis.
In summary, we contribute:

- FaDE, a database engine that supports fast what-if query evaluation for ad-hoc SPJAU queries with nested aggregates, and exposes a general and expressive what-if API that subsumes most prior specialized what-if systems
- A suite of effective optimizations including: software and hardware data parallelism, such as multi-threading, batched execution, and SIMD-vectorization; incremental intervention evaluation when the aggregation function is expressible as a ring; a sparse representation for mutually exclusive intervention sets, such as all conjunctive equality predicates of a fixed arity; and efficient provenance pruning to reduce the space and runtime complexity of provenance-based intervention evaluation.
- Extensive experimental comparisons between FaDE, the IVM engine DBToaster, and provenance based engine ProvSQL. In relative terms, FaDE is on average 1,000× and 10,000× faster than DBToaster and ProvSQL, respectively. In absolute terms, FaDE can interactively evaluate hundreds of thousands of interventions on multi-join queries in <100ms.

2 MOTIVATING USE CASE AND API

This section describes a data-driven use case [18] that uses sensitivity analysis, what-if, and how-to functionality as well as two other use cases that FaDE can accelerate. We include code snippets in the use case, and then formally introduce FaDE's API and semantics.

2.1 Analytics Use Case

Mona is a data scientist studying customer churn rates at her company. She runs an initial query and plots the result as a line chart:

```
SELECT EXTRACT(MONTH FROM date) AS month,
       LinearRegressionUDF(churn, clicks) AS slope
FROM sales JOIN custs USING (cid)
WHERE date >= CURRENT_DATE - INTERVAL '6 months'
GROUP BY EXTRACT(MONTH FROM date) ORDER BY month
```

(U1) What-if Analysis. Mona has a \$30K budget and from past experience, knows that adding \$5K to an email campaign in a city should increase the email click rate by 10% for that city's customers. She thus asks: *What if we increase email click rates by 80% in all California cities?* This is expressed as:

```
O.whatif({'custs.click': 1.8, 'where':"custs.state='CA'"})
```

(U2) Sensitivity Analysis. Mona notices that the churn rate has steadily risen in the past two months. She wants to understand this rise, so asks the system to identify data slices that the rise is most sensitive to. She suspects it may be related to the state of sale and age of the customers, and performs sensitivity analysis that deletes every combination of `state` and `age` (e.g., `state=CA∧age=20s`) and finds those that minimize the churn rate. To answer the question *Which combination of state and age that if removed, minimizes the churn rate the most?*, she first defines a metric that computes the churn rate in the post-intervention query result and then calls FaDE to return the top-3 state, age combinations that minimize the metric:

```
metric = lambda pre,post:abs(post["churn"])
O.whatif({'where':"custs.state=? and custs.age=?"},
        {'k':3, 'metric':metric,  'objective':"minimize"})
```

(U3) How-to Analysis. Mona then submits a How-to question to identify which subset of Californian cities she should target, assuming she increases their click rate by 30% each. Formally, she asks *Which city if we increase their click rates by 30% would minimize the churn rate the most?*, and executes:

```
O.whatif({'custs.click':1.3, 'where':"custs.city=?"},
         {'k':3, 'metric':metric, 'objective':'minimize'})
```

Focusing now on Palm Springs, she submits a second How-to question to understand how many resources to put into her campaign. Specifically, for different increases in the email click rate, how much would the churn reduce and is the marginal reduction worth the increased budget? This is expressed as *How much should we increase click rates for customers in Palm Springs city that would minimize the churn rate the most?*. She searches between 0 to 100% increase in click rates:

```
O.whatif({'custs.click': range(1, 2, 0.1),
          'where':"city='Palm Springs'"},
         {'k':3, 'metric':metric, 'objective':'minimize'})
```

To summarize, the above use case switches between several related tasks:

(1) Deletion Intervention: removing one or more subsets of input (U2)
(2) Scaling Intervention: scale the aggregated attribute of the subset of input tuples that match one (U1) or multiple (U3) predicates.
(3) Ranking interventions based on a custom metric over the intervened query result and return the top-K answers (U2, U3).

2.2 Additional Use Cases

In addition to extending data analysis with interactive sensitivity, what-if, and how-to analysis, FaDE supports many other use cases.

Interactive Cross-Filtering. Prior work [29] aims to interactively update cross filtering based visualizations. In a typical cross filtering setup, users highlight data of interest in one view and the results of another view update to consider only the selected subset. While [29] already makes use of provenance metadata to identify the selected data in one view, it could also use FaDE instead of IVM to update the results of the other view.

For example, if a user selects state 'CA' in a map visualization, a linked histogram visualization is updated by a hypothetical delete request to remove all other influence from tuples that are not in 'CA' state.

```
O.whatif({'where':"cust.state<>'CA'"})
```

FaDE can further accelerate the interactions further by prefetching the results for multiple states at once:

```
O.whatif({'where':"custs.state<>?"})
```

Probabilistic Databases. Probabilistic databases [12] annotate each tuple with the probability that it exists in the database. Calculating probabilistic query results evaluates the provenance polynomial over input tuple probabilities. Since this must be done for every possible output row, query evaluation is #*P*-complete. Monte Carlo methods were proposed in [12] to estimate output tuple probabilities by sampling database instances from the probability distribution and averaging the query results over the samples. FaDE can express this by specifying the target list as a random sample whose tuple probabilities are given by attribute prob in the table.

```
O.whatif({'where': { 'B': {p: 'B.prob', n: 1000}}})
```

2.3 Fade Python API

The FaDE Python API extends a query's result cursor with a general whatif() method that specifies the set desired intervention(s), optional semantics if there are interventions over multiple tables, and an optional objective to rank the interventions and return the top k:

```
O = con.execute(Q)
O' = O.whatif(intervention, operation, objective)
```

2.3.1 Interventions and Operation. The core component of deletion and scaling interventions is the whereclause, which specifies the subsets of the query's input tuples to intervene. Deletion interventions are fully described by the whereclause, while scaling interventions also specify how attribute(s) are scaled:

```
whereclause       // deletion
{ [attrname: scalefactors]+,  whereclause }   // scaling
```

Below, we introduce whereclause, scaling interventions, and then the semantics of interventions on multiple tables.

The whereclause is the core component of an intervention, which specifies the subsets of the query's input tuples to intervene upon. The whereclause maps an input relation R to a *Target Matrix* $M = [S_1, \ldots, S_n]$ that encodes a batch of n subsets of R, where $S_i \subseteq R$:

```
{ [relationname: targetmatrix]+ }   // whereclause
```

By default, M is a dense numpy array. S_i is bitmask over R, where 1 deletes/scales and 0 preserves the corresponding tuple. Although flexible, materializing the target matrix and passing it to FaDE is impractical when R and batch size are large. For instance, 1M tuples and a batch size of 100K would require 12.5GB. Thus, we introduce two declarative target matrix specifications for common use cases:

(1) Predicates: Notice that each bitmask in the target matrix is logically a predicate over the relation. Thus, we express the target matrix with a list of predicate strings, where each string represents one or more predicates that are logically concatenated into the target matrix. Each predicate string is of the form `attr op v` where `attr` is a fully qualified attribute, `op` is a comparison operator, and `v` is a constant. A common application is to search a space of conjunctive equality predicates, so for equality predicates, the constant can be replaced with a parameter ?, which binds all domain values. For instance, `A=? and B=? and C<1` expresses all combinations of A and B values.
(2) Random Samples: given a list p of tuple probabilities (or uniform if not specified), we generate n samples where each sample draws tuple i with probability $p[i]$. For instance, `{p: [0.5, 0.5], n: 10}` generates 10 samples, each sample includes both tuples with uniform probability. p may also be specified as an attribute in the sampled relation.

Scaling Interventions are specified by mapping fully qualified attribute names to a scaling factor sf. The input attributes marked 1 in the target matrix are scaled by sf, and the rest retain their original value. As a convenience, users can supply a list of scaling factors and FaDE evaluates each one. All references to the scaled attribute in the query plan will use the scaled values instead. We restrict attributes to those that are not part of any control flow decision (e.g. filter or join conditions) to ensure new tuples are not added to the query result through attribute scaling/updates. For instance, the following scales the first and third rows in B in two ways: decreasing them by 50% and doubling their values.

```
{ 'B.q': [-0.5, 1.0], 'where':{'B':[[1,0,1]]} }
```

Multi-table Semantics. If the whereclause specifies target matrices for multiple relations, either explicitly or using a string predicate (e.g., `A.a=1 and B.b=1`), then we assume that each matrix has the same number of columns (interventions). For deletion interventions, each intervention is logically a predicate over a base relation, but how to combine them across multiple relations is ambiguous. There are two semantics that the user can choose from using the operation argument. OR semantics simulates deleting tuples from the base relations: if any tuple is deleted, *all* derives tuples are deleted as well. In contrast, AND semantics simulates a conjunctive predicate that spans the base relations, where an intermediate tuple is deleted only if *all* participating tuples are deleted.

