Noname manuscript No.
(will be inserted by the editor)

DBSP: Automatic Incremental View Maintenance for Rich Query Languages

Mihai Budiu · Leonid Ryzhyk · Gerd Zellweger · Ben Pfaff · Lalith Suresh · Simon Kassing · Abhinav Gyawali · Matei Budiu · Tej Chajed · Frank McSherry · Val Tannen

the date of receipt and acceptance should be inserted later

Abstract Incremental view maintenance (IVM) has long been a central problem in database theory and practice. Many solutions have been proposed for restricted classes of database languages (such as the relational algebra or Datalog), restricted classes of queries, and restricted classes of database changes. In this paper we give a general, heuristic-free solution to this problem in 4 steps: (1) we describe a simple but expressive language called DBSP for describing computations over data streams; (2) we give a new mathematical definition of IVM using DBSP; (3) we give an algorithm for converting any DBSP program into an incremental program; this algorithm reduces the problem of incrementalizing a complex query to the problem of incrementalizing the primitive operations that compose the query. Finally, (4) we show that practical database query languages, such as SQL and Datalog, can be directly implemented on top of DBSP, using primitives that have efficient incremental implementations. As a consequence, we obtain a general recipe for efficient IVM for essentially arbitrary queries written in all these languages.

1 Introduction

This paper is an extended version of a VLDB 2023 publication [24], adopting the notations from [25]. The major changes are: adding new incremental program examples (§5.2, §6.1), an expanded implementation section §7, and an experimental evaluation §8. This paper includes only a few short mathematical proofs; all the proofs can be found in an extended technical report [26]; the proofs have also been formalized and verified in the Lean proof assistant [30].

Mihai Budiu, Leonid Ryzhyk, Gerd Zellweger, Ben Pfaff, Lalith Suresh, Simon Kassing, Abhinav Gyawali, Matei Budiu
Feldera.com

Tej Chajed
University of Wiconsin-Madison

Frank McSherry
Materialize.com

Val Tannen
University of Pennsylvania

1.1 Problem and Solution Overview

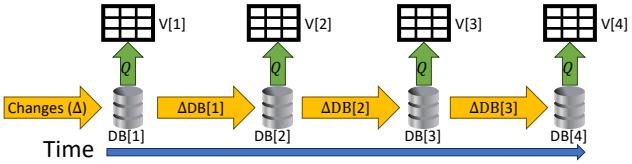
The IVM problem can be stated as follows: we are given a database DB and a view V, described by a query Q. The goal of IVM is to keep the contents of V up-to-date in response to changes of the database.

Consider the following SQL statement:

```
CREATE VIEW V AS
SELECT * FROM T WHERE Age >= 10
```

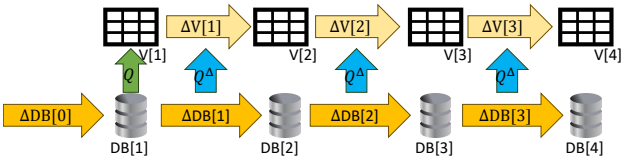
In this example the query Q defining the view V is the `SELECT` statement. The view `V` always contains all the rows of table `T` whose value for the column `Age` is greater than or equal to 10.

In general a query is a function applied to the contents of a database: $V = Q(DB)$. A naïve solution re-executes query Q every time the database changes, as illustrated in the following diagram. Time is the horizontal axis; the horizontal arrows labeled with Δ depict changes to the database. The "up" arrows show the re-evaluation of Q for each database snapshot.



The naive solution is expensive. After the first version of the view has been constructed, an ideal algo-

rithm would compute only *changes* to the view ΔV doing work $O(|\Delta DB|)$. Ideally, we want to construct a new query Q^Δ with the property that $\Delta V = Q^\Delta(\Delta DB)$, i.e., Q^Δ can compute the change of the view from the change of the database:



We call Q^Δ the *incremental* version of Q. If we want Q^Δ to be a function of ΔDB, one can show that the ideal solution as described above is impossible to reach.

In this paper we propose a new way to define Q^Δ, as a form of *computation on streams*. Our model is inspired by Digital Signal Processing DSP [83], applied to databases, hence the name DBSP.

Q^Δ can be significantly more efficient than the naïve solution. As is the case for traditional database queries, the performance of Q^Δ depends both on the query Q but also on the actual data that the query is applied to. Informally, Q^Δ built by our algorithm, is faster than Q by a factor of $O(|DB|/|\Delta DB|)$. In practice this may be an improvement of several orders of magnitude.

Instead of treating the database as a large, changing object, we model it as a sequence or *stream* of database snapshots, shown as $DB[1], DB[2], \ldots$ previously. Similarly, consecutive view snapshots form a stream. DBSP is a simple programming language computing on streams; inputs and outputs are streams of arbitrary values.

The DBSP language has only 4 operators. However, it can express a rich set of computations on streams, including repeated computations (similar to the repeated queries Q above), recursive computations that compute fixed points (like Datalog programs), more general streaming computations, and incremental computations (defined shortly).

The central result of this paper is Algorithm 4 in §4. The input to the algorithm is a DBSP program that computes on a stream of data; the algorithm mechanically transforms it into an incremental DBSP program that computes on a stream of changes.

DBSP is not tied to databases in any way; it is in fact a Turing-complete language that can be used for many other purposes. But it works particularly well in the area of databases, for two reasons:

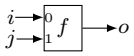
– DBSP operates on values from a commutative group. Databases can be modeled as a commutative group.

– DBSP reduces the problem of incrementalizing a complex program to the problem of incrementalizing each primitive operation that appears in the program. For databases there are known efficient incremental implementations for all primitive operations.

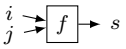
1.2 Core abstractions

1.2.1 Circuits

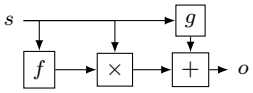
In this paper we use circuit diagrams to depict programs. In a circuit a rectangle represents a function, and an arrow represents an input or output value. The following diagram shows a function f consuming two inputs i (input 0) and j (input 1) and producing one output $o = f(i, j)$:



Most of the functions we deal with are commutative, so we omit input labels, showing the circuit above as:

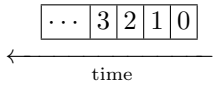


Functions, and their circuits, can be composed, as in the following example showing $o = g(s) + (f(s) \times s)$:

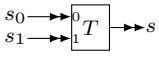


1.2.2 Streams

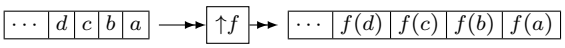
The core notion of DBSP is the **stream**. Given a set A, a **stream** *of values from* A is an infinite sequence of values from A. $\mathcal{S}_A$ denotes the set of all streams with values from A. We write $s[t]$ for the t-th element of the stream s. Think of t as the "time" and of $s[t] \in A$ as the value of the stream s "at time" t. We show streams as a sequence of boxes, with time from *right to left*: e.g., the stream $id[t] \stackrel{\text{def}}{=} t$ is:



A **stream operator** is a function that computes on streams and produces streams. We use "operator" for streams, and "function" for computations on "scalar" values. In circuits we use arrows with a double head to depict streams. The following diagram shows a stream operator T consuming two input streams s_0 and s_1, producing one output stream s; the difference from the previous figure is in the use of double arrows.



We write $s = T(s_0, s_1)$. Given a function $f : A \to B$, we define a stream operator $\uparrow f : \mathcal{S}_A \to \mathcal{S}_B$ (read as "f lifted") by applying function f to each input value independently:



1.3 Databases as streams

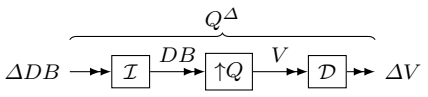
We generally think of streams as sequences of "small" values, such as insertions or deletions in a database. However, we also treat the whole database as a *stream of database snapshots*. We model a database as a stream DB. Time is not wall-clock time, but counts the transactions executed. Since transactions are linearizable, they have a total order. $DB[t]$ is the snapshot of the database contents after t transactions have been applied. We use this notation in the diagrams in §1.1.

Database transactions also form a stream ΔDB, this time a stream of *changes*, or *deltas*, that are applied to the database. The values of this stream are defined by $(\Delta DB)[t] = DB[t] - DB[t-1]$, where "$-$" stands for the difference between two databases, a notion that we will soon make more precise. The ΔDB stream can be produced from the DB stream by the *stream differentiation* operator $\mathcal{D}$; this operator produces as its output the stream of changes from its input stream; we have thus $\mathcal{D}(DB) = \Delta DB$.

Conversely, the database snapshot at time t is the cumulative result of applying all transactions up to t: $DB[t] = \Delta DB[0] + \Delta DB[1] + \ldots + \Delta DB[t]$. The stream operator $\mathcal{I}$ is defined to produce each output by adding up all previous inputs. We call $\mathcal{I}$ *stream integration*, the inverse of differentiation. The following diagram shows the relationship between the streams ΔDB and DB:

In this model a database view is also a stream. Suppose query Q defining a view V. For each snapshot of the database stream we have a snapshot of the view: $V[t] = Q(DB[t])$. A view is thus just a lifted query: $V = (\uparrow Q)(DB)$.

Armed with these basic definitions, we can precisely define IVM. A maintenance algorithm computes the *changes* to the view given the changes to the database. Given a query Q, a key contribution of this paper is the definition of its *incremental version* Q^Δ, using stream integration and differentiation, depicted graphically as:



Mathematically: $Q^\Delta = \mathcal{D} \circ (\uparrow Q) \circ \mathcal{I}$. The incremental version of a query Q is a *streaming operator* Q^Δ which computes directly on changes and produces changes. The incremental version of a query is thus always well-defined. The above definition gives us one way to compute a query incrementally, but applying it naively produces an inefficient execution, since it reconstructs the database at each step. It is in fact as bad as the naive solution. In §3 we show how we can optimize the implementation of Q^Δ. The key property is that the we can "push" the $.^\Delta$ operator "down" in a query plan: $(Q_1 \circ Q_2)^\Delta = {Q_1}^\Delta \circ {Q_2}^\Delta$.

Armed with this general theory of incremental computation, in §4 we show how to model relational queries in DBSP. This immediately gives us a general algorithm to compute the incremental version of any relational query. These results were previously known, but they are cleanly modeled by DBSP. We show how programs containing recursion can be implemented §5 and incrementalized §6 in DBSP. For example, given an implementation of transitive closure in the natural recursive way, our algorithm produces a program that efficiently maintains the transitive closure of a graph as nodes and edges are added and deleted.

1.4 Contributions

This work makes the following contributions:

1. We introduce DBSP, a simple but expressive language for streaming computation. DBSP gives an elegant formal foundation unifying the manipulation of streaming and incremental computations.
2. We describe algorithm 4 for incrementalizing any streaming computation expressed in DBSP that handles arbitrary insertions and deletions from any of the data sources.
3. We describe how DBSP can model various classes of practical queries, such as relational algebra, nested relations, aggregations, and Datalog.
4. We provide the first general and machine-checked theory of IVM. All the theoretical results in this paper [24] have been checked [30] using the Lean proof assistant [37].
5. We describe a practical open-source implementation of this theory as a runtime and a SQL compiler.
6. We give an evaluation of the performance of our implementation.

2 Stream computations

The core notion of our theory of IVM is the **stream**. In this section we introduce streams as infinite sequences of values, and define computations on streams.

2.1 Streams and stream operators

$\mathbb{N}$ is the set of natural numbers (from 0), $\mathbb{B}$ is the set of Booleans, $\mathbb{Z}$ is the set of integers, and $\mathbb{R}$ is the set of real numbers.