A wrinkle arises when the scaled attribute and whereclause reference disjoin tables (e.g., scale A.$ where B.q>1). It is not obvious which tuples in A should be scaled prior to joining A and B. Given the restriction that A.$ (and any derivatives) cannot be used for control flow, we rewrite the base query plan before it is executed into a canonical form by pushing all projection expressions and group-by aggregations that are not involved in control flow above the joins. The tuples to scale are thus defined by the target matrix after the join results. Note that projections can further be removed by inlining their expressions into the aggregation functions that reference them. Canonicalization simplifies the query plan by eliminating projections that might involve scaled attributes.

2.3.2 Optimization Objective. Many applications wish to find "the best" interventions given a search space. To support this use case, the user can specify a metric to optimize, whether to minimize or maximize the metric, and a top K value. This is primarily to avoid the communication costs of sending all intervention results to the client, and to avoid the need to materialize result values for all interventions.

A metric is a lambda function given by the user that is evaluated for each intervention. It takes the original and updated query results $pre = Q(D)$ and $post = Q(I(D))$, respectively, as dataframes indexed by the column names and returns a numerical score value per intervention. For instance, the following measures the total difference between half the original query's churn rates and the churn rates of the intervened queries:

```
metric = lambda pre,post:sum(pre["churn"]/2-post["churn"])
```

3 BACKGROUND

We now introduce provenance and its use for view maintenance, existing limitatons, and the motivation behind FaDE.

3.1 Provenance and View Maintenance

Provenance polynomials is a general model based on semi-ring annotations [19]. Each output tuple is annotated with a symbolic polynomial expression that represents its derivation tree: each variable represents an input tuple, and operators union (+) or join (×) tuples together (see Figure 1 for an example). K-semimodules [6] extend this framework to support aggregation by also logging the expression values passed into the aggregation function (e.g., the values of $B.q$ in Figure 1(a)).

Provenance polynomials are sufficient for view maintenance under deletions for monotonic queries including SPJAU queries [6, 19] because deletions do not generate new tuples. Provenance polynomials express deletion interventions by setting the variables of the deleted tuples to the semiring zero element (e.g., *false* for normal relational algebra). The logic is very simple: σ and Union deletes its output tuple if its contributing tuple is deleted, $\bowtie$ if either contributing tuple is deleted, and γ if the entire group is deleted. K-semimodules support attribute scaling as long as 1) the values of the variables referenced in the aggregation expression are logged, and 2) the scaled attributes do not affect the query's control flow in a way that introduces new tuples under intervention (e.g., not used in filter, grouping, join conditions). Once the aggregate's input values have been updated, it simply re-evaluates itself over the non-deleted inputs.

In theory, provenance-based view maintenance should be competitive or more efficient than IVM because it skips filters and joins re-execution, does not need to build nor maintain hash tables, and does not perform work that does not contribute to tuples not in the output.

3.2 Limitations of Existing Approaches

The dominant ways to evaluate deletion and scaling interventions are based on provenance circuits [14, 15, 34] and Provisioning Autonomous Representation (PAR) [14]. We now describe their limitations.

3.2.1 Provenance Circuits. All modern provenance management systems capture provenance as circuits (e.g., Figure 1(d)). A provenance circuit is a directed acyclic graph (DAG) where the leaves are tuple variables, and interior nodes are operators (semiring + or ×, or an aggregation function). An operator takes its children as input and routes its output to its parents. During query execution, each operator computes each output tuple's provenance circuit from the circuits of its inputs.

To the best of our knowledge, ProvSQL [34] is the only publicly available DBMS that supports provenance-based interventions. It maps tuple variables to semiring elements, and evaluates the circuit top-down using user defined types and functions (e.g., Figure 1(d)). For instance, deletion interventions would use the boolean semiring and map (×, +) to (&, |).

Circuits benefit from simplicity. The representation is independent of the query plan and how provenance was tracked, which also simplifies the evaluation logic. Unfortunately, operator independence comes at a deep performance cost due to poor instruction and data locality. Circuit evaluation relies heavily on branching and pointer-based accesses to find the next instruction, which prevents efficient out-of-order execution and leads to non-sequential memory accesses. These access patterns make poor use of CPU caches and memory bandwidth.

Provisioning Autonomous Representations and followup work [8, 14, 16] are similar to ERDB [33] in that they pre-compute data structures—specifically provenance circuits—to accelerate pre-registered classes of deletion and scaling intervention such as those supported by FaDE. Theoretically, the general approach requires provenance size that is exponential in the number of hypothetical scenarios [8], and the main techniques are to reduce the granularity of interventions (e.g., allow users to delete at year rather than month granularity). However, this still requires the expensive overhead of capturing provenance circuits, compressing them, and then evaluating the circuits, all of which take between seconds to hours on TPC-H SF=10. Ultimately, this approach is neither fast nor suitable for ad-hoc analysis.

3.3 The Opportunity

Provenance circuits are stored on a per-tuple basis, and incur considerable overhead when used for intervention evaluation. However, unlike general circuits, provenance circuits have structured access patterns that are amenable to a more efficient per-operator representation and execution, analogous to the difference between row and columnar query execution. In addition, recent provenance systems like Smoke [31] and SmokedDuck [27] generate provenance in precisely this representation.

3.3.1 Per-operator Provenance-based Evaluation. All circuit nodes produced by the same query operator have the same operation type and access their inputs in the same way. For instance, σ evaluates the same predicate for every input tuple. This circuit-based execution is a consequence of the circuit-based representation that existing provenance-capture DBMSes such as ProvSQL [34] capture and expose.

An alternative to this per-tuple circuit representation is a per-operator representation [31] where each operator's provenance is modeled as a 1D or 2D integer arrays, where the i^{th} element stores the tuple offsets of the inputs that contributed to the i^{th} output tuple. For instance, filter's provenance is a 1D array B_σ where $B_\sigma[i]$ stores the input tuple offset for the i^{th} output tuple. The join provenance in Figure 1(e) consists of integer arrays B_l (and B_r) where their i^{th} element stores store the offset of the left (and right) input tuple that contributed to the i^{th} output tuple. Similarly, the group-by provenance B_γ is an array where the i^{th} element stores the offsets for all input tuples in the group that emits the i^{th} output tuple.

This array representation is amenable to fast evaluation because the structure of the computations for all tuples of a given operator are identical, and the regularity results in intervention logic is amenable to data parallelism and improved instruction and data locality. Figure 1(e) shows example intervention evaluation code for the join-aggregation query. Rather than the base tables A and B, it takes as input interventions I_A and I_B where each element is set if we want to intervene on the corresponding input tuple (e.g. delete/scale). Since the join's provenance arrays B_l and B_r specify which input tuples contribute to each intermediate join result, $J = I_A[B_l] \vee I_B[B_r]$ computes which join results are still present under the intervention. Similarly, the aggregation provenance B_γ is used to locate all contributing input tuples and exclude any selected tuples from the updated results by inverting J. Since these are standard matrix operations, we can batch interventions by simply adding extra columns of interventions to I_A and I_B.

3.3.2 Fast Provenance Systems. The key opportunity is that recent work developed a columnar analytical database that efficiently captures per-operator provenance polynomials for SPJAU queries. The system, SmokedDuck [27], instruments DuckDB's [32] physical operators to emit dense integer arrays in precisely the above representation (e.g., B_l, B_r, B_γ in Figure 1(e)). On TPC-H queries, its provenance capture incurs an average runtime slowdown of 10% (max: 33%) across SF=1 to 10; the resulting lineage is compact (8MB vs 1GB database). As we show in the end-to-end experiments, this overhead is sufficiently low to support ad-hoc provenance capture and provenance applications such as FaDE while out-performing specialized systems like Caravan, ERDB, and others.

3.4 Scope

FaDE is an application of fine-grained provenance-capturing databases [27, 31], meaning it takes as input the dense array provenance representation generated during query execution by a system such as SmokedDuck [27], and uses the provenance to accelerate hypothetical deletion and scaling what-if queries for SPJAU queries. It supports nested group-by over the last group-by operator, similar to supported class of queries by ERDB [33] where SPJU operators appear below all aggregates. In general, we limit scaling to attributes used in aggregate functions and that do not affect the query's data flow (e.g., filter or join conditions, grouping expressions)—these are often called "measures" in BI analytics. This ensures new tuples are not added to the query result through attribute scaling/updates. We assume Union under bag semantics concatenates its input tables and under set semantics is followed with a group-by operator.

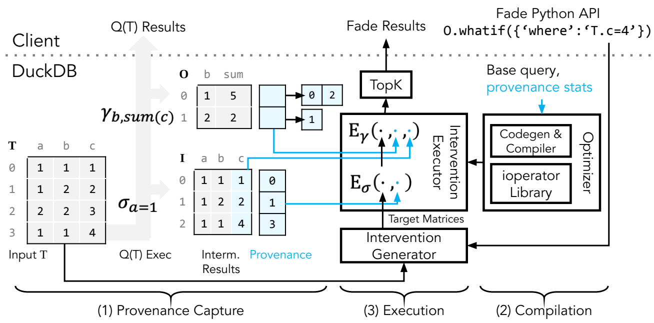


Figure 2: FaDE runs in two phases within SmokedDuck. In phase 1, SmokedDuck captures per-operator provenance and k-semimodules (e.g., values of c) during base query execution (blue data). In phase 2, the user submits a what-if query, and the optimizer chooses and configures the optimal physical intervention evaluation operators and plan. FaDE then quickly generates and evaluates the interventions, and returns the top-k based on the optimization metric.