Definition 1 (stream) Given a set A, a **stream** *of values from* A, or an A*-stream*, is a function $\mathbb{N} \to A$. $\mathcal{S}_A \stackrel{\text{def}}{=} \{s \mid s : \mathbb{N} \to A\}$ is the set of all A-streams.

When $s \in \mathcal{S}_A$ and $t \in \mathbb{N}$ we write $s[t]$ for the t-th element of the stream s.

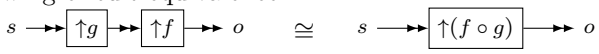
Definition 2 (stream operator) A **stream operator** is a function $T : \mathcal{S}_{A_0} \times \cdots \times \mathcal{S}_{A_{n-1}} \to \mathcal{S}_B$.

Definition 3 (lifting) Given a (scalar) function $f : A \to B$, we define a stream operator $\uparrow f : \mathcal{S}_A \to \mathcal{S}_B$ by *lifting* the function f pointwise in time: $(\uparrow f)(s) \stackrel{\text{def}}{=} f \circ s$. Equivalently, $((\uparrow f)(s))[t] \stackrel{\text{def}}{=} f(s[t])$. This extends to functions of multiple arguments.

For example, $(\uparrow(\lambda x.(2x)))(id) = \boxed{\cdots}\boxed{6}\boxed{4}\boxed{2}\boxed{0}$.

Proposition 1 (distributivity) *Lifting distributes over function composition:* $\uparrow(f \circ g) = (\uparrow f) \circ (\uparrow g)$.

We say that two DBSP programs are **equivalent** if they compute the same input-output function on streams. We use the symbol $\cong$ to indicate that two circuits are equivalent. For example, Proposition 1 states the following circuit equivalence:



2.2 Streams over abelian groups

For the rest of the technical development we require the set of values A of a stream $\mathcal{S}_A$ to form a commutative group $(A, +, 0_A, -)$. The *plus* defines what it means to add new data, while the *minus* allows us to compute differences (deltas). We show later that this restriction is not a problem for using DBSP with relational data.

2.2.1 Delays and time-invariance

Definition 4 (Delay) The **delay operator**[1] z^{-1} produces an output stream by delaying its input by one step: $z_A^{-1} : \mathcal{S}_A \to \mathcal{S}_A$:

$$z_A^{-1}(s)[t] \stackrel{\text{def}}{=} \begin{cases} 0_A & \text{when } t = 0 \\ s[t-1] & \text{when } t \geq 1 \end{cases}$$

We often omit the type parameter A, and write just z^{-1}. For example, $z^{-1}(id) = \boxed{\cdots}\boxed{2}\boxed{1}\boxed{0}\boxed{0}$.

Definition 5 (Time invariance) A stream operator $S : \mathcal{S}_A \to \mathcal{S}_B$ is **time-invariant** (TI) if $S(z_A^{-1}(s)) = z_B^{-1}(S(s))$ for all $s \in \mathcal{S}_A$; in other words, if the following two circuits are equivalent:

The output of a TI operator only depends on its input, but never on the "logical clock" value. For example the operator $S(s)[t] = s[t] + t$ is *not* TI. The composition of TI operators of any number of inputs is TI. The delay operator z^{-1} is TI. DBSP only uses TI operators.

2.2.2 Causal and strict operators

The definitions in this section are used to argue that some circuits with cycles are well-defined.

Definition 6 (Causality) A stream operator $S : \mathcal{S}_A \to \mathcal{S}_B$ is **causal** when $\forall s, s' \in \mathcal{S}_A$, and $\forall t \in \mathbb{N}$ we have: $(\forall i \leq t \;.\; s[i] = s'[i]) \Rightarrow S(s)[t] = S(s')[t]$.

In other words, the output value at time t can only depend on input values from times $t' \leq t$ (they cannot "look" into the future). Operators produced by lifting are causal, and z^{-1} is causal. The composition of causal operators is causal. DBSP only uses causal operators.

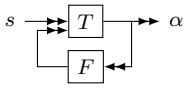
Definition 7 (Strictness) A stream operator $F : \mathcal{S}_A \to \mathcal{S}_B$ is **strict** if $\forall s, s' \in \mathcal{S}_A, \forall t \in \mathbb{N}$ we have: $(\forall i < t \;.\; s[i] = s'[i]) \Rightarrow F(s)[t] = F(s')[t]$.

In other words, the t-th output of $F(s)$ can depend only on "past" values of the input s, between 0 and $t-1$. In particular, $F(s)[0] = 0_B$ is the same for all $s \in \mathcal{S}_A$. Strict operators are causal, but in addition, the current output can only depend on previous inputs. Lifted operators in general are *not* strict. z^{-1} is strict. Strict operators can compute their k-th output before having received their corresponding input.

Proposition 2 *For a strict* $F : \mathcal{S}_A \to \mathcal{S}_A$ *the equation* $\alpha = F(\alpha)$ *has a unique solution* $\alpha \in \mathcal{S}_A$*, denoted by* $fix\alpha.F(\alpha)$.

Thus every strict operator from a set to itself has a unique fixed point. The simple proof relies on strong induction, showing that the solution $\alpha[t]$ depends only on the values of α prior to t.

Consider a circuit with a strict "feedback" edge F:



This circuit is a well-defined function on streams, because the F operator can produce an output before having received the corresponding input, enabling T to compute the first output immediately.

Lemma 1 *If* $F : \mathcal{S}_B \to \mathcal{S}_B$ *is strict and* $T : \mathcal{S}_A \times \mathcal{S}_B \to \mathcal{S}_B$ *is causal, the operator* $Q(s) = fix\alpha.T(s, F(\alpha))$ *is well-defined and causal. If, moreover,* F *and* T *are TI then so is* Q.

[1] The name z^{-1} comes from the DSP literature, and is related to the z-transform [83].

Most DBSP computations are built using just lifted functions and delays. We add two more operators in §6 to support recursive functions.

2.3 Integration and differentiation

Remember that we require the elements of a stream to come from an abelian group A. Streams themselves form an abelian group:

Proposition 3 *The structure $(\mathcal{S}_A, +, 0, -)$, obtained by lifting the $+$ and unary $-$ operations of A, is an abelian group. 0 is the stream with all values 0_A.*

To simplify the notation, we write $a+b$ for streams a, b instead of $a(\uparrow+)b$; we also write $-a$ instead of $(\uparrow-)a$. Stream addition and negation are causal, TI operators.

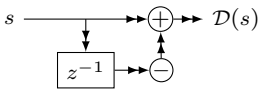
Definition 8 Given abelian groups A and B we call a stream operator $S : \mathcal{S}_A \to \mathcal{S}_B$ **linear** if it is a group homomorphism, that is, $S(a+b) = S(a) + S(b)$ (and therefore $S(0) = 0$ and $S(-a) = -S(a)$).

We write LTI for "linear and TI". Given a linear function $f : A \to B$, the stream operator $\uparrow f$ is LTI. z^{-1} is also LTI.

Definition 9 (bilinear) A function of two arguments $f : A \times B \to C$ where A, B, C are groups, is *bilinear* if it is linear separately in each argument (i.e., it distributes over addition): $\forall a, b, c, d \;.\; f(a+b, c) = f(a, c) + f(b, c)$, and $f(a, c+d) = f(a, c) + f(c, d)$.

This definition extends to stream operators. The lifting of a bilinear function f is a bilinear stream operator $\uparrow f$. An example is lifted multiplication: $f : \mathcal{S}_{\mathbb{N}} \times \mathcal{S}_{\mathbb{N}} \to \mathcal{S}_{\mathbb{N}}, f(a, b)[t] = a[t] \cdot b[t]$.

Definition 10 (Differentiation) The **differentiation operator** $\mathcal{D}_{\mathcal{S}_A} : \mathcal{S}_A \to \mathcal{S}_A$ is: $\mathcal{D}(s) \stackrel{\text{def}}{=} s - z^{-1}(s)$.



We generally omit the type, and write just $\mathcal{D}$. The value of $\mathcal{D}(s)[t] = s[t] - s[t-1]$ if $t > 0$. If s is a stream, then $\mathcal{D}(s)$ is the *stream of changes* of s.

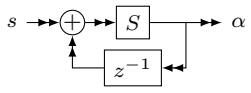
As an example:

$$\mathcal{D}(\boxed{\cdots}\boxed{1}\boxed{2}\boxed{1}\boxed{0}) =$$
$$\boxed{\cdots}\boxed{1}\boxed{2}\boxed{1}\boxed{0} - z^{-1}(\boxed{\cdots}\boxed{1}\boxed{2}\boxed{1}\boxed{0}) =$$
$$\boxed{\cdots}\boxed{1}\boxed{2}\boxed{1}\boxed{0} - \boxed{\cdots}\boxed{2}\boxed{1}\boxed{0}\boxed{0} = \boxed{\cdots}\boxed{-1}\boxed{1}\boxed{1}\boxed{0}$$

Proposition 4 *$\mathcal{D}$ is causal and LTI.*

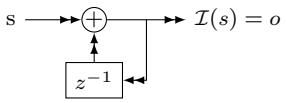
The "feedback loop" built from linear operator and a delay is linear:

Proposition 5 *Is S is causal and LTI, the operator $Q(s) = fix\,\alpha.S(s + z^{-1}(\alpha))$ is well-defined and LTI:*



The integration operator "reconstitutes" a stream from its changes:

Definition 11 (Integration) The **integration operator** $\mathcal{I}_{\mathcal{S}_A} : \mathcal{S}_A \to \mathcal{S}_A$ is $\mathcal{I}(s) \stackrel{\text{def}}{=} \lambda s.\text{fix}\,\alpha.(s + z^{-1}(\alpha))$:



We also omit the type, and write just $\mathcal{I}$. This is the circuit from Proposition 5 using the identity for S.

Proposition 6 *$\mathcal{I}(s)$ is the discrete (indefinite) integral applied to the stream s: $\mathcal{I}(s)[t] = \sum_{i \le t} s[i]$.*

Proof Using the notation $o = \mathcal{I}(s)$ to make formulas more readable, we can see the contents of stream o is produced step by step. Here are the first three steps:

$$o[0] = s[0] + (z^{-1}(o))[0] = s[0] + 0 = s[0]$$
$$o[1] = s[1] + (z^{-1}(o))[1] = s[1] + o[0] = s[1] + s[0]$$
$$o[2] = s[2] + (z^{-1}(o))[2] = s[2] + o[1] = s[2] + (s[1] + s[0])$$

Proposition 7 *$\mathcal{I}$ is causal and LTI.*

Theorem 1 (Inversion) *$\mathcal{I}$ and $\mathcal{D}$ are inverses of each other: $\forall s \;.\; \mathcal{I}(\mathcal{D}(s)) = \mathcal{D}(\mathcal{I}(s)) = s$.*

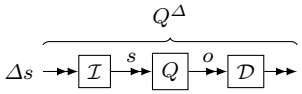
3 Incremental view maintenance

Here we define IVM and analyze its properties.

Definition 12 Given a unary stream operator $Q : \mathcal{S}_A \to \mathcal{S}_B$ we define the **incremental version** of Q as:

$$Q^{\Delta} \stackrel{\text{def}}{=} \mathcal{D} \circ Q \circ \mathcal{I}. \tag{3.1}$$

Q^{Δ} has the same "type" as Q: $Q^{\Delta} : \mathcal{S}_A \to \mathcal{S}_B$. For an operator with multiple inputs we define the incremental version by applying $\mathcal{I}$ to each input independently: e.g., if $T : \mathcal{S}_A \times \mathcal{S}_B \to \mathcal{S}_C$ then $T^{\Delta}(a, b) \stackrel{\text{def}}{=} \mathcal{D}(T(\mathcal{I}(a), \mathcal{I}(b)))$.