4 SYSTEM ARCHITECTURE

FaDE extends SmokedDuck to support fast and high-throughput what-if analysis. This section describes the FaDE architecture and its components.

4.1 Architecture

FaDE is implemented inside SmokedDuck to avoid Inter-Process Communication between FaDE and SmokedDuck. Figure 2 shows the end-to-end workflow for $Q = \gamma_{b,sum(c)}(\sigma_{a=1}(T))$. FaDE runs in three phases. When the client submits the base query Q , FaDE first rewrites the physical plan into a canonical form (Section 2.3.1) to ensure that, when the query engine captures provenance during plan execution, FaDE caches the values of all attributes referenced in aggregation expressions (e.g. `T.c` for Q). During query execution, FaDE uses SmokedDuck's instrumented operators [27] to efficiently capture backward provenance on a per-operator basis (blue data in Figure 3). FaDE then walks the query plan and generates an intervention evaluation plan where each operator translates into a corresponding intervention operator (*ioperator*) (e.g. γ is replaced with E_γ and σ with E_σ). The optimizer uses memory resources and provenance statistics gathered during base query execution to decide the appropriate physical ioperators and the optimal level of batching and parallelism (Section 5.5). Finally, when the user makes a `whatif()` call (e.g. `whatif({'where':'T.c=4'})`), FaDE parses any string predicates in the `whereclause` and scans the database to generate an internal target matrix representation. (e.g. `'T.c=4'` is replaced with `[0,0,0,1]`), (Section 5.2). The executor loads the previously captured provenance into the intervention plan and processes the interventions as a batched stream. The optional top-k module evaluates the optimization metric and returns the optimal interventions, and their updated query results.

Many of the above steps can be freely scheduled depending on application needs. Although provenance capture is low overhead, if the base query cannot accept *any* slowdown, it can be run without provenance capture and then scheduled to run again with provenance capture during user think time [9]. Similarly, the optimizer and compilation can run immediately after provenance capture, or deferred to when the user makes a `whatif()` call via the Python API; in this work we use the latter policy.

4.2 Naive Design

The major challenge to designing FaDE is to ensure that the end-to-end what-if query process is fast. This means that any pre-processing must be cheap, that intervention generation and evaluation are high-throughput, and that no component is a bottleneck. We outline a naive design that fails to meet these requirements, and the challenges that still must be addressed.

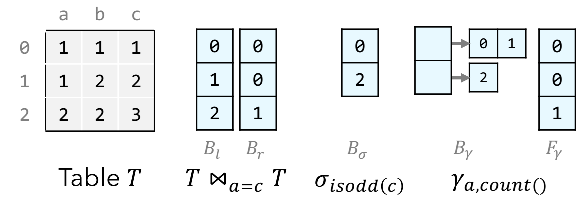


Figure 3: FaDE takes as input backward provenance produced by SmokedDuck (denoted $B_\square$), which maps output to input tuple offsets for physical SPJAU operators. Fade transforms the group-by provenance to a forward representation F_γ that maps input to output offsets.

4.2.1 Intervention Plan. FaDE walks the base query plan and generates an intervention plan with the identical structure, but each *intervention operator* (*ioperator*) is specialized to evaluate deletion or scaling interventions. Each ioperator is initialized with its corresponding operator's provenance (blue arrows in Figure 2); it takes as input a target matrix and outputs a target matrix for the parent ioperator (Section 4.2.2). Aggregation ioperators additionally take the cached values of the attributes referenced in the aggregation functions and an optional scaling specification as input, and also output updated aggregate values. Execution is bottom up on a per-operator basis. We now describe naive templates for each operator using numpy-style pseudocode. Projection under bag semantics is 1-to-1 so can be ignored, and modeled as a group-by under set semantics. Figure 3 lists example provenance representations used by each ioperator.

Selection (E_σ)**.** The backward provenance B_σ`[o]` stores the offset i of the input tuple that contributed to the o^{th} output. It scans the provenance and copies the corresponding bits from target matrix *in* to the target matrix *out*:

```
for o, i in enumerate(B_σ):
    out[o,:] = in[i,:]
```

Join ($E_{\bowtie}$)**.** The backward provenance B_l`[o]` and B_r`[o]` respectively contain the input tuple offsets i_l and i_r that generated it. Under `AND` semantics, an output tuple is deleted if both inputs are deleted (their bits are both 1), while under `OR` semantics, the output is deleted if either input is deleted. In the following code, in_l and in_r are the target matrices from the left and right tables, `op` corresponds to logical `&` or `|` depending on the semantics:

```
for o, (i_l, i_r) in enumerate(zip(B_l,B_r)):
    out[o,:] = in_l[i_l,:] op in_r[i_r,:]
```

Aggregation (E_γ)**.** The backward lineage B_γ`[o]` stores a list of input offsets (*iids*) for the o^{th} group (output tuple). The ioperator also takes the cached values of all attributes referenced in the aggregation function and an optional list of scaling factors `SFs` for scaling interventions. For deletion interventions, if any input is not deleted, then its group is non-empty and the output tuple is preserved. For legibility, the code assumes a single scale factor `sf` and `sum` aggregation over a single attribute cached in *vals*, but the logic supports arbitrary aggregations e.g., `median(a+b)`:

```
for o, iids in enumerate(B_γ):
 vs = vals[iids]
 out[o,:]=in[iids,:].apply(&)      // deletion only
 agg[o,:]=(vs×!in[iids,:]).sum(axis=0) // deletion only
 agg[o,:]=(vs×(1+(sf×in[iids,:]))).sum(axis=0) // scaling
```

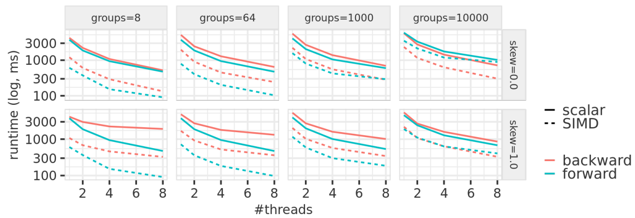


Figure 4: Group-by intervention evaluation latency using forward (Listing 5.1.2) vs. backward provenance (Listing 4.2.1), varying groups & threads for scalar & SIMD variations.

`in` is inverted via `!` because 1 means delete, whereas it means scale for the scaling intervention. Subsequent group-by ioperators consume the target matrix `out` and updated values `agg`.

4.2.2 Intervention Generation. FaDE by default represents the target matrix as a n×m dense matrix to encode m interventions for a relation with n rows. Each column is called a target list and encodes the subset of intervened tuples using a bit mask. The application can directly submit the target matrix that FaDE passes along to the intervention evaluator. However, if the user specifies conjunctive predicate strings, then FaDE splits the string into separate per-relation predicates p_R for each relation R. FaDE then computes a target list TL_R by running: $TL_R = \Pi_{p_R}(R)$. m interventions (a parameterized predicate such as `A=?` or an explicit list of predicate strings) can be batched by executing $\Pi_{p_{R_1},...,p_{R_m}}(R)$ to construct the target matrix.

4.3 Challenges

The naive design exhibits numerous performance bottlenecks that cripple its end-to-end throughput. We ran a microbenchmark using TPC-H Q7 with SF=1 and 1000 random interventions over all 6 input tables. The naive design processes 8 interventions/second end-to-end, when taking intervention generation and evaluation into account. We list the main reasons for the poor performance below, along with the proposed solution that we detail in the following sections. Our final optimized system increases the throughput by 1920× to 174K interventions/second.

4.3.1 Group-by Operator Design. Although the above ioperator designs for join and filter based on their backward provenance are already high-throughput, aggregation (Listing 4.2.1) exhibits severe scalability limitations that lead to poor throughput. Figure 4 illustrates the results of a scalability benchmark. We initialize the provenance B_γ for a group-by over $10M$ input records with few (8) or many (10K) groups, where the distribution of bucket sizes are either uniform (zipf=0) or skewed (zipf=1). We then vary the degree of parallelization along the x-axis by assigning each worker thread an output tuple to update in a round robin fashion, and report the time to process 1024 interventions. The red line in Figure 4 shows poor scalability for skewed bucket sizes when there are fewer than <1000 groups. This is because the naive work distribution results in imbalance in assigned workload per thread. In general across all groups, scalar FaDE-forward is 1.5× (max: 4×) faster than FaDE-backward, and SIMD variants is 1.8× (max: 3.7×) faster. We discuss SIMD optimization further in Section 6.5.4.

Solution: we design and optimize an alternative ioperator to use forward provenance, which scales linearly with the number of threads (Section 5.1.2).

4.3.2 Intervention Generation. Our declarative target matrix specification—both string predicates and random samples—makes it very easy for users to specify large sets of intervention. Suppose table `T(a,b,c)` contains 1M tuples and the domain of each attribute is 100. Then the string predicate `A=? and B=? and C=?` would generate $100^3 = 1M$ interventions, and the fully materialized target matrix would be 125GB. Even if FaDE can generate the target matrix in batches, intervention generation would cripple the end-to-end throughput.