If $Q(s) = o$ is a streaming operator, then Q^Δ operates on streams of changes: $Q^\Delta(\Delta s) = \Delta o$.

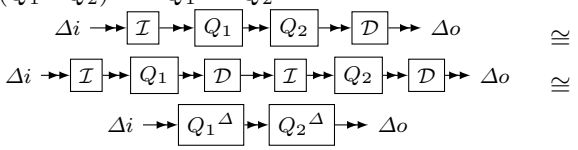
Notice that our definition of incremental computation is meaningful only for *streaming* computations; this is in contrast to classic definitions, e.g. [51] which consider only one change. Generalizing the definition to operate on streams gives us additional power, especially when operating with recursive queries.

The following is one of our central results:

Proposition 8 *(Properties of the incremental version):*

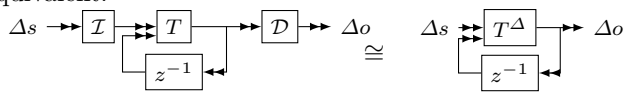
inversion: $Q \mapsto Q^\Delta$ *is bijective;* $Q \mapsto \mathcal{I} \circ Q \circ \mathcal{D}$ *is its inverse*
push/pull: $Q \circ \mathcal{I} = \mathcal{I} \circ Q^\Delta$; $\mathcal{D} \circ Q = Q^\Delta \circ \mathcal{D}$
chain: $(Q_1 \circ Q_2)^\Delta = {Q_1}^\Delta \circ {Q_2}^\Delta$
add: $(Q_1 + Q_2)^\Delta = {Q_1}^\Delta + {Q_2}^\Delta$
cycle: $(\lambda s.fix\alpha.T(s, z^{-1}(\alpha)))^\Delta = \lambda s.fix\alpha.T^\Delta(s, z^{-1}(\alpha))$

Here is the proof of the **chain rule**, which states that $(Q_1 \circ Q_2)^\Delta = {Q_1}^\Delta \circ {Q_2}^\Delta$.



In other words, **to incrementalize a composite query you can incrementalize each sub-query independently**. This gives us a simple, syntax-directed, deterministic recipe for computing the incremental version of an arbitrarily complex query.

The **cycle rule** states that the following circuits are equivalent:



In other words, the incremental version of a feedback loop around a query is just the feedback loop with the incremental query for its body. This result will be used to implement recursive queries in §5.

To execute incremental queries efficiently, we want to compute directly on streams of changes, without integrating them. The following theorems show how this can be done for linear and bi-linear operators:

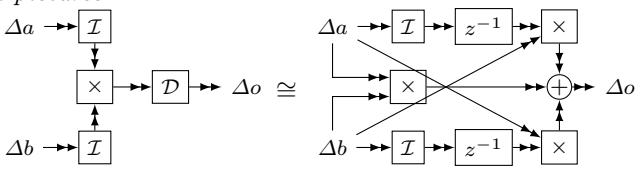
Theorem 2 (Linear) *If* Q *is LTI, we have* $Q^\Delta = Q$.

This implies that stream operators $+$, $-$, $\mathcal{I}$, $\mathcal{D}$, and z^{-1} are identical to their incremental versions.

Theorem 3 (Bilinear) *Using the infix notation: if* $\times$ *is bilinear TI, we have:*

$$(\Delta a \times \Delta b)^\Delta =$$
$$(\Delta a \times \Delta b \ + \ z^{-1}(\mathcal{I}(\Delta a)) \times \Delta b \ + \ \Delta a \times z^{-1}(\mathcal{I}(\Delta b)) =$$
$$\Delta a \times \Delta b + z^{-1}(a) \times \Delta b + \Delta a \times z^{-1}(b)$$

In pictures:



This equation is the well-known formula for join delta queries [62] in terms of streaming computations.

4 IVM for the Relational Algebra

Results in §2 and §3 apply to streams of arbitrary group values. In this section we apply these results to IVM for relational databases. As explained in the introduction, our goal is to efficiently compute the incremental version of a relational query Q defining a view.

However, we face a technical problem: the $\mathcal{I}$ and $\mathcal{D}$ operators were defined on abelian groups, and relational databases in general are not abelian groups, since they operate on sets. Fortunately, there is a well-known tool in the database literature which converts set operations into group operations by using $\mathbb{Z}$-sets (also called z-relations [46]) to represent sets.

We start by defining the $\mathbb{Z}$-sets group, and then we review how relational queries are converted into DBSP circuits over $\mathbb{Z}$-sets. We show in §4.2 that this translation is efficiently incrementalizable because many primitive relational operations use LTI $\mathbb{Z}$-set operators.

4.1 $\mathbb{Z}$-sets as an abelian group

A $\mathbb{Z}$-set is a database table where each row has an associated weight, possibly negative. Given a set A, we define **$\mathbb{Z}$-sets** over A as functions with *finite support* from A to $\mathbb{Z}$. These are functions $f : A \to \mathbb{Z}$ where $f(x) \neq 0$ for at most a finite number of values $x \in A$. We also write $\mathbb{Z}[A]$ for the type of $\mathbb{Z}$-sets with elements from A. Values in $\mathbb{Z}[A]$ can be thought of as key-value maps with keys in A and values in $\mathbb{Z}$, justifying the array indexing notation. If $m \in \mathbb{Z}[A]$ we write $m[a]$ instead of $m(a)$, again using an indexing notation.

A particular $\mathbb{Z}$-set $m \in \mathbb{Z}[A]$ can be denoted by enumerating its elements that have non-zero weights and their corresponding weights: $m = \{x_1 \mapsto w_1, \ldots, x_n \mapsto w_n\}$. We call $w_i \in \mathbb{Z}$ the **weight** of $x_i \in A$. Weights can be negative. We say that $x \in m$ iff $m[x] \neq 0$.

The following table shows an example $\mathbb{Z}$-set with three rows, let's call it R. The first row has value `joe` and weight 1. We never show rows with weight 0.

Row	Weight
joe	1
mary	2
anne	-1

R has three elements in its domain, `joe` with weight 1 (so $R[\texttt{joe}] = 1$), `mary` with weight 2, and `anne` with weight -1. So $\texttt{joe} \in R$, $\texttt{mary} \in R$, and $\texttt{anne} \in R$.

Since $\mathbb{Z}$ is an abelian ring, $\mathbb{Z}[A]$ is also an abelian ring, and thus a group $(\mathbb{Z}[A], +_{\mathbb{Z}[A]}, 0_{\mathbb{Z}[A]}, -_{\mathbb{Z}A})$. Addition and subtraction are defined pointwise: $(f +_{\mathbb{Z}[A]} g)(x) = f(x) + g(x) \forall x \in A$. The 0 element of $\mathbb{Z}[A]$ is the function $0_{\mathbb{Z}[A]}$ defined by $0_{\mathbb{Z}[A]}(x) = 0 . \forall x \in A$. For example, $R + R = \{\texttt{joe} \mapsto 2, \texttt{mary} \mapsto 4, \texttt{anne} \mapsto -2\}$. Since $\mathbb{Z}$-sets form a group, all results from §2 apply to streams over $\mathbb{Z}$-sets.

$\mathbb{Z}$-sets generalize sets and bags. A set with elements from A can be represented as a $\mathbb{Z}$-set by associating a weight of 1 with each element. Bags are $\mathbb{Z}$-sets where all weights are positive. Crucially, $\mathbb{Z}$-sets can also represent arbitrary *changes* to sets and bags. Negative weights in a change represent elements that are being "removed".

The remaining definitions in this section will be used to argue that circuits based on $\mathbb{Z}$-sets can exactly implement the relational algebra operators.

Definition 13 We say that a $\mathbb{Z}$-set represents a **set** if the weight of every element is one. We define a function to check this property $\text{isset} : \mathbb{Z}[A] \to \mathbb{B}$ given by:

$$\text{isset}(m) \stackrel{\text{def}}{=} \begin{cases} \text{true} & \text{if } m[x] = 1, \forall x \in m \\ \text{false} & \text{otherwise} \end{cases}$$

For our example $\text{isset}(R) = \text{false}$, since $R[\texttt{anne}] = -1$.

Definition 14 We say that a $\mathbb{Z}$-set is **positive** (or a **bag**) if the weight of every element is positive. We define a function to check this property $\text{ispositive} : \mathbb{Z}[A] \to \mathbb{B}$. given by

$$\text{ispositive}(m) \stackrel{\text{def}}{=} \begin{cases} \text{true} & \text{if } m[x] \geq 0, \forall x \in A \\ \text{false} & \text{otherwise} \end{cases}$$

If $\text{isset}(m)$, then $\text{ispositive}(m)$. For our example $\mathbb{Z}$-set, $\text{ispositive}(R) = \text{false}$.

We write $m \geq 0$ when m is positive. For positive $m, n \in \mathbb{Z}[A]$ we write $m \geq n$ for iff $m - n \geq 0$. $\geq$ is a partial order.

We call a function $f : \mathbb{Z}[A] \to \mathbb{Z}[B]$ **positive** if it maps positive values to positive values: $\forall x \in \mathbb{Z}[A] . x \geq 0_{\mathbb{Z}[A]} \Rightarrow f(x) \geq 0_{\mathbb{Z}[B]}$. We use the same notation for functions: $\text{ispositive}(f)$.

Definition 15 (distinct) The function $dist : \mathbb{Z}[A] \to \mathbb{Z}[A]$ "converts" a $\mathbb{Z}$-set into a set:

$$dist(m)[x] \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } m[x] > 0 \\ 0 & \text{otherwise} \end{cases}$$

Notice that *dist* "removes" duplicates from multisets, and it also eliminates elements with negative weights. $dist(R) = \{\texttt{joe} \mapsto 1, \texttt{mary} \mapsto 1\}$. While very simple, this definition of *dist* has been carefully chosen to enable us to implement the relational (set) operators using $\mathbb{Z}$-sets operators.

4.2 Relational operators on $\mathbb{Z}$-sets

The fact that relational algebra can be implemented by computations on $\mathbb{Z}$-sets has been shown before, e.g. [47]. The translation of the relational operators to DBSP is shown in Table 1. The first row of the table shows that a composite query is translated recursively. This gives us a recipe for translating an arbitrary relational query plan into a DBSP circuit.

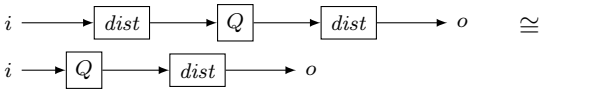
The translation is fairly straightforward, but many operators require the application of a *dist* **to produce sets**. For example, $a \cup b = dist(a + b)$, $a \setminus b = dist(a - b)$, $(a \times b)((x, y)) = a[x] \times b[y]$. Notice that the use of the *dist* operator allows DBSP to model the *full relational algebra*, including set difference (and not just the positive fragment).

Prior work (e.g., Proposition 6.13 in [46]) has shown how some invocations of *dist* can be eliminated from query plans without changing the query semantics; we will see that the incremental version of *dist* incurs significant space costs.

Proposition 9 *Let Q be one of the following $\mathbb{Z}$-sets operators: filtering σ, join $\bowtie$, or Cartesian product $\times$. Then we have $\forall i \in \mathbb{Z}[A]$. ispositive$(i) \Rightarrow Q(dist(i)) = dist(Q(i))$.*

This rule allows us to delay the application of *dist*.