Solution: we design specialized target matrix representations for the common case of parameterized conjunctive equality predicate strings, which enables a compact sparse representation of size $N \times k$, where N is the relation cardinality and k is the number of parameterized attributes in the predicate. We then specialize the physical ioperators to efficiently evaluate over this sparse representation. Section 5.2 describes this in more detail.

4.3.3 Provenance Size. The ioperators use provenance to access values in their input target matrix, however this can lead to non-sequential memory access patterns if the base operator is highly selective. For instance, the backward provenance for group-by can lead to scattered memory accesses of the target matrix when scanning the provenance of a given bucket.

In addition, provenance capture is performed bottom-up, so it could have materialized a considerable amount of irrelevant provenance data if the base query was highly selective. Thus, intervention execution performs unnecessary work that is proportional to the amount of irrelevant provenance. For instance, if a base query plan executes $\sigma_{false}(A \times B)$, the cross-product provenance size with be quadratic in size, yet the query does not generate any results so all of its provenance is irrelevant. Note that this contrasts from circuit-based provenance representations which by definition, only contain derivations of tuples in the query result.

Solution: FaDE post-processes the provenance to prune irrelevant provenance and transform backward to forward provenance representations if needed (Section 6.5.3). We show that after pruning, provenance evaluation is linear in the number of tuples in the intermediate result after all joins.

4.3.4 Large Plan Space. Aggregation necessitates multiple physical ioperators that consume backward and forward provenance. In general, aggregation and scaling interventions introduce many more complex trade-offs that are dependent on the properties of the base query, database contents, and interventions. For instance, aggregation ioperators need to read and write the attribute values referenced in their expressions, and the user may want to update many aggregate expressions in the output. This greatly increases memory usage as compared to the bit-packed target matrices, and adversely affects the degree of parallelism and batching that the intervention executor can employ. Furthermore, aggregation exhibits different constraints performance characteristics depending on whether using backward or forward provenance representations; for instance, the forward approach is restricted to aggregates that can be incrementally updated.

Solution: we design an optimizer to quickly search through the large physical plan space to choose the appropriate physical operators and configure them for high *end-to-end* throughput. A benefit of using provenance-based evaluation is that all cardinalities and needed statistics are apriori known.

5 HIGH-THROUGHPUT FADE DESIGN

In this section, we describe optimizations for multi-dimensional parallelism using both multi-threading and vectorization, as well as a technique we refer to as provenance pruning which provides upper bounds for execution SPJAU queries that are linear in the join output size.

5.1 Provenance Post-processing

5.1.1 Provenance Pruning. In contrast to provenance circuits, FaDE can "over-produce" provenance that does not contribute to any output results. Prior works have proposed pruning provenance [7, 11, 26, 35] as a way to reduce its size, however it has not been proposed in a general workflow context for the purpose of supporting provenance-based intervention evaluation for SQL queries. Pruning greatly accelerates intervention evaluation—up to 80× even taking pruning cost into account (Section 6.5.3).

The high level algorithm is straight forward. Starting with a desired subset of the output tuple offsets (say, the user only wants to update two

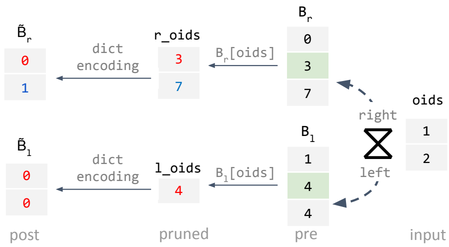


Figure 5: Provenance pruning propagating filtered tuples `oids` from the parent of a join to each of its children.

of the output tuples), we recursively walk the tree top down, use each operator's backward provenance to lookup the input tuple offsets, and use those as the desired output offsets for the child operator. For example, Figure 5 prunes the provenance of a join operator. `[1,2]` are the output ids we wish to keep, so we look up in B_l and B_r for their contributing input offsets—`[4]` and `[3,7]` are the outputs for the left and right child, respectively. We then recompute the backward provenance so they index into the `l/r_oids` arrays for the child operators.

Since group-by does not filter tuples and is always after filter and join in the canonicalized query plan, we only require pruning logic for filter and join. The logic is simple and efficient. In the below pseudocode, `oids` is the set of output ids to keep, $B_{\sigma,l,r}$ denotes the backward provenance for filter and join's left and right children, and `np.unique()` deduplicates its first argument and indices into the deduplicated array to reconstruct the original (the pruned backward provenance). `{c,l,r}_oids` denote output ids of the child/left/right operators to keep.

```
// filter
iids = Bσ[oids] // oids is output ids to keep
c_oids, Bσ = np.unique(iids, return_inverse=1)
// join
l_iids, r_iids = Bl[oids], Br[oids]
l_oids, Bl = np.unique(l_iids, return_inverse=1)
r_oids, Br = np.unique(r_iids, return_inverse=1)
```

5.1.2 Forward Provenance. The group-by ioperator design that uses backward provenance suffers from poor scalability due to random lookups and work imbalance across threads (Section 4.3.1). Even though it is necessary for holistic aggregates that require processing all of a group's tuples at once, aggregates that are distributive can be computed incrementally and in parallel. Thus, we propose a design based on forward provenance, which is a simple 1D array that maps input tuple offset to the output tuple (bucket) it contributes to (F_γ in Figure 3). This design can horizontally partition the input and evenly distribute work across threads. *vals* caches the aggregated attribute values, *in*, *out* are the input and output target matrices, `sf` is the scaling factor, and *agg* is the updated aggregates.

```
for i, o in enumerate(Fγ):
  out[o,:] &= in[i,:]                  // deletion only
  agg[o,:] = vals[i] × !in[i,:]        // deletion only
  agg[o,:] = vals[i] × (1+(sf×in[i,:])) // scaling only
```

Figure 4 shows that this forward provenance design scales linearly and is generally robust across experimental conditions.

5.2 Efficient Intervention Generation

As described in Section 4.3, the dense target matrix representation is prohibitively large to materialize when there are hundreds or thousands of interventions—easily the case when using parameterized predicate strings to specify the target matrix. We now describe an efficient sparse representation for conjunctive equality predicates of the form `a=? and b=? ...`, as well a generalization that supports conjunctions of range predicates. Both accelerate all ioperators prior to aggregation—the former guarantees memory and runtime costs linear in the input relation, while the latter defers target matrix materialization until aggregation. Our examples will focus on equality and ≤ operators, but inequality and other range operations are straightforward to support via logical transformations ($a! = 1$ is $!(a = 1)$).

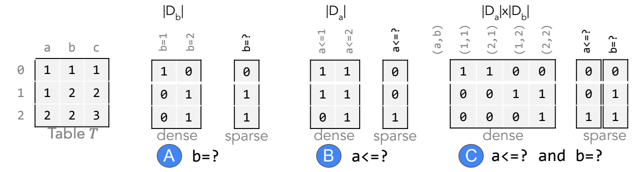


Figure 6: The dense target matrix uses a column per value to model parameterized predicates. (A) Equality predicates match one parameter value per tuple; the sparse array stores the matching value's index. (B) Range predicates order the values such that matching the i^{th} value matches all larger indexes; the sparse array stores the first matching value's index. (C) Conjunctions store a sparse array per clause.

5.2.1 Conjunctive Equality Predicates. This class of predicates is commonly used in applications like explanation engines [33, 36] to search for user-interpretable predicates that the output is most sensitive to. A The key property is that each tuple matches exactly one predicate (intervention), and thus the target matrix can be represented as a single integer array the size of the input relation. The sparse array stores the index of the non-zero column index in the dense matrix. For a predicate string of the form `A.a=?` and `A.x=?` and `B.b=?`, we first partition by table (e.g., `A.a=?` and `A.x=?`, `B.b=?`), and generate sparse representations for the predicates over each table (e.g., one for A and one for B). Note that the sparse representation is equivalent to a dictionary encoding of the column, so in many DBMSes, constructing this is free for string attributes. Let N_x be the active domain size for attribute x, and $Attrs(p)$ be the set of unique attributes in the string predicate p; the total number of interventions is $\Pi_{attr \in Attrs(p)} N_{attr}$.

The join propagates the left and right sparse target matrices (distinguished by a 'sparse' prefix); N_r is the number of interventions in r:

```
for o, (il, ir) in enumerate(zip(Bl,Br)): // join
    sparse_out[o] = sparse_inl[il] × Nr + sparse_inr[ir]
```

The first group-by after joins and filters needs to materialize the updated aggregate values for each intervention, so it emits a $N \times M$ matrix where N (M) is the number of output rows (interventions). We illustrate deletion intervention evaluation and the variation for scaling interventions is very similar. The core logic enumerates over the intervention space. For the itv'th intervention, we test if the input row is included in the intervention set by replacing `pred(sparse_in[i], itv)` with `sparse_in[i]==itv`.

```
out[:,:] = 1
for i, o in enumerate(Fγ):
  for itv in range(sin.max()):
    pval = pred(sparse_in[i], itv)
    out[o,k] &= pval  //deletion
    agg[o,k] += vals[i] × !pval      //deletion
    agg[o,k] += vals[i] × (1+(sf×pval) //scaling
```

The subsequent group-bys consume fully materialized target matrices, and reuse the previous templates.