Proposition 10 *Let Q be one of the following $\mathbb{Z}$-sets operators: filtering σ, projection π, map(f)[2], addition $+$, join $\bowtie$, or Cartesian product $\times$. Then we have ispositive$(i) \Rightarrow dist(Q(dist(i))) = dist(Q(i))$.*



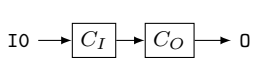
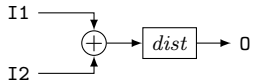
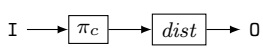
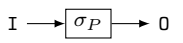
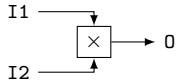
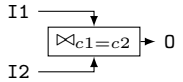
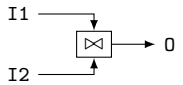
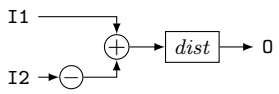
These properties allow us to "consolidate" distinct operators by performing one *dist* at the end of a chain of computations. This optimization is also used in traditional database optimizers.

4.3 Incremental view maintenance

Let us consider a relational query Q defining a view V. To create a circuit that maintains incrementally V we apply the following mechanical steps:

[2] Technically, map (applying a user-defined function to each row) is not relational.

Table 1 Implementation of SQL relational set operators as circuits computing on $\mathbb{Z}$-sets.

Operation	SQL example	DBSP circuit	Details
Composition	`SELECT ... AS O FROM (SELECT ... AS I FROM IO)`		C_I circuit for inner query, C_O circuit for outer query.
Union	`(SELECT * FROM I1) UNION (SELECT * FROM I2)`		*dist* eliminates duplicates. An implementation of `UNION ALL` does not need the *dist*.
Projection	`SELECT DISTINCT I.c FROM I`		Project each row with its weight unchanged. Add up weights of identical rows.
Filtering	`SELECT * FROM I WHERE P(...)`		P is a predicate applied to each row. Select each row separately. If the row is selected, preserve the weight, else make the weight 0.
Cartesian product	`SELECT I1.*, I2.* FROM I1, I2`		The weight of the pair (a,b) is the product of the the weights of a and b.
Equi-join	`SELECT I1.*, I2.* FROM I1 JOIN I2 ON I1.c1 = I2.c2`		Multiply the weights of the rows that appear in the output.
Intersection	`(SELECT * FROM I1) INTERSECT (SELECT * FROM I2)`		Special case of equi-join when both relations have the same schema.
Difference	`SELECT * FROM I1 EXCEPT SELECT * FROM I2`		*dist* removes rows with negative weights from the result.

Algorithm 4

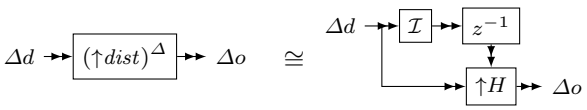
1. *Translate Q into a circuit using the rules in Table 1.*
2. *Apply dist elimination rules (Propositions 9, 10)*[3].
3. *Lift the whole circuit, by applying Proposition 1, converting it to a circuit operating on streams.*
4. *Incrementalize the whole circuit "surrounding" it with $\mathcal{I}$ and $\mathcal{D}$.*
5. *Apply the chain rule from Proposition 8 on the circuit to optimize the implementation.*

This algorithm is deterministic and its running time is given by the number of operators in the query. Step (2) generates an equivalent circuit, with possibly fewer *dist* operators. Step (3) yields a circuit that consumes a *stream* of complete database snapshots and outputs a stream of complete view snapshots. Step (4) yields a circuit that consumes a stream of *database changes* and outputs a stream of *view changes*; however, the internal operation of the circuit is non-incremental, as it rebuilds the complete database using integration operators. Step (5) incrementalizes the circuit by replacing each primitive operator with its incremental version.

[3] If the rules are applied until convergence, the order in which the rules are applied does not matter, since the algorithm is confluent: it always produces the same final result.

Most of the operators that appear in the circuits in Table 1 are linear, and thus have very efficient incremental versions (we discuss complexity in §4.5). A notable exception is *dist*. The next proposition shows that the incremental version of *dist* is also efficient, and it can be computed by doing work proportional to the size of the input change:

Proposition 11 (Incremental *dist*)



where $H : \mathbb{Z}[A] \times \mathbb{Z}[A] \to \mathbb{Z}[A]$ is defined as:

$$H(i,d)[x] \stackrel{def}{=} \begin{cases} -1 & \text{if } i[x] > 0 \text{ and } (i+d)[x] \leq 0 \\ 1 & \text{if } i[x] \leq 0 \text{ and } (i+d)[x] > 0 \\ 0 & \text{otherwise} \end{cases}$$

This incremental implementation of *dist* has been known for a long time; for example, it is called the $\exists$ operator in [77]. This implementation has several interesting features:

– The implementation uses an integral operator $\mathcal{I}$ to reconstitute the *entire input set* of the distinct operator from the set of changes. This is the "top" input of the H function. The implementation needs to maintain the *entire input set* (similar to joins) in order to discover whether an item is new or not.
– Despite this fact, the result of *dist* for an input change can still be computed with work proportional to the size of the change. H detects whether the weight of a row in the full set is changing sign (from negative to positive on a row insertion, and from positive to negative on a deletion) when the row appears in a new change. Only tuples that appear in the input change Δd can appear in the output change Δo, so the work performed is $O(|\Delta d)|$.

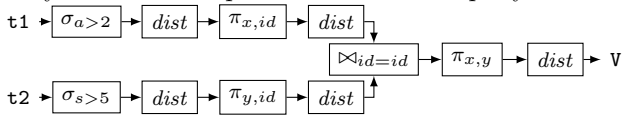
The algorithm 4 reduces the problem of incremental execution of a query plan to the incremental execution of subplans/primitive operators. However, this algorithm can be applied even if we use a primitive P for which no efficient incremental version is known: we can always use the inefficient "brute-force" implementation given by $P^\Delta = \mathcal{D} \circ \uparrow P \circ \mathcal{I}$.

4.4 Relational Query Example

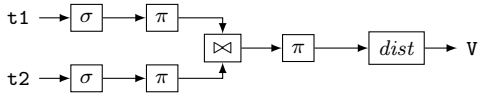
Let's apply the IVM algorithm to the following SQL query:

```
CREATE VIEW v AS
SELECT DISTINCT a.x, b.y FROM (
     SELECT t1.x, t1.id FROM t1 WHERE t1.a > 2
) a JOIN (
     SELECT t2.id, t2.y FROM t2 WHERE t2.s > 5
) b ON a.id = b.id
```

Step 1: Create a DBSP circuit to represent this query using the rules in Table 1; this circuit is essentially a dataflow implementation of the query:

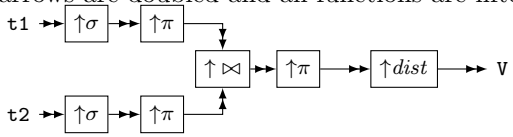


Step 2: eliminate *dist* operators, producing an equivalent circuit: (we omit the subscripts to save space):

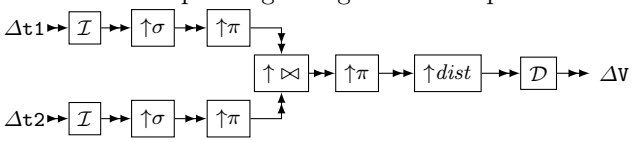


Note that some arrows that were sets in the original circuit may by multisets in the optimized circuit.

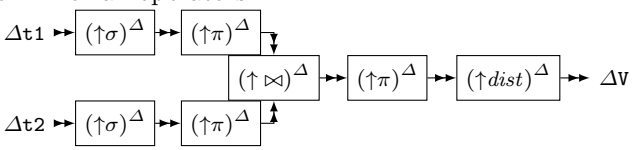
Step 3: lift the circuit to compute over streams; all arrows are doubled and all functions are lifted:



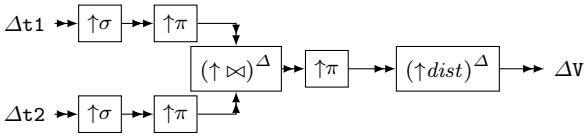
Step 4: incrementalize circuit, obtaining a circuit that computes over changes; this circuit receives changes to relations `t1` and `t2` and for each such change it produces the corresponding change in the output view `V`:



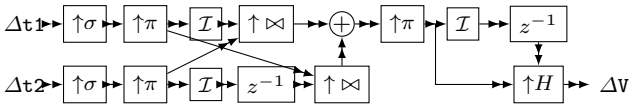
Step 5: apply the chain rule to rewrite the circuit as a composition of incremental operators; notice the use of $.^\Delta$ for all operators:



Use the linearity of σ and π to simplify this circuit (notice that all linear operators no longer have a $\cdot^\Delta$):



Finally, replace the incremental join and the incremental *dist*, with their incremental implementations, obtaining the following circuit (we have used a slightly different expansion for the join than the one shown previously; this one only contains two integrators):



Notice that the resulting circuit contains three integration operations: two from the join, and one from the *dist*. It also contains two join operators. However, the work performed by each operator for each new input is proportional to the size of its input change.

4.5 Complexity of incremental circuits

Incremental circuits are efficient. We evaluate the cost of a circuit while processing the t-th input change. Even if Q is a pure function, Q^Δ is actually a streaming system, with internal state. This state is stored entirely in the delay operators z^{-1}, some of which appear in $\mathcal{I}$ and $\mathcal{D}$ operators[4]. The result produced by Q^Δ on the t-th input depends in general not only on the new t-th input, but also on all prior inputs it has received.

We argue that each operator in the incremental version of a circuit is efficient in terms of work and space. We make the standard IVM assumption that the in-

[4] For standard relational queries, after applying step 5 of the algorithm there will be no $\mathcal{D}$ operators left in the circuit.

put changes *of each operator* are small: $|\Delta DB[t]| \ll |DB[t]| = |(\mathcal{I}(\Delta DB))[t]|$[5].

An unoptimized incremental operator $Q^\Delta = \mathcal{D} \circ Q \circ \mathcal{I}$ evaluates query Q on the whole database DB, the integral of the input stream: $DB = \mathcal{I}(\Delta DB)$; hence its time complexity is the same as that of the non-incremental evaluation of Q. In addition, each of the $\mathcal{I}$ and $\mathcal{D}$ operators uses $O(|DB[t]|)$ memory.

Step (5) of the incrementalization algorithm applies the optimizations described in §3; these reduce the time complexity of each operator to be $O(|\Delta DB[t]|)$. For example, Theorem 2, allows evaluating S^Δ, where S is a linear operator, in time $O(|\Delta DB[t]|)$. The $\mathcal{I}$ operator can also be evaluated in $O(|\Delta DB[t]|)$ time, because all values that appear in the output of $\mathcal{I}(\Delta DB)[t]$ must be present in current input change $\Delta DB[t]$. Similarly, while the *dist* operator is not linear, $(\uparrow dist)^\Delta$ can also be evaluated in $O(|\Delta DB[t]|)$ according to Proposition 11. Bilinear operators, including join, can be evaluated in time $O(|DB[t]| \times |\Delta DB[t]|)$, which is a factor of $|DB[t]/\Delta DB[t]|$ better than full re-evaluation.

The space complexity of linear operators is 0 (zero), since they store no data persistently. The space complexity of operators such as $(\uparrow dist)^\Delta$, $(\uparrow \bowtie)^\Delta$, $\mathcal{I}$, and $\mathcal{D}$ is $O(|DB[t]|)$. They need to store their input or output relations in full.

5 Recursive queries

Recursive queries are very useful in a many applications. For example, graph algorithms such as graph reachability or transitive closure are naturally expressed using recursive queries.

We introduce two simple DBSP stream operators that are used for expressing recursive query evaluation. These operators allow us to build circuits implementing looping constructs, which are used to iterate computations until a fixed-point is reached. The following definition allows us to describe what a fixed point is in terms of streams:

Definition 16 We say that a stream $s \in \mathcal{S}_A$ is **zero almost-everywhere** if it has a finite number of non-zero values, i.e., there exists a time $t_0 \in \mathbb{N}$ s.t. $\forall t \geq t_0 \ . \ s[t] = 0$. Denote the set of streams that are zero almost everywhere by $\overline{\mathcal{S}_A} \subset \mathcal{S}_A$.

Stream introduction.

The delta function (named from the Dirac delta function) $\delta : A \to \mathcal{S}_A$ produces a stream from a scalar value:

$$\delta(v)[t] \stackrel{\text{def}}{=} \begin{cases} v & \text{if } t = 0 \\ 0_A & \text{otherwise} \end{cases}$$

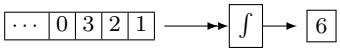
The input of δ has a single arrow, while the output has a double arrow. For example:

Stream elimination.

We define the function $\int : \overline{\mathcal{S}_A} \to A$, over streams that are zero almost everywhere, as $\int(s) \stackrel{\text{def}}{=} \sum_{t \geq 0} s[t]$. $\int$ simply adds up all values in the input stream and produces a scalar result with the sum. $\int$ is closely related to $\mathcal{I}$; if $\mathcal{I}$ is the indefinite (discrete) integral, $\int$ is the definite (discrete) integral on the interval $0 - \infty$. For example, $\int(\boxed{\cdots}\boxed{0}\boxed{3}\boxed{2}\boxed{1}) = 6$.

In circuits constructed for many classes of queries, including relational and Datalog queries given below, the $\int$ operator can be "approximated" without loss of precision by integrating until it encounters the first 0.

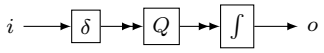
Notice that the input of $\int$ is a double arrow, while the output is a single arrow. E.g.,:



Proposition 12 *δ and $\int$ are LTI.*

Nested time domains.

So far we have used a tacit assumption that "time" is common for all streams in a program. For example, when we add two streams, we assume that they use the same "clock" for the time dimension. However, the δ operator creates a stream with a "new", independent time dimension. We require *well-formed circuits* to "insulate" such nested time domains by "bracketing" them between a δ and an $\int$ operator:



In the above circuit Q is a streaming operator, but the entire circuit is a scalar function, shown by the single input and output arrows.

Algorithm 5 below, which translates recursive queries to DBSP circuits, always produces well-formed circuits.

5.1 Implementing recursive queries

We describe the implementation of recursive queries in DBSP for stratified Datalog. In general, a recursive Datalog program defines a set of mutually recursive relations $O_1, .., O_n$ as an equation $(O_1, .., O_n) =$

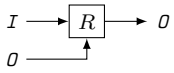
[5] One can write SQL programs where each output row can change even for very small input changes, so this assumption does not always hold in practice, but even in this case, the asymptotic work performed by the incremental query is not worse than the work of the original query.

$R(I_1, .., I_m, O_1, .., O_n)$, where $I_1, .., I_m$ are input relations and R is a non-recursive query.

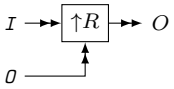
We describe here the algorithm for the simpler case of a single-input, single-output query[6]. The input of our algorithm is a Datalog query of the form $O = R(I, O)$, where R is a relational, non-recursive query, producing a set as a result, but whose output O is also an input. The output of the algorithm is a DBSP circuit which evaluates this recursive query producing output O when given the input I. In this section we build a non-incremental circuit, which evaluates the Datalog query; in §6 we incrementalize this circuit.

Algorithm 5

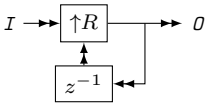
1. *Implement the non-recursive relational query R as described in §4 and Table 1; this produces an acyclic circuit whose inputs and outputs are $\mathbb{Z}$-sets:*



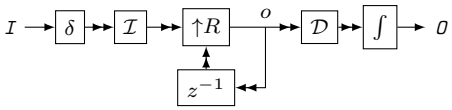
   In all these diagrams we show input 0 of operator R on the left, and input 1 on the bottom.
2. *Lift this circuit to operate on streams:*



   We construct $\uparrow R$ by lifting each operator of the circuit individually according to Proposition 1.
3. *Build a cycle, connecting the output to the corresponding recursive input via a delay:*



4. *"Bracket" the circuit, first with $\mathcal{I}$ and $\mathcal{D}$ nodes, and then in δ and $\int$:*



We argue that the cycle inside this circuit computes iteratively the fixed point of R. The $\mathcal{D}$ operator yields the set of new Datalog facts (changes) computed by each iteration of the loop. When the set of new facts becomes empty, the fixed point has been reached:

Theorem 6 (Recursion correctness) *If isset($\mathtt{I}$), the output of the circuit above is the relation $\mathtt{O}$ as defined by the Datalog semantics of given program R as a function of the input relation $\mathtt{I}$.*

[6] The general case in the companion technical report [26] is only slightly more involved.

Proof Let us compute the contents of the o stream, produced at the output of R. We show that this stream contains increasing approximations of the value of $\mathtt{O}$.

Define the following one-argument function: $S(x) = \lambda x.R(\mathtt{I}, x)$. Notice that the left input of the $\uparrow R$ block is a constant stream with the value $\mathtt{I}$. Due to the stratified nature of the language, we must have ispositive(S), so $\forall x \,.\, S(x) \geq x$. We get the following system of equations:

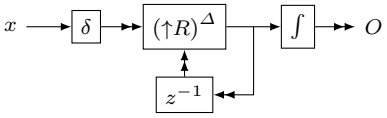
$$\begin{aligned} o[0] &= S(0) \\ o[t] &= S(o[t-1]) \end{aligned}$$

So, by induction on t we have $o[t] = S^t(0)$, where by S^t we mean $\underbrace{S \circ S \circ \ldots \circ S}_{t}$. S is monotone; thus, if there is a time k such that $S^k(0) = S^{k+1}(0)$, we have $\forall j \,.\, S^{k+j}(0) = S^k(0)$. Applying a $\mathcal{D}$ to this stream will then produce a stream that is zero almost everywhere, and integrating this result will return the last distinct value in the stream o.

This is essentially the definition of the semantics of a recursive Datalog relation: $\mathtt{O} = \text{fix}\, x.R(\mathtt{I}, x)$. □

Note that if the query R computes over unbounded data domains (e.g., using integers with arithmetic), this construction does not guarantee that at runtime a fixed point is reached. But if a program does converge, the above construction will find the least fixed point.

In fact, this circuit implements the standard **naïve evaluation** algorithm (e.g., see Algorithm 1 in [45]). Notice that the inner part of the circuit is the incremental form of another circuit, since it is sandwiched between $\mathcal{I}$ and $\mathcal{D}$ operators. Using the cycle rule of Proposition 8 we can rewrite this circuit as:



This circuit implements **semi-naïve evaluation** (Algorithm 2 from [45]). We have just proven the correctness of semi-naïve evaluation as an immediate consequence of the cycle rule!

5.2 Example recursive query

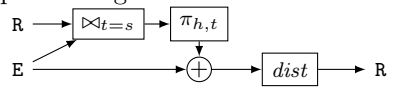
Let us apply the algorithm 5 to a concrete Datalog program, which computes the transitive closure of a directed graph:

```
// Edge relation with head and tail
input relation E(h: Node, t: Node)
// Reach relation with source s and sink t
output relation R(s: Node, t: Node)
R(x, y) :- E(x, y).
R(x, y) :- E(x, z), R(z, y).
```

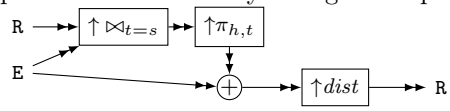
Step 1: we ignore the fact that R is both an input and an output and we implement the DBSP circuit corresponding to the body of the query. This query could be expressed in SQL as:

```
( SELECT * FROM E )
UNION
( SELECT E.h , R.t
  FROM E JOIN R ON E.t = R.s )
```

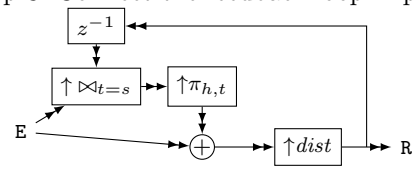
Step 1: This generates a circuit with inputs E and R:



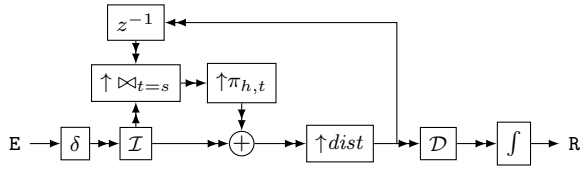
Step 2: Lift the circuit by lifting each operator:



Step 3: Connect the feedback loop implied by R:

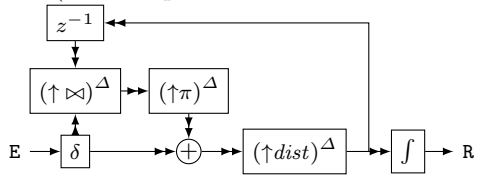


Step 4: "bracket" the circuit with $\mathcal{I}$-$\mathcal{D}$, and with δ-$\int$:

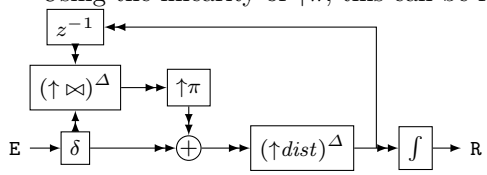


The above circuit is a complete implementation of the non-streaming recursive query; given an input relation E it will produce its transitive closure R as output.

Now we use seminaïve evaluation to rewrite the circuit (to save space we omit the indices from π and σ):



Using the linearity of $\uparrow\pi$, this can be rewritten as:



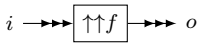
6 Incremental recursive programs

In §2–4 we showed how to incrementalize a relational query by compiling it into a circuit, lifting the circuit to compute on streams, and applying the $\cdot^{\Delta}$ operator. In §5 we showed how to compile a recursive query into a circuit that employs incremental computation internally (using semi-naïve evaluation), to compute the fixed point. Here we combine these results to construct a circuit that evaluates a *recursive query incrementally*. The circuit receives a stream of updates to input relations, and for every update recomputes the fixed point. To do this incrementally, it preserves the stream of changes to recursive relations produced by the iterative fixed point computation, and adjusts this stream to account for the modified inputs. Thus, every element of the input stream yields a stream of adjustments to the fixed point computation, using *nested streams*.

Nested streams, or streams of streams, $\mathcal{S}_{\mathcal{S}_A} = \mathbb{N} \to (\mathbb{N} \to A)$, are well defined, since streams form an abelian group. Equivalently, a nested stream is a value in $\mathbb{N} \times \mathbb{N} \to A$, i.e., a matrix with an infinite number of rows, indexed by two-dimensional time (t_0, t_1). where each row is a stream. In circuits we show nested streams with arrows with *triple* heads.

The same way we lift functions to produce stream operators, we can lift stream operators to produce operators on streams of streams.

Lifting a stream operator $S : \mathcal{S}_A \to \mathcal{S}_B$ yields an operator over nested streams $\uparrow S : \mathcal{S}_{\mathcal{S}_A} \to \mathcal{S}_{\mathcal{S}_B}$, such that $(\uparrow S)(s) = S \circ s$, or, pointwise: $(\uparrow S(s))[t_0][t_1] = S(s[t_0])[t_1], \forall t_0, t_1 \in \mathbb{N}$. In particular, a scalar function $f : A \to B$ can be lifted twice to produce an operator between streams of streams: $\uparrow\uparrow f : \mathcal{S}_{\mathcal{S}_A} \to \mathcal{S}_{\mathcal{S}_B}$.