5.2.2 Conjunctive Range Predicates. Range predicates are challenging because each input tuple can satisfy multiple interventions. B We observe that the range clause a≤i implies a≤j if $i \leq j$. Thus, for each tuple, the sparse representation for a≤? can simply store the minimum attribute value that satisfies the predicate. We replace `pred(sparse_in[i], itv)` with `sparse_in[i]<=itv` to test if the i'th row belongs to the intervention set.

The ioperator designs are similar to those for equality predicates. The filter and join ioperators simply propagate the sparse target matrices and the group-by ioperator materializes all interventions. C Given a predicate over relation R with k parameterized clauses, we construct a $|R| \times k$ matrix where each parameterized clause is encoded in a separate column. The inclusivity test is the conjunction of each parametrized clause test.

5.2.3 Variations. There are many variations of the above code templates omitted for brevity, such as batch-wise intervention processing or supporting multi-attribute aggregation functions. For instance, suppose we wish to scale y in the query $\gamma_{f(x+a),f(x*x)}(\gamma_{a,f(y)\to x}(T))$. The outer group-by references the nested aggregated value x as well as the value of a. x will be emitted from the child ioperator as a matrix that encodes its value across all interventions, while a is cached as an array. For these cases, FaDE compiles a custom aggregation ioperator based on the base query and which final aggregates the `whatif()` call requests.

5.3 Incrementally Removable Aggregations

The primary downside for the preceding group-by ioperator designs is that the cost to update a single output tuple's aggregate is still linear in `vals` because we need to scan `vals` in order to remove the deleted values and recompute the output. This heavily penalizes group-bys with many groups. Incrementally removable [36] aggregation functions, such as `sum` and `avg`, are amenable to an alternative design where updating each output tuple is linear in its provenance size. The idea is to subtract the deleted values from the original aggregation results. Below, we assume a sparse target matrix representation and `sum` aggregation for clarity.

The first group-by ioperator initializes *agg[i,:]* with the original aggregation result in the i^{th} output tuple, with one copy per intervention. It then incrementally updates the original result by subtracting the deleted value or adding the scaled value. The output target matrix at *out[i,j]* is set to 1 only if all tuples mapped to the i^{th} output group are deleted after applying the j^{th} intervention. As such, we record how many tuples map to an output group i in `count[i]`, and compare it with `del_count`. If they are equal, then the corresponding output group is deleted.

```
del_count[:,:] = 0
for i, o in enumerate(F_γ):
  del_count[o, sparse_in[i]] += 1        // deletion only
  agg[o,sparse_in[i]] -= vals[i]         // deletion only
  agg[o,sparse_in[i]] += vals[i]×(sf-1) // scaling only
out = del_count == count[:]
```

This forward provenance-based design is always beneficial for incremental aggregates when using the sparse target matrix representation because the sparse representation reduces the target matrix from quadratic to linear in size, and the total ioperator cost to linear in the number of input values. Thus, we exclusively use the forward ioperator design in the experiments.

5.4 Parallel Execution

FaDE executes the entire plan over a batch of interventions at a time. This is to simplify the executor design, and because a what-if application may wish to dynamically decide which batch of interventions to evaluate next based on the results of the current batch (e.g., to support pruning heuristics). Thus the goal is to process each batch as quickly as possible.

For horizontal parallelization, the provenance and input target matrices are stored in shared memory, and each worker is assigned a subset of the provenance to scan and process. For filter and join, they use backward provenance so this logically partitions the output tuples across workers; it does not exhibit any write contention and scales linearly. For the forward-provenance group-by designs presented in this section, we logically partition by input tuples so there can be write contention for updating the same aggregate values. Instead, each worker writes their results to a private buffer, we merge the buffers to construct the final results at the end. The need for private buffers whose size are proportional to the number of interventions necessitates an optimizer to trade-off memory utilization and throughput improvements. FaDE also uses SIMD vectorization as a form of vertical parallelization.

5.5 Optimizer

A benefit of provenance-based evaluation is that we have full selectivity and cardinality statistics about the inputs (N_o) and outputs (O_o) of every operator. Given these statistics, we developed a simple optimizer to determine, per-operator, the number of interventions per batch B_o and the number of workers W_o to horizontally parallelize. We first use the cost model to evaluate all combinations of (W_o, B_o) for each operator o, where we discretize batches to powers of 2. We then identify the bottleneck operator with the highest cost, and use its optimal batch size B^* as the global batch size. We then iterate over each operator and find the optimal W_o given B^*.

5.5.1 Cost Model. Let A be the number of aggregation functions in the group-by. We describe simplified operator cost models that estimate 1) the total memory needed to execute an operator, which must not exceed a fixed memory bound M, and 2) the expected runtime to evaluate I total interventions. For clarity, we ignore operator selectivity and assume each aggregation references a single and different attribute. The cost of a plan is the sum of all operator costs.

$$mem_o = \begin{cases} W_o * ((A * O_o * B) + O_o \frac{B}{64})) + N_o \frac{B}{64} + (A * N_o) & \text{if groupby} \\ \frac{B}{64} * O_o + \frac{B}{64} * N_o & \text{else} \end{cases}$$

$$cost_o \sim N_o \frac{B}{W_o} + \infty * (mem_o > M)$$

The group-by memory usage includes each of the W_o workers' output buffers, the input target matrix, and cached attribute values; the other operators simply require input and output target matrix buffers. The operator cost is proportional to the level of horizontal parallelization, but is infinity if the memory usage exceeds M.

5.5.2 Additional Heuristics. We use several heuristics for other optimization decisions. We decide to prune the provenance if the base query's join and filter operators filtered more than 30% of the input tuples. We also generate group-by templates that process between $A \in \{1, 2, 3, 4\}$ aggregation functions at a time, and run the optimizer for each A to choose the best setting. If the number of output groups exceed 1000 or if the group-by uses holistic aggregates, then we revert to the slower backward-provenance ioperator design; since this design partitions based on output groups, there is not worker contention and we simply use all workers. Finally, if the `whatif()` call specifies a metric function, we can restrict interventions to only update the output aggregated attributes (often only one aggregation) that are referenced by the metric.

5.6 Cascade-Delete

The user may wish to define a target matrix over a table T not referenced by the base query Q. Although it may appear that the intervention does not affect Q, it can due to referential integrity constraints—deleting a tuple in T may cause a cascading delete that affects a table referenced by Q. Prior work [33] requires knowing the desired IVM intervention apriori, and composes a join query to translate it into a delta over the queried relations. FaDE uses a similar mechanism but supports arbitrary ad-hoc interventions. For each path $T \rightsquigarrow R$ where R is referenced by Q, it executes the query $Q_{T\rightsquigarrow R} = \gamma_{T.rid}(T \bowtie \ldots \bowtie R)$ and capture its provenance. FaDE then uses this provenance to propagate the target matrix over T into a target matrix over R, and then executes the intervention plan as usual.

6 EXPERIMENTS

We compare FaDE with state-of-the-art IVM and circuit-based view maintenance systems in terms of raw intervention evaluation throughput on the standard TPC-H benchmark. We report end-to-end comparison with the ERDB on real-world datasets and conclude with ablation studies.

6.1 Experimental Settings

6.1.1 Systems. **DBT**: We configure DBT to generate C++ code it evaluates one intervention at a time. For each table used by the query, we apply all deletions/scaling interventions of a single table $\forall e \in \Delta D$. We report code compilation and initial database loading separately from intervention evaluation runtime. Scaling interventions are treated as update deltas (delete then insert). **DBT-P** is a variation that first uses backward provenance to prune the input database to the sufficient subset to evaluate the interventions.

All systems are logically equivalent under deletion and scaling interventions. Given the same interventions, all systems return the same solution.

ProvSQL: To the best of our knowledge, ProvSQL [34] is the only other provenance system that supports provenance-based view maintenance. We run the ProvSQL extension on PostgresSQL 13 and report the arithmetic circuit evaluation time relative to base query execution. ProvSQL's circuits do not contain non-contributing tuples, so pruning is not needed. View maintenance systems need to consistently beat from scratch execution to be practically useful, and ProvSQL often fails to do so.

FaDE: We report provenance capture overhead, code generation and compilation, intervention generation and evaluation; end-to-end results consider all four costs. The suffix lists the optimizations used: W_n for horizontal parallelization where n is the number of workers, `P` for pruning, `B` for batching, `D` for SIMD FaDE without any suffix indicate the optimal configuration. FaDE is configured to recompute all aggregates to report worst case results.

ERDB [33] rewrites a set of non-overlapping interventions (each input tuple contributes to at most one intervention) into a single SQL query that can be materialized offline to accelerate intervention evaluation. Our end-to-end evaluation compares with ERDB's reported results on two queries from different workloads (NSF [1] and flights datasets [2]).

6.1.2 Workload and Metrics. We evaluate 11 out of 22 TPC-H queries that both IVM and provenance-based view maintenance can support (namely, SPJAU queries) and vary scale factor from 1 to 10. All queries have one aggregate function except for Q1 with 8 aggregate functions. All queries have an average of 3 output tuples except Q3 (11620 tuples), Q9 (175 tuples), Q10 (37967 tuples). Provenance on SF=1 is on average 19MB (0.8 − 45MB) and 5MB ($3KB$ − 45MB) after pruning; it scales linearly with DB size.