To define recursive nested queries, we need a slightly different definition of strictness. If we think of a nested stream $F : \mathcal{S}_{\mathcal{S}_A} \to \mathcal{S}_{\mathcal{S}_B}$ as a function of timestamps (i, j), then the prior definition of strictness corresponds to strictness in the first dimension i, which we extend here to allow F to be strict in its second dimension j: for any $s, s' \in \mathcal{S}_{\mathcal{S}_A}$ and all times $t \in \mathbb{N}$, $\forall i, j < t \,.\, s[i][j] = s'[i][j]$ implies $F(s)[i][t] = F(s')[i][t]$. Proposition 2 holds for this extended notion of strictness, i.e., the fixed point operator $\mathrm{fix}\,\alpha.F(\alpha)$ is well defined for a strict operator F.

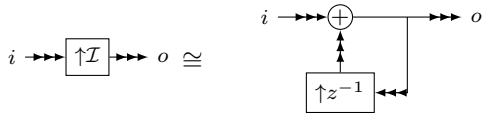
Proposition 13 *The operator $\uparrow z^{-1} : \mathcal{S}_{\mathcal{S}_A} \to \mathcal{S}_{\mathcal{S}_A}$ is strict (in its second dimension).*

The operator z^{-1} on nested streams delays "rows" of the matrix, while $\uparrow z^{-1}$ delays "columns". The $\mathcal{I}$ operator on $\mathcal{S}_{\mathcal{S}_A}$ operates on rows of the matrix, treating each row as a single value. Lifting a stream operator computing on $\mathcal{S}_A$, such as $\mathcal{I} : \mathcal{S}_A \to \mathcal{S}_A$, also produces an operator on nested streams, but this time computing on the columns of the matrix $\uparrow\mathcal{I} : \mathcal{S}_{\mathcal{S}_A} \to \mathcal{S}_{\mathcal{S}_A}$.

Proposition 14 (Lifting cycles) *For a binary, causal T we have: $\uparrow(\lambda s.\mathit{fix}\,\alpha.T(s, z^{-1}(\alpha))) =$*

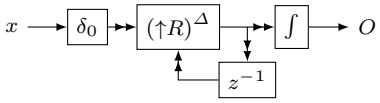
$\lambda s.fix\,\alpha.(\uparrow T)(s,(\uparrow z^{-1})(\alpha))$ *i.e., lifting a circuit containing a "cycle" can be accomplished by lifting all operators independently, including the* z^{-1} *back-edge.*

This means that lifting a DBSP stream operator can be expressed within DBSP itself. For example, we have:



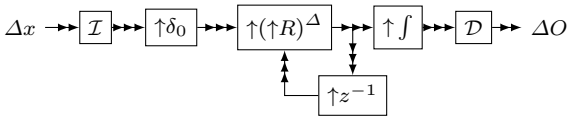
This proposition gives the ability to lift entire circuits, including circuits computing on streams and having feedback edges, which are well-defined, due to Proposition 13. With this machinery we can now apply Algorithm 4 to arbitrary circuits, even circuits built for recursively-defined relations.

Step 1: Start with the "semi-naive" circuit:

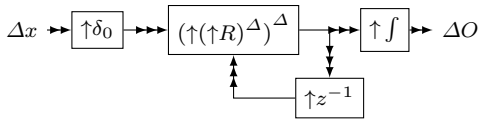


Step 2: nothing to do for *dist*.

Steps 3 and 4: Lift the circuit and incrementalize:



Step 5: use the chain rule and linearity of $\uparrow\delta_0$ and $\uparrow\int$:

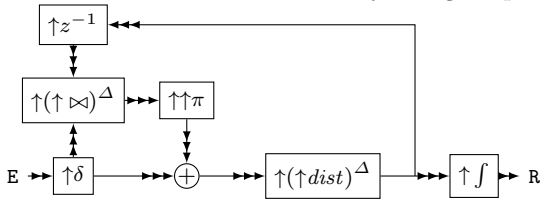


This is the incremental version of a recursive query!

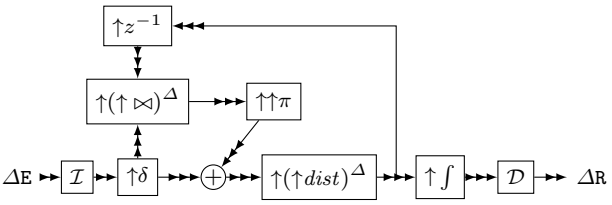
6.1 Example incrementalized recursive query

The discussion above is very abstract, so let's see a concrete example. We take the DBSP circuit for the transitive closure of a graph generated in §5.2 and convert it to an incremental circuit using algorithm 4. The resulting circuit maintains the transitive closure as edges are inserted or removed.

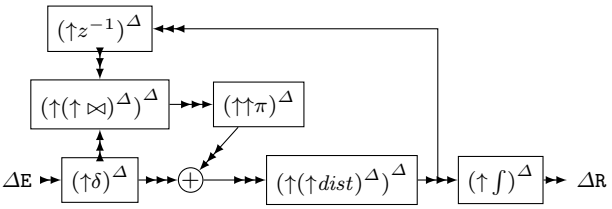
First we lift the circuit entirely, using Proposition 14:



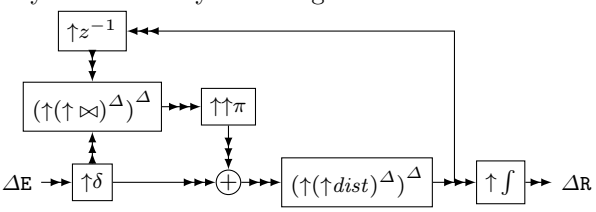
We convert this circuit into an incremental circuit, which receives in each transaction the changes to relation `E` and produces the corresponding changes to `R`:



Apply the chain and cycle rules:

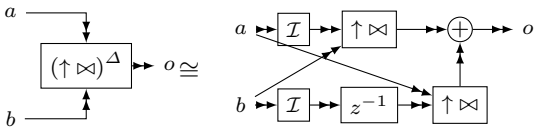


Use the linearity of $\uparrow\delta$, $\uparrow\int$, $\uparrow z^{-1}$, and $\uparrow\uparrow\pi$ to simplify the circuit by removing some $\cdot^{\Delta}$ invocations:

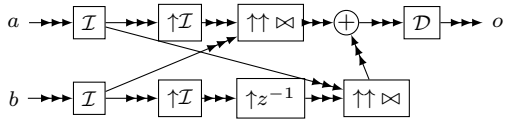


There are two applications of $\cdot^{\Delta}$ remaining in this circuit: $(\uparrow(\uparrow\bowtie)^{\Delta})^{\Delta}$ and $(\uparrow(\uparrow dist)^{\Delta})^{\Delta}$. We expand their implementations separately, and we stitch them into the global circuit at the end. This ability to reason about sub-circuits highlights the modularity of DBSP.

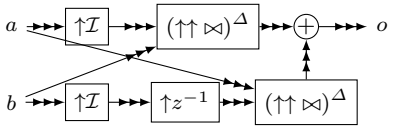
The join is expanded twice, using the bilinearity of $\uparrow\bowtie$ and $\uparrow\uparrow\bowtie$. Let's start with the inner circuit, implementing $(\uparrow\bowtie)^{\Delta}$, given by Theorem 3:



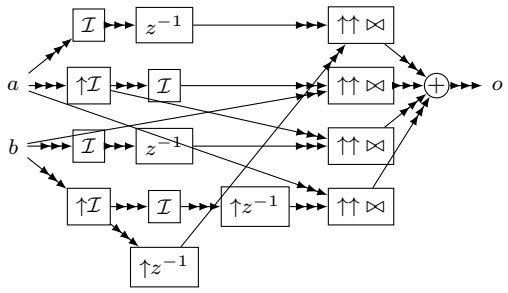
Lift and incrementalize to get the circuit for $(\uparrow(\uparrow\bowtie)^{\Delta})^{\Delta}$:



Using the chain rule and linearity of $\uparrow\mathcal{I}$ and $\uparrow z^{-1}$:

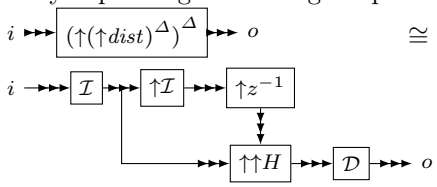


We now have two applications of $(\uparrow\uparrow\bowtie)^{\Delta}$. Each of these is the incremental form of a bilinear operator, so it in the end we will have $2\times 2 = 4$ applications of $\uparrow\uparrow\bowtie$. Here is the final form of the expanded join circuit:



(6.1)

Returning to $(\uparrow(\uparrow dist)^{\Delta})^{\Delta}$, we can compute its circuit by expanding once using Proposition 11:



Finally, stitching all these pieces together we get the final circuit shown in Figure 1.

6.2 Cost of incremental recursive queries

Time complexity.

The time complexity of an incremental recursive query can be estimated as a product of the number of fixed point iterations and the complexity of each iteration. The incrementalized circuit never performs more iterations than the non-incremental circuit: once the non-incremental circuit reaches the fixed point, its output is constant, and the derivative of corresponding value in the incrementalized circuit is 0.

Moreover, the work performed by each operator in the incremental circuit is asymptotically less than the non-incremental one. A detailed analysis can be found in our companion report [26].

Space complexity.

Integration ($\mathcal{I}$) and differentiation ($\mathcal{D}$) of a stream $\Delta s \in \mathcal{S}_{\mathcal{S}_A}$ use memory proportional to $\sum_{t_2}\sum_{t_1}|s[t_1][t_2]|$, i.e., the total size of changes aggregated over columns of the matrix. The unoptimized circuit integrates and differentiates respectively inputs and outputs of the recursive program fragment. As we move $\mathcal{I}$ and $\mathcal{D}$ inside the circuit using the chain rule, we additionally store changes to intermediate streams. Effectively we cache results of fixed point iterations from earlier timestamps to update them efficiently as new input changes arrive. Notice that space usage is proportional to the *number of iterations of the inner loop* that computes the fixed-point. Fortunately, many recursive algorithms converge in a relatively small number of steps (for example, transitive closure requires a number of steps log(graph diameter).

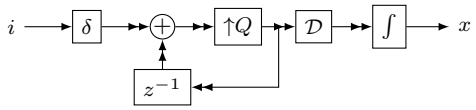
6.3 Beyond relational queries

DBSP can express programs that go beyond Datalog, SQL, or incremental versions of these: see the non-monotonic semantics for Datalog$^{\neg}$ and Datalog$^{\neg\neg}$[8]. DBSP can model many classes of streaming operators and stateful streaming computations.

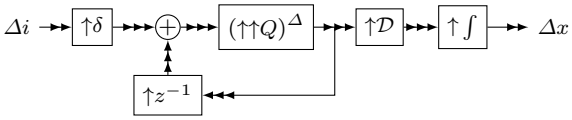
For example, to illustrate the Turing-completeness of DBSP, we implement the following *while* program, where Q is an arbitrary query:

```
x := i;
while (x changes)
    x := Q(x);
```

The DBSP implementation of this program is:



This circuit can be converted to a streaming circuit that computes a stream of values i by lifting it; it can be incrementalized using Algorithm 4 to compute on changes of i:



At runtime the execution of this circuit is not guaranteed to terminate; however, if the circuit does terminate, it will produce the correct output, i.e., the least fixpoint of Q that includes i.

7 Implementation

In this section we describe an implementation of a SQL compiler and runtime that targets DBSP. Figure 2 shows the compiler structure.