We report throughput as interventions/second, and relative speedup when studying optimizations and scalability. To keep the text easy to read, we will primarily report results for queries {1, 3, 5, 7, 9, 10, 12} as they are representative of the rest of the workload. For completeness, Section 6.2 reports results for all supported TPC-H queries.

6.1.3 Interventions. Our initial experiments generate random interventions with varying tuple intervention probability on all input tables of a query to simulate complex interventions across many database tables. We will then generate sets of interventions using parameterized conjunctive equality predicates, and control the number of interventions by varying the number of unique attributes values for the intervened attributes.

6.1.4 Implementation. All our experiments are executed on Google Cloud `c2-standard-16` machines (16 vCPU, 64GB memory). Vectorized operations use AVX-512, a 512-bit SIMD instruction set on Intel processors. We generate and compile C++ programs using `GCC 11.2.0`.

FaDE uses code generation and compilation if the base query 1) computes aggregation over an expression or 2) contains multiple aggregation functions. In the latter case, we partition the aggregation functions into groups of up to 4 and generate code templates for all four conditions; the optimizer then picks the appropriate templates. FaDE's code generation and compilation takes 1s and is independent of the size of the database.

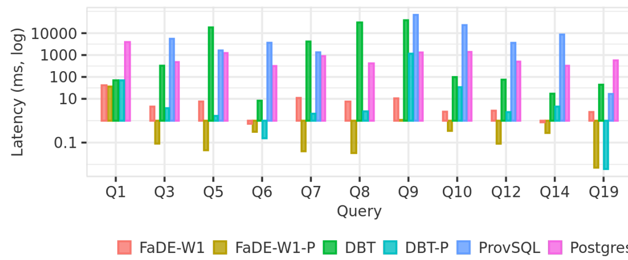


Figure 7: Latency of FaDE-W_1, FaDE-W_1-P, DBT, DBT-P, with intervention probability of 0.1, and the original query on PostgresSQL and ProvSQL without deletions at SF 1. ProvSQL runs out of memory for Q1 & Q8

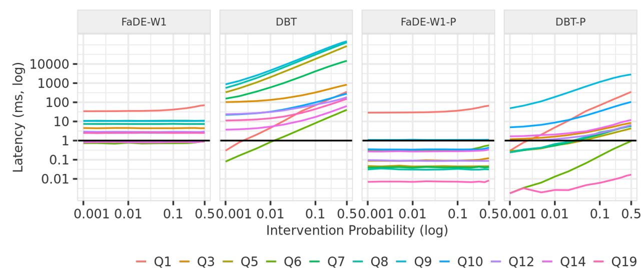


Figure 8: Latency of FaDE-W_1, DBT, FaDE-W_1-P, and DBT-P, varying intervention probability (x-axis) at SF 1.

6.2 Comparison with Baselines

In this first experiment, we compare the performance of FaDE against IVM baselines DBT, and DBT-P on TPC-H SF=1. For a fair comparison, we evaluate single-threaded FaDE without SIMD, using the dense target matrix, and vary pruning (FaDE-W_1 and FaDE-W_1-P).

6.2.1 Setup and Pre-processing cost. FaDE takes on average 141ms to capture provenance, 480ms to cache input to aggregates values, 54ms (max 113ms) to post-process the provenance data and allocate output buffers for the ioperators, and 30ms for provenance pruning—a total of 705ms on average. Both DBT and DBT-P requires a code generation and compilation step, each step having an average latency of 5 seconds. DBT-P also takes on average 14s to prune the base tables (though the procedure is not optimized). Since our focus here is to evaluate raw evaluation throughput, they are excluded from throughput calculations.

6.2.2 Single Deletion Intervention Evaluation. Figure 7 evaluates all systems for a single intervention with intervention probability 0.1 on SF=1. FaDE-W_1 and FaDE-W_1-P take on average 6ms (max: 41ms) as compared to 116ms (max: 1sec) for DBT-P and 8sec (max: 39sec) for DBT. FaDE-W_1-P is faster than DBT-P on average by 129× (max: 1000×) except for Q19 that is instantaneous for both systems and Q6 – a selective single table aggregate query that benefit's from the incremental evaluation of DBT-P. Even without pruning, FaDE-W_1 is competitive with DBT-P; when >99% of tuples are pruned, FaDE-W_1 is only 3ms (max: 9ms) slower than DBT-P.

Figure 8 shows that FaDE variants are insensitive to intervention probability with overall lower execution runtime compared to DBT variants.

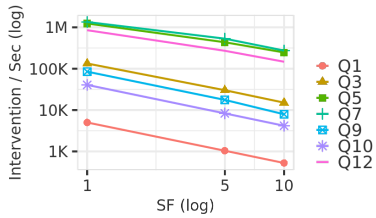


Figure 9: Throughput of intervention evaluation using 8 threads and vectorization for TPC-H scale factors 1, 5 and 10.

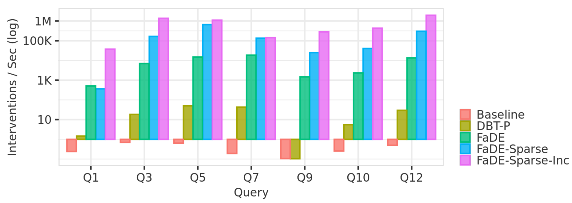


Figure 10: Throughput of the base query execution (Baseline), and throughput when evaluating 2048 interventions using DBT-P, FaDE, FaDE-Sparse, and FaDE-Sparse-Δ.

Excluding Q19 (<0.1ms), FaDE-W_1 and FaDE-W_1-P are consistently 5−>1000× faster than DBT and DBT-P for all queries except Q1 and Q6. The two queries are essentially a single table group-by query of which FaDE re-evaluates its aggregates over the original input, while DBT-P takes advantage of the incrementally removable property of the aggregates and only evaluates aggregates over ΔD.

6.2.3 Single Scaling Intervention Evaluation. We evaluate scaling intervention by selecting a random attribute from lineitem table, applying a fixed scaling factor, and varying the number of scaled tuples by varying the intervention probability between 10^{-3} and 0.5. DBToaster implements scaling as deletion followed by insert. We see similar trends to deletion interventions, DBT-P wins over FaDE-W_1-P for Q1 and Q6 with small target list.

6.2.4 ProvSQL. FaDE-W_1 is faster than ProvSQL by between 7× (Q19) to 10,000× (Q14). Although some of the performance gap can be attributed to differences in the underlying DBMSes, ProvSQL is also on average 15×, and up to 50×, slower than re-running the base query on PostgresSQL. Q1 and Q8 did not finish due to a circuit limit error.

6.3 Scalability

6.3.1 Database Size. Data locality is more important with larger data sizes because the relative size of the CPU cache shrinks. Figure 9 reports the throughput of FaDE with 8 workers, SIMD, and the maximum batch size within memory limits for SF 1 to 10. Throughput decreases linearly with scale factor, yet FaDE still evaluates up to 0.5M interventions per second at SF=10, which for many applications is interactive speed.

6.3.2 Workload-Specific Optimizations. We ablate the sparse encoding and incrementally removable optimizations by evaluating a parameterized conjunctive equality predicate. We intervene the `lineitem` table because it's used by all TPC-H queries. To control the intervention size, we add a synthetic `asynth` attribute to lineitem, assign to it 2048 unique values uniformly, and run a deletion intervention predicate `asynth=?` at SF=10. We include pruning time overhead to FaDE variants.

Figure 10 shows throughput of FaDE, the two optimizations, DBT-P and the base query as a reference. Evaluating 2048 interventions using FaDE, FaDE-Sparse, and FaDE-Sparse-Δ are on average 15×, 260×, and 1000× faster than the base query, respectively. As expected, sparse encoding reduces memory and compute from quadratic to linear, and has the largest wins. Incremental computation particularly benefits aggregate-heavy queries like Q1, where it is 100× faster than FaDE-Sparse.

6.3.3 Memory. From SF=1 to 10, the memory (provenance and dense target matrix) used by naive FaDE with batch size 2048 grows linearly on average from 1.9GB to 19GB. Provenance pruning and sparse encoding reduces used memory by 28× (67MB to 672MB).

6.4 End-to-end Evaluation

We now reproduce experiments from the ERDB explanation experiments [33]. ERDB uses two base queries: a group-by on the NSF awards dataset, and a nested group-by on the Flights dataset. We restate ERDB's reported numbers because their code and exact settings are not available. The numbers are not strictly apples-to-apples because ERDB runs as SQLServer queries.