7.1 Supporting SQL

In §4 we have shown how to implement the relational algebra using DBSP. However, the SQL language is significantly richer than the relational algebra.

Multisets.

SQL operates on *multisets* (or bags), e.g., `UNION ALL`. Since $\mathbb{Z}$-sets generalize bags, they can model all SQL operations. Many queries on multisets can be implemented by just omitting *dist* operators.

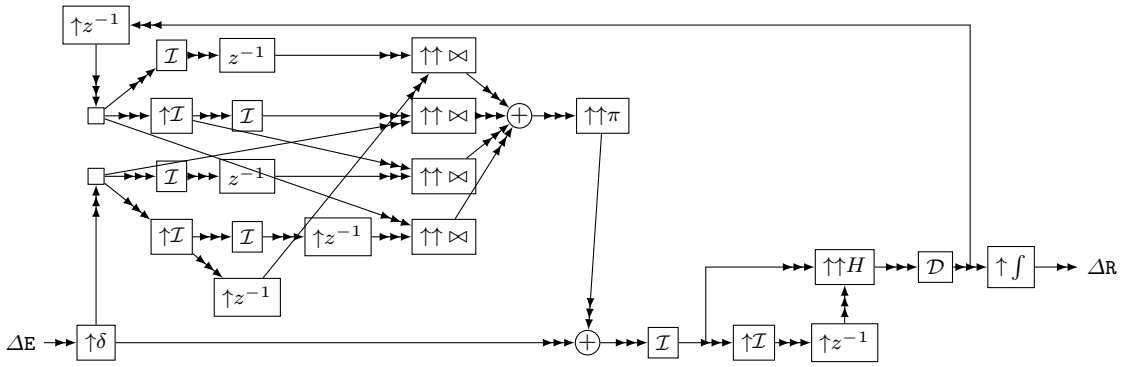


Fig. 1 Final form of circuit from §5.2 which is incrementally maintaining the transitive closure of a graph.

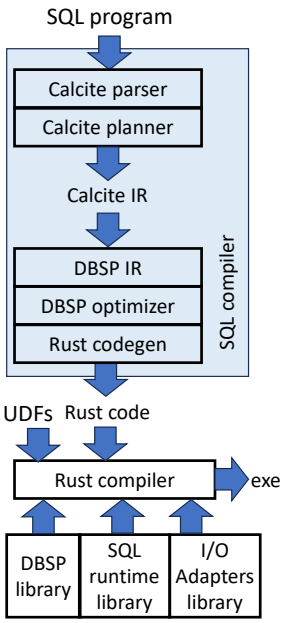


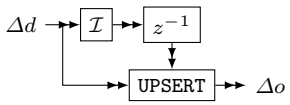
Fig. 2 Architecture of the SQL compiler.

NULLs.

DBSP says nothing about the data types and the functions that are executed by the operators in each node. In our Rust SQL runtime (described in Section §7.3) nullable types are represented using Rust `Option<>` types, and SQL `NULL` is the value `None`. Some care is required in implementing the unusual semantics of `NULL` (e.g., in SQL two `NULL` values are neither equal, nor different).

Primary keys.

A primary key changes a table's behavior on insertion: inserting a tuple can cause another tuple to be deleted. This behavior looks roughly like a modified *dist* operator, and can be implemented incrementally similarly:



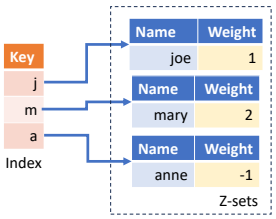
The `UPSERT` operator converts a $\mathbb{Z}$-set describing an insertion with key k: $\{(k, v) \mapsto 1\}$ to a $\mathbb{Z}$-set $\{(k, v) \mapsto 1, (k, v') \mapsto -1\}$, where (k, v') is the previous value of the record, obtained from the $\mathcal{I}$ operator.

Constant values.

One can write in SQL queries that have constant outputs, e.g., `SELECT 2`. Technically an operator that produces a constant result is not TI. However, constants can be accommodated easily by modeling them mathematically as constant input streams.

7.1.1 Grouping and indexed $\mathbb{Z}$-sets

Let K be a set of "key" values. The finite maps from K to $\mathbb{Z}$-sets are functions $K \to \mathbb{Z}[A] = \mathbb{Z}[A][K]$. We call values i of this type **indexed $\mathbb{Z}$-sets**: for each key $k \in K$, $i[k]$ is a $\mathbb{Z}$-set. Because the codomain $\mathbb{Z}[A]$ is an Abelian group, this structure is itself an Abelian group. Here is an example indexed $\mathbb{Z}$-set:



This structure is used to implement the SQL `GROUP BY` operator in DBSP. Consider a **partitioning function** $p : A \to K$ that assigns a key to any value in A. We define the grouping function $G_p : \mathbb{Z}[A] \to \mathbb{Z}[A][K]$ as $G_p(a)[k] \stackrel{\text{def}}{=} \sum_{x \in a.p(x)=k} \{x \mapsto a[x]\}$ (just map each element of the input a to the $\mathbb{Z}$-set grouping corresponding to its key). When applied to a $\mathbb{Z}$-set this function returns an indexed $\mathbb{Z}$-set: each key k maps to a $\mathbb{Z}$-set

containing all elements of the group (as in SQL, each group is a multiset). Consider our example $\mathbb{Z}$-set R from §4, and a key function $p(s)$ that returns the first letter of the string s. The resulting indexed $\mathbb{Z}$-set is shown in the previous diagram.

The operation building an indexed $\mathbb{Z}$-set from a $\mathbb{Z}$-set is linear for any key function! It follows that group-by is incremental: each row changed in the input relation produces a row changed in a group, obtained by applying the partitioning function.

Notice that, unlike SQL, DBSP can express naturally computations on indexed $\mathbb{Z}$-sets, they are just an instance of a group structure. In DBSP one does not need to follow grouping by aggregation, and DBSP can represent nested groupings of arbitrary depth. Indeed, our compiler can recognize classes of such computations expressed using SQL windows (e.g., TopK for each group), and can generate efficient incremental code.

7.1.2 UNNEST (flatmap)

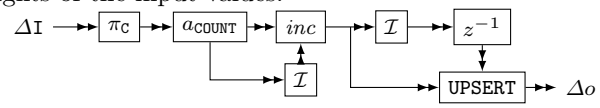
A useful operation on nested relations is **flatmap** (or UNNEST in SQL), which one can view as the inverse of grouping, converting an indexed $\mathbb{Z}$-set into a $\mathbb{Z}$-set. This operation is also a linear DBSP operator.

7.1.3 Aggregation

Aggregation in SQL applies a function a to a set of values of type A producing a "scalar" result with some result type B. In DBSP an aggregation function has a signature $a : \mathbb{Z}[A] \to B$. When operating on $\mathbb{Z}$-sets, most aggregation functions have to multiply the contribution of each value by its associated weight (but not aggregates such as MAX). DISTINCT aggregates have to first apply a DISTINCT operator.

Distributive (e.g. SUM) and some algebraic aggregation (AVG) functions can be implemented by a pair of linear functions between $\mathbb{Z}$-sets and the target group: the actual aggregation and the post-processing (e.g., dividing the sum by the counter for average). Since these are linear functions, it would seem that they can be implemented using linear operators in DBSP. However, this is not true!

The subtle point is that in SQL the result of these aggregation must be a (singleton) set containing an element of B, and not a value with type B. Thus an aggregate based on a linear function is decomposed into a linear DBSP operator, followed by additional operators, which are needed to convert the values of B into values of $\mathbb{Z}[B]$. The later operators are *not* linear. Consider the following query: SELECT COUNT(*) FROM I. The lifted incremental version of this query is the following circuit, where $a_{\texttt{COUNT}}$ is a linear operator implementing the "count" aggregation function, which just sums up the weights of the input values:



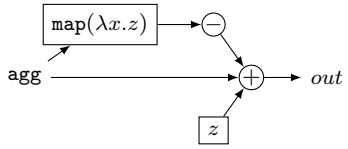
Let's say the input table I has 2 elements, and thus the previous aggregation result was 2. When adding a new row, the *inc* increments the previous aggregation result (obtained from $\mathcal{I}$) result with the current increment, and then the UPSERT converts the insertion of the tuple $\{3 \mapsto 1\}$ into the $\mathbb{Z}$-set $\{3 \mapsto 1, 2 \mapsto -1\}$, since the output set no longer contains the value 2. The *inc*, $\mathcal{I}$ and UPSERT operators only do work proportional to the size of the change, and they only store $O(1)$ state.

Aggregate functions, such as MIN, are *not* linear in the presence of deletions. The incremental form of such aggregates needs to maintain the entire input collection, using an $\mathcal{I}$ operator, similar to the *dist* operator. They can be implemented efficiently by keeping the data organized as a priority heap sorted by the value compared.

In SQL, NULL values do not participate in aggregation, so one needs two insert two extra operators for collections of nullable values:

- an operator that counts the number of rows aggregated (as described by Mumick [75]), which is used to detect empty groups
- a filtering operator (linear), which eliminates NULLs prior to aggregation

For many SQL aggregation functions aggregating over an empty set should produce a NULL result. All aggregation circuits described so far will return an empty $\mathbb{Z}$-set for an empty input. Let z be the result expected for the empty input (e.g., NULL). The following circuit, applied after aggregation, will produce the result expected by SQL:



When agg is the empty $\mathbb{Z}$-set, the "map" node produces an empty $\mathbb{Z}$-set, and the final result is just z. When agg is non-empty, the top and bottom branches of the adder cancel each other, and the result is agg. This scheme is also implemented by Materialize Inc.'s IVM engine.

7.1.4 GROUP BY-AGGREGATE

Grouping in SQL is always followed by aggregation. This can be modeled by the composition of our solutions for grouping and aggregation described above. In

this case, the `UPSERT` operator indexes data using the group key. For linear aggregates this operator needs to maintain state proportional to the number of groups. For aggregates based on non-linear functions, this maintains state proportional to the entire input collection.

7.1.5 Other operations on SQL groups

SQL constructs such as `PARTITION BY/OVER` make it possible to write queries over groups, e.g., TOP-K. These can be implemented in DBSP naturally as functions over indexed $\mathbb{Z}$-sets. Moreover, many such functions (`LAG`, `RANGE`) can be implemented using highly efficient incremental DBSP operators, which perform work proportional to the size of the change.

7.2 Formal verification

We have formalized and verified all the definitions, lemmas, propositions, theorems, and examples in this paper using the Lean theorem prover; we make these proofs available at [30]. The formalization builds on mathlib [70], which provides support for groups and functions with finite support (modeling $\mathbb{Z}$-sets). We believe the simplicity of DBSP enabled completing these proofs in relatively few lines of Lean code (5K) and keeping a close correspondence between the paper proofs in [26] and Lean. The existence of the proofs bolsters our confidence in the correctness of our implementation.

7.3 The DBSP Rust runtime

We have built an implementation of DBSP as part of an open-source [41] project with an MIT license [40]. The implementation consists of a Rust library for building circuits and a runtime that executes these circuits using a pool of worker threads.

The library provides APIs for basic algebraic data types: such as groups, finite maps, $\mathbb{Z}$-sets, indexed $\mathbb{Z}$-sets. The core data structure of the library for representing data processed is the "time-indexed, indexed $\mathbb{Z}$-set". This is the most general data structure needed in recursive circuits. It represents a vector (indexed by time) of indexed $\mathbb{Z}$-sets. Simple indexed $\mathbb{Z}$-sets are represented by a vector with a single element, while $\mathbb{Z}$-sets are represented as indexed $\mathbb{Z}$-sets with an empty value.[7] The starting point of this implementation was the differential dataflow trace data structure [71].