6.4.1 NSF Awards. This query references two tables: Awards (table A, 400K tuples) and Institution (table B, 419K). The user computes the top 5 institutions that received CS NSF funding, where the core logic is:

$$\gamma_{\text{B.instName},SUM(\text{A.amount})\rightarrow Total}(\sigma_{\text{dir}='CS' \wedge \text{year}\geq 1990}(A \bowtie B))$$

The user then asks why the funding gap between UIUC and CMU is so high. To provide explanations, ERDB evaluates the effect of applying 170K pre-generated deletion interventions that in total delete 1.3M tuples from Awards. Each intervention deletes less than 10 tuples on average. These interventions are specified by 8 parameterized predicates. Some are simple (e.g. "institution.name=?") and benefit from FaDE's sparse representation (Section 5.2). Others are complex expressions that need to be executed explicitly to construct a dense target matrix. For example the following query finds Top-10 PIs with highest average award amounts:

```
investigator.pi IN (SELECT pi FROM (
  SELECT i.pi, AVG(a.award) AS totalAward
  FROM awards AS a JOIN investigator AS i ON a.ai = i.ai
  GROUP BY i.pi ORDER BY totalAward DESC LIMIT 10))
```

Following ERDB, we submit their 8 predicates ahead of time to FaDE and pre-compute their corresponding target lists. ERDB does not report the pre-processing times, so we cannot compare. However, FaDE takes 2 seconds to generate the target matrices, <5ms to capture provenance, and 3ms to post-process the provenance. In total, the provenance is 5MB.

The user submits a what-if query that requests the top 10 pre-registered predicates (by passing in the `targetmatrixid` returned from pre-processing) that minimize the difference between the total awards between both schools (with UIUC having group 0 and CMU having group 1):

```
O.whatif(targetmatrixid,  {'k':3, 'objective':'minimize",
'metric': lambda _, post: post["total"][0]-post["total"][1] })
```

ERDB evaluates this query in 3.4 seconds (1.6s to evaluate interventions, 1.8s to score and rank) despite favorable IVM settings where each intervention deletes ∼10 tuples. In contrast, FaDE evaluates all interventions <5ms—multiple orders of magnitude faster. Even when taking pre-processing time into account, FaDE is faster than ERDB. We believe that at this latency range, FaDE can be used interactively for ad-hoc what-if analyses.

6.4.2 Flights Dataset. The flights dataset contains 123M tuples and is 11GB. The base query runs in 10s and uses a nested aggregation to compute the slope of the linear regression line between scheduled departure and actual departure times. The user asks why the slope is high, and ERDB evaluates 887 interventions in 82s. They do not report pre-computation time.

In FaDE, the base query takes 2.8s to run with additional 53ms overhead from provenance capture and caching input attributes to the aggregate functions. Pre-processing overhead takes 140ms to post-process the provenance, and 1s for compilation. Unfortunately, the ERDB paper does not

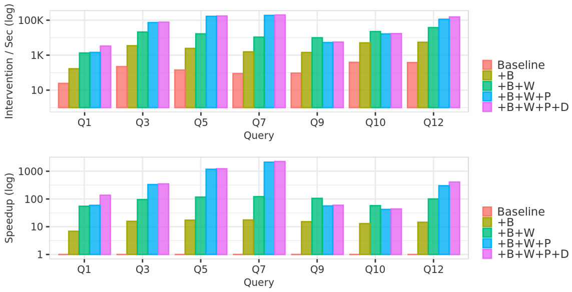


Figure 11: Throughput and Speedup against FaDE-W_1 from Figure 7 as we incrementally stack optimizations.

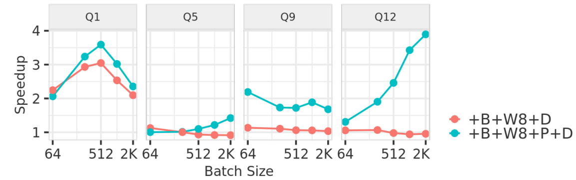


Figure 12: Vectorization speedup with and without pruning over multi-threading execution for representative queries as we vary size of interventions in a batch.

report what their interventions are, so we conservatively generate 887 random interventions with 0.1 tuple intervention probability, which takes 400ms. Finally, evaluating the interventions takes 250ms. Even taking all pre-processing time into account, FaDE is still >28× faster than ERDB's intervention evaluation time.

6.5 Ablation Study

Although FaDE is much faster than all baselines for processing one intervention (Figure 7), it only processes $<1K$ intervention/second (Baseline bar in figure 11). We now incrementally add optimizations from Section 5 that cumulatively increase the throughput to hundreds of thousands of interventions per second. These include batching (+B), horizontal parallelization (+W), provenance pruning (+P), and SIMD (+D).

6.5.1 Interventions Batching. Figure 11 shows that batching 2048 interventions (+B) is 13× faster than the baseline because 1) FaDE packs 64 interventions in one `int64` and only uses one `&` and `|` instruction, 2) the bit-packed target matrix representation improves data locality for all operators, and 3) batching amortizes fixed ioperator call overheads.

6.5.2 Worker Threads. FaDE reduces single operator evaluation by horizontally partitioning input data equally across threads. Figure 11 (+B+W) improves throughput by on average 87× over the baseline when using 8 workers. This is also a linear 8× speedup over batching only, with the exception of Q10 (4.7× faster) because it generates a huge number of output groups, which slows down the final merge step. In general, the merge step will need to aggregate more data than the original group-by operation if the ratio of input to output tuples is below the number of workers.

6.5.3 Provenance Pruning. Provenance strictly reduces the amount of work intervention evaluation needs to perform. Figure 11 (+B+W+P) is on average 579× faster than the baseline and 5× (max: 17×) faster than +B+W. For instance, Q7 has the largest speedup because pruning removes more than 90% of the join and filter provenance. In contrast, pruning is slower than +B+W for Q9 and Q10. This is because we include the pruning cost (avg 30ms, max 340ms) and pruning only removes <30% of the provenance, which does not offset the cost. This is a similar reason why the filter-aggregation Q1 doesn't see strong pruning benefits. This motivated our optimizer heuristic threshold of 30% for deciding whether or not to prune.

6.5.4 SIMD. FaDE leverages the bit-packed target matrix to evaluate multiple interventions at a time using SIMD instructions for the join and filter, and for aggregates whose instructions can be vectorized (e.g., counts, sums).

Figure 11(+B+W+P+D) is on average 1.8× (max: 3.8×) faster than +B+W+P. Pruning is the key ingredient to benefit from SIMD because it increases the goodput of the CPU. Without pruning, most operators are memory-bound so improving CPU utilization has little benefit. After pruning, group-by becomes the bottleneck because it is compute-heavy. Q1 is an exception because it's already compute-heavy query and pruning removes few tuples. At the same time, larger batch sizes are not always better. Figure 12 varies the batch size and reports SIMD speedups with and without pruning relative to (+B+W+P) and (+B+W) respectively for representative queries. We see that larger batch sizes have varying wins for different queries, it can have negative effect as in Q1 and Q9, no effect as in Q5, or increased wins as in Q12. The degree of speedup is affected by the memory pressure from 8 worker as we increase the batch size.

7 RELATED WORK

Section 3.2 presented related work that focuses on IVM and provenance-based view maintenance. This section introduces other relevant works.

Niu et al. [28] also use provenance to accelerate query evaluation, but for data-skipping. The base query's provenance identifies which input tuples or blocks satisfied the base query's predicates $p_\perp$. A new query with predicate $p \subseteq p_\perp$ only needs to read that provenance. They further study how coarse-grained capture can balance provenance capture and space overhead with pruning effectiveness. FaDE focuses on re-evaluating the same query under deletion (filtering) and scaling interventions, however if a new query shares subplans with the base query vis a vis deletions or scaling, FaDE could be used to accelerate their re-execution.

iOLAP [37] combines IVM and provenance to accelerate approximate query processing (AQP) [17]. As AQP processes streams of data, the same tuple may be repeatedly inserted and deleted in an intermediate relation. Rather than de/allocate a tuple each time, iOLAP creates the tuple once and then toggles its deletion status using provenance metadata. Unlike FaDE, iOLAP still uses IVM for the actual query execution.

Panda [22] accelerates view maintenance when the input database becomes stale due to external updates and need to be refreshed. Panda uses provenance to selectively refresh the inputs, but still uses IVM to update the target view after refreshing the inputs.

8 CONCLUSION

The majority of what-if analyses—from sensitivity analysis, deletion-based explanation engines, and how-to analyses—are predicated on the ability to quickly evaluate and rank many deletion and/or scaling interventions. Unfortunately, evaluating interventions is slow: IVM scales poorly with intervention size and requires expensive materialization, while provenance-based view refresh has poor data locality and inefficient execution strategies. At the same time, application-specific solutions use ad-hoc heuristics to prune the search space and still take seconds or minutes to run.

FaDE is a provenance-based intervention evaluation engine that is sufficiently high-throughput to brute force solve many what-if applications within a second. FaDE leverages the relational structure of provenance circuits to generate efficient, parallel evaluation code, and evaluate $>1M$ interventions per second—orders of magnitude higher throughput than any prior approach. This enabling interactive time query explanations over more complex queries and data than previously possible.

REFERENCES

[1] 2023. http://www.nsf.gov/awardsearch/download.jsp.

[2] 2024. Bureau of Transportation Statistics. https://www.transtats.bts.gov/Homepage.asp. Accessed: April 30, 2024.

[3] Firas Abuzaid, Peter Kraft, Sahaana Suri, Edward Gan, Eric Xu, Atul Shenoy, Asvin Ananthanarayan, John Sheu, Erik Meijer, Xi Wu, Jeffrey F. Naughton, Peter Bailis, and Matei Zaharia. 2021. DIFF: a relational interface for large-scale data explanation. *VLDB J.* 30, 1 (2021), 45–70. https://doi.org/10.1007/s00778-020-00633-6

[4] Firas Abuzaid, Peter Kraft, Sahaana Suri, Edward Gan, Eric Xu, Atul Shenoy, Asvin Anathanaraya, John Sheu, Erik Meijer, Xi Wu, Jeffrey F. Naughton, Peter Bailis, and Matei Zaharia. 2018. DIFF: A Relational Interface for Large-Scale Data Explanation. *Proc. VLDB Endow.* 12, 4 (2018), 419–432. https://doi.org/10.14778/3297753.3297761

[5] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. 2012. DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. *Proc. VLDB Endow.* 5, 10 (2012), 968–979. https://doi.org/10.14778/2336664.2336670

[6] Yael Amsterdamer, Daniel Deutch, and Val Tannen. 2011. Provenance for aggregate queries. In *Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2011, June 12-16, 2011, Athens, Greece*, Maurizio Lenzerini and Thomas Schwentick (Eds.). ACM, 153–164. https://doi.org/10.1145/1989284.1989302

[7] Manish Kumar Anand, Shawn Bowers, and Bertram Ludascher. 2010. Techniques for efficiently querying scientific workflow provenance graphs.. In *EDBT*, Vol. 10. 287–298.

[8] Sepehr Assadi, Sanjeev Khanna, Yang Li, and Val Tannen. 2015. Algorithms for provisioning queries and analytics. *arXiv preprint arXiv:1512.06143* (2015).

[9] Leilani Battle and Jeffrey Heer. 2019. Characterizing exploratory visual analysis: A literature review and evaluation of analytic provenance in tableau. In *Computer graphics forum*, Vol. 38. Wiley Online Library, 145–159.

[10] Omar Benjelloun, Anish Das Sarma, Alon Halevy, and Jennifer Widom. 2006. ULDBs: databases with uncertainty and lineage. In *Proceedings of the 32nd International Conference on Very Large Data Bases* (Seoul, Korea) *(VLDB '06)*. VLDB Endowment, 953–964.

[11] Ang Chen, Yang Wu, Andreas Haeberlen, Boon Thau Loo, and Wenchao Zhou. 2017. Data Provenance at Internet Scale: Architecture, Experiences, and the Road Ahead.. In *CIDR*.

[12] Nilesh N. Dalvi and Dan Suciu. 2004. Efficient Query Evaluation on Probabilistic Databases. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*, Mario A. Nascimento, M. Tamer Ozsu, Donald Kossmann, Renee J. Miller, Jose A. Blakeley, and K. Bernhard Schiefer (Eds.). Morgan Kaufmann, 864–875. https://doi.org/10.1016/B978-012088469-8.50076-0

[13] D. Deutch and A. Gilad. 2016. QPlain: Query by explanation. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1358–1361. https://doi.org/10.1109/ICDE.2016.7498344

[14] Daniel Deutch, Zachary G. Ives, Tova Milo, and Val Tannen. 2013. Caravan: Provisioning for What-If Analysis. In *Sixth Biennial Conference on Innovative Data Systems Research, CIDR 2013, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2013/Papers/CIDR13_Paper100.pdf

[15] Daniel Deutch, Tova Milo, Sudeepa Roy, and Val Tannen. 2014. Circuits for Datalog Provenance. In *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*, Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy (Eds.). OpenProceedings.org, 201–212. https://doi.org/10.5441/002/icdt.2014.22

[16] Daniel Deutch, Yuval Moskovitch, Itay Polak, and Noam Rinetzky. 2018. Towards Hypothetical Reasoning Using Distributed Provenance.. In *EDBT*. 461–464.

[17] Minos N. Garofalakis and Phillip B. Gibbons. 2001. Approximate Query Processing: Taming the TeraBytes. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass (Eds.). Morgan Kaufmann. http://www.vldb.org/conf/2001/tut4.pdf

[18] Sneha Gathani, Zhicheng Liu, Peter J Haas, and Cagatay Demiralp. 2022. Understanding Business Users' Data-Driven Decision-Making: Practices, Challenges, and Opportunities. *arXiv preprint arXiv:2212.13643* (2022).

[19] Todd J. Green, Gregory Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China*, Leonid Libkin (Ed.). ACM, 31–40. https://doi.org/10.1145/1265530.1265535

[20] Daniel Haas, Sanjay Krishnan, Jiannan Wang, Michael J. Franklin, and Eugene Wu. 2015. Wisteria: nurturing scalable data cleaning infrastructure. *Proc. VLDB Endow.* 8, 12 (aug 2015), 2004–2007. https://doi.org/10.14778/2824032.2824122

[21] Daniel Hernandez, Luis Galarraga, and Katja Hose. 2021. Computing how-provenance for SPARQL queries via query rewriting. *Proceedings of the VLDB Endowment* 14, 13 (2021), 3389–3401.

[22] Robert Ikeda, Semih Salihoglu, and Jennifer Widom. 2011. Provenance-based refresh in data-oriented workflows. In *Proceedings of the 20th ACM Conference on Information and Knowledge Management, CIKM 2011, Glasgow, United Kingdom, October 24-28, 2011*, Craig Macdonald, Iadh Ounis, and Ian Ruthven (Eds.). ACM, 1659–1668. https://doi.org/10.1145/2063576.2063816

[23] Tomasz Imielinski and Witold Lipski Jr. 1984. Incomplete information in relational databases. *Journal of the ACM (JACM)* 31, 4 (1984), 761–791.

[24] Zachary G. Ives, Todd J. Green, Grigoris Karvounarakis, Nicholas E. Taylor, Val Tannen, Partha Pratim Talukdar, Marie Jacob, and Fernando Pereira. 2008. The ORCHESTRA Collaborative Data Sharing System. *SIGMOD Rec.* 37, 3 (sep 2008), 26–32. https://doi.org/10.1145/1462571.1462577

[25] Ravi Jampani, Fei Xu, Mingxi Wu, Luis Leopoldo Perez, Christopher Jermaine, and Peter J Haas. 2008. Mcdb: a monte carlo approach to managing uncertain data. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 687–700.

[26] Seokki Lee, Sven Kohler, Bertram Ludascher, and Boris Glavic. 2017. Efficiently computing provenance graphs for queries with negation. *arXiv preprint arXiv:1701.05699* (2017).

[27] Haneen Mohammed, Charlie Summers, Sughosh Kaushik, and Eugene Wu. 2023. SmokedDuck Demonstration: SQLStepper. In *Companion of the 2023 International Conference on Management of Data*. 183–186.

[28] Xing Niu, Boris Glavic, Ziyu Liu, Pengyuan Li, Dieter Gawlick, Vasudha Krishnaswamy, Zhen Hua Liu, and Danica Porobic. 2021. Provenance-based Data Skipping. *Proc. VLDB Endow.* 15, 3 (2021), 451–464. http://www.vldb.org/pvldb/vol15/p451-niu.pdf

[29] Fotis Psallidas and Eugene Wu. 2018. Demonstration of Smoke: A Deep Breath of Data-Intensive Lineage Applications. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 1781–1784. https://doi.org/10.1145/3183713.3193537

[30] Fotis Psallidas and Eugene Wu. 2018. Provenance for interactive visualizations. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*. 1–8.

[31] Fotis Psallidas and Eugene Wu. 2018. Smoke: Fine-grained Lineage at Interactive Speed. *Proc. VLDB Endow.* 11, 6 (2018), 719–732. https://doi.org/10.14778/3184470.3184475

[32] Mark Raasveldt and Hannes Muhleisen. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*. 1981–1984.

[33] Sudeepa Roy, Laurel J. Orr, and Dan Suciu. 2015. Explaining Query Answers with Explanation-Ready Databases. *Proc. VLDB Endow.* 9, 4 (2015), 348–359. https://doi.org/10.14778/2856318.2856329

[34] Pierre Senellart, Louis Jachiet, Silviu Maniu, and Yann Ramusat. 2018. ProvSQL: Provenance and Probability Management in PostgreSQL. *Proc. VLDB Endow.* 11, 12 (2018), 2034–2037. https://doi.org/10.14778/3229863.3236253

[35] Azadeh Tabiban, Heyang Zhao, Yosr Jarraya, Makan Pourzandi, Mengyuan Zhang, and Lingyu Wang. 2022. ProvTalk: Towards Interpretable Multi-level Provenance Analysis in Networking Functions Virtualization (NFV).. In *NDSS*.

[36] Eugene Wu and Samuel Madden. 2013. Scorpion: Explaining Away Outliers in Aggregate Queries. *Proc. VLDB Endow.* 6, 8 (2013), 553–564. https://doi.org/10.14778/2536354.2536356

[37] Kai Zeng, Sameer Agarwal, and Ion Stoica. 2016. iOLAP: Managing Uncertainty for Efficient Incremental OLAP. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Ozcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 1347–1361. https://doi.org/10.1145/2882903.2915240