A circuit construction API allows users to create DBSP circuits by inserting operator nodes — boxes in our diagrams — and connecting them with streams — the arrows in our diagrams. The library provides more than 70 pre-built generic operators for integration, differentiation, delay, nested integration and differentiation, and basic $\mathbb{Z}$-set incremental operators, corresponding to plus, negation, grouping, joining, semi-joins, anti-joins, temporal joins, primitive aggregates, generic aggregates (fold), *dist*, flatmap, window aggregates, indexing, upsert, etc. Some operators only exist in an pure incremental form (e.g., they only operate correctly when fed deltas), and thus, when used in a non-incremental circuit, have to be "inverted" using the inversion property from Proposition 8.

For iterative computations the library provides the δ operator and an operator that generalizes $\int$ by terminating iteration when all the operators in the corresponding circuit cycle have reached a fixed point. The low level library allows users to construct incremental circuits manually by stitching together incremental and non-incremental versions of primitive operators.

The library also provides many "helper" operators that are used by the code generator in the implementation of some streaming queries, e.g., for state garbage-collection (a subject not discussed in this paper).

7.3.1 Parallelization and Scale-out

Besides computing on streams, DBSP circuits look very much like other dataflow query engines. As such, all standard parallelization algorithms described in the literature [44] can be applied. Our core circuits library automatically parallelizes each circuit by sharding each operators to execute using a specified number of worker threads (all operators use the same number of threads, which is statically-defined). The library automatically inserts exchange operators to re-shard data when necessary (e.g., shard on the common key in an equi-join). The same scheme can be used to implement scale-out solutions across multiple machines, but this part of the runtime is still under development.

7.3.2 State management

Incremental computation is not free. It is in fact a trade-off between time and space. While many incremental query primitives are "stateless", some important classes of database operations, including joins, *dist*, and group-by-aggregate use $\mathcal{I}$ operators in their incremental expansion. This state is kept in *indexes*. (In the DBSP theoretical model the state is stored in delay operators z^{-1} and $\uparrow z^{-1}$, but these are always inside integrators $\mathcal{I}$.) All other operators are stateless.

[7] Rust is very efficient at eliding empty data structures.

The size of these indexes is proportional to the size of the total input data of these operators — and thus the total state of a circuit can even exceed the size of the original database. (Many traditional IVM schemes opt to recompute this state on demand, rather than store it permanently; DBSP can model this strategy.)

At runtime, linear operators are essentially free. The performance of a DBSP program is given by the cost of maintaining and accessing the indexes.

Indexes provide two essential operations:

- Merging an existing (large) index with a new (small) change.
- Looking up a (small) set of values.

Our implementation of indexes performs both these operations in amortized time $O(k \log n)$, where k is the size of the changes, and n is the size of the index. The implementation of indexes is essentially a Log-Structured Merge (LSM) Tree [80].

The data structure used for indexes, (called a *trace* in the code), is shown in Figure 3. Each index is represented as a sequence of sorted immutable lists (called *batches*), of exponentially increasing sizes (1, 2, 4, 8, etc.) — a generalization of binary signed digits [3]. Any batch is sorted lexicographically, first on the index, then on the value. Each batch is a $\mathbb{Z}$-set, and a trace represents the $\mathbb{Z}$-set that is the sum of all component batches. The same tuple can appear in different batches with different weights, but appears at most once in each batch. Since addition is commutative and associative, parts can be added in any order.

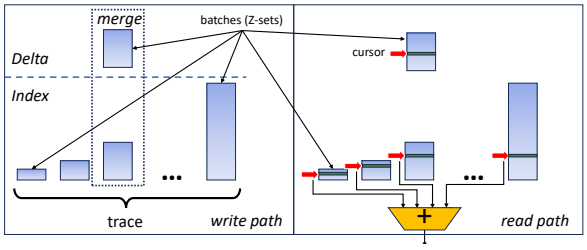


Fig. 3 Index representation and access.

A change added to an index is represented as a single batch, sorted using the same order as the index. When the index ingests a new change (left side of Figure 3), the new batch is simply appended. As the index grows, batches of similar size are lazily merged. Merges are performed by background compaction threads; each merge may span multiple circuit steps. The result of merging 2 batches with n_1 and n_2 tuples has $n_1 + n_2$ or fewer tuples, and can even be empty if all weights add to 0.

Exponential search [21] is used to lookup the tuples of a change inside an index (right side of Figure 3) . When the same tuple is found in multiple lists, the corresponding weights are added. This is how the operator H works (from the implementation of *dist* in Proposition 11).

7.4 Secondary storage

Since indexes can be very large, in many applications they need to be spilled to disk. Persistent storage also helps for fault tolerance.

Initially we considered reusing an existing storage engine for persisting state. Using RocksDB [39] seemed a great choice due to its architectural similarities to the way we manage state in-memory. RocksDB is mature, widely used, and well-maintained software.

RocksDB is a generic key-value store that can represent multiple indexes using its column family feature, with distinct namespaces for keys. It also offers all the APIs we needed: quick value retrieval for a given key and iteration over keys and values (both forward and backward) from a starting point. RocksDB is also based on an LSM-Tree.

The mature RocksDB Rust library provides all needed operations, including custom comparators for keys, zero-copy get operations, bulk inserts, and control over merging of entries with the same key during compaction.

Integrating RocksDB into our system was straightforward. We implemented the trace API described in the previous section. Unfortunately, we encountered several critical issues:

Lack of Scaling.

Our implementation utilizes multiple threads effectively by sharding data, allowing the system to scale well across many CPUs. To avoid contention, we placed each persistent index in a separate column family in RocksDB. However, we discovered that RocksDB doesn't scale well beyond a few threads. In fact, with RocksDB, our pipelines performed best with a single thread.

Figure 4 illustrates the severity of the issue (note the log-scale on the y-axis). It shows the performance for a subset of the Nexmark queries that use indexes, comparing RocksDB with a single thread against RocksDB with eight threads. For reference, we also include the performance of our system configured to keep everything in DRAM data structures (which, as expected, performs much better).

Unable to Leverage Zero-Copy Deserialization.

Besides the scalability limitations, we also observed significant overheads even when running on a single thread. This was primarily due to the cost of deseri-

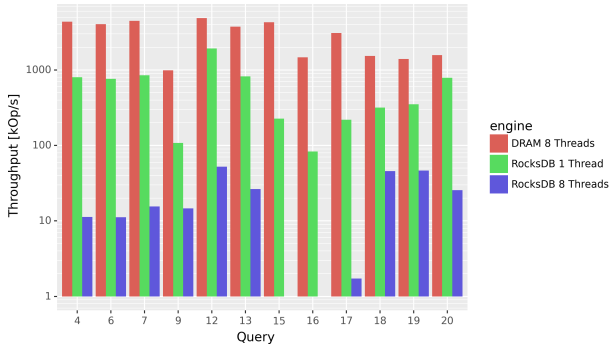


Fig. 4 Query throughput using RocksDB as a storage layer; higher is better. Note the logarithmic Y axis.

alizing keys and values from RocksDB (which stores data as byte arrays) into the corresponding Rust types.

Overwhelming Configuration Complexity.

RocksDB offers an overwhelming number of configuration options, making it nearly impossible for non-experts to ensure optimal settings. The complexity is so significant that [87] resorted to training a large language model (LLM) to identify good configurations.

Our attempts did not result in substantial improvements in performance or scalability. The most effective adjustment was enabling BlobDB, which increased throughput by approximately 20%.

Slow Tests Due to Column Families.

Our core engine is tested using property-based testing, and therefore runs the same unit-tests with thousands of different inputs. When these tests required an index, RocksDB would quickly generate thousands of short-lived column families (one for each instantiated test). This caused our test suite to slow down significantly, extending the total run time from around 2 minutes to approximately 30 minutes. We traced this issue to a known performance degradation in RocksDB when creating many column families, which is unresolved since 2019.

Since we could not find a suitable pre-existing storage system, the remaining option was to build our own. Building a key-value embedded database is a substantial endeavor, so we did not make this choice lightly.

For this purpose, we implemented our own SStable-like [31] file format. The traces write each batch that is large enough to an individual file. Batches are always created in sorted order, which allowed us to write these files sequentially without any seeks and with minimal in-memory buffering. Because batches are never modified in-place, the file format and the code that implements it does not need to make allowances for adding, removing, or modifying data.

Moreover, the implementation of storage extends the in-memory shared-nothing architecture: each worker thread processes an independent stream of data, so the storage layer can be per-thread as well.

7.4.1 Checkpointing and fault-tolerance

The state in delay z^{-1} operators is the only piece of information that needs to be persisted, checkpointed, or migrated to make DBSP computations fault-tolerant. Since DBSP circuits operate synchronously in steps, by checkpointing the state between two execution steps one obtains a consistent snapshot of the circuit's state. There is no need for a complicated synchronization protocol. Since the index data structures are immutable, taking a snapshot can be done atomically using copy-on-write. To complete a checkpoint we just need to ensure that each snapshot is written on secondary storage.

7.5 SQL compiler

We have built a compiler that accepts SQL programs and generates Rust programs targeting the DBSP library. The architecture of the compiler is shown in Figure 2. The implementation follows Algorithm 4 very closely. The input of the algorithm is a non-incremental query plan, produced by a query planner. The algorithm produces an incremental plan that is "similar" to the input plan.

The compiler front-end, including the parser, validator, and the plan generator, are based on the Apache Calcite [19] infrastructure. Because the incrementalization algorithm starts from a standard, non-incremental query plan, it can reuse in principle any existing planner. We rely on Calcite to decorrelate queries into joins, for optimizing join ordering and performing a host of traditional optimizations.

A relational algebra query can be implemented by multiple plans, each with a different data-dependent cost. Standard query planners use cost-based heuristics and data statistics to optimize plans. A generic IVM planner many not have this luxury, since the plan sometimes must be generated *before* any data has been fed to the query. Nevertheless, all standard query optimization techniques, perhaps based on historical statistics, can be used to generate the initial query plan.

The compiler can compile any number of views; each view can depend on any number of tables or other views. Given a query Q, the compiler can generate both incremental and non-incremental circuits (Q and Q^{Δ}). The non-incremental circuits are used for validating the

compiler, because they must have the same semantics as a standard ad-hoc SQL query.

The compiler supports "standard" SQL and is mature enough to pass 5+ million SQL Logic Tests [4].

- **types:** `NULL`s using the standard SQL ternary logic, all standard SQL datatypes (including `DATE`/`TIME`/`TIMESTAMP`), structured and semi-structured types, such as arrays, maps, JSON, multisets, user-defined types.
- **operators:** `SELECT`, `WHERE`, `FILTER`, `HAVING`, `ORDER BY`, `LIMIT`, `DISTINCT`, `EXCEPT`, `INTERSECT`, `UNION`, `GROUP BY`, aggregation, inner and outer `JOIN`s, `PIVOT`, `ROLLUP`, `CUBE`, temporal `ASOF JOIN`, windows (`PARTITION BY ... OVER`), `UNNEST`, table functions, common table expressions, correlated subqueries, and some streaming extensions, like tumbling and hopping windows, etc.
- A large assortment of SQL functions, including user-defined functions in SQL and Rust.

The compiler does not yet support recursive queries (although the runtime does).

SQL has many constructs that may cause runtime exceptions, such as arithmetic overflows; in a traditional ad-hoc query system these would surface as queries that terminate with an error. Currently these would also cause a running pipeline to fail completely, but this solution is unacceptable for a platform for long-running computations. We are exploring alternative solutions, where a runtime crash would block the pipeline, giving a chance to the operators to remove incorrect input data that causes issues.

7.6 Interacting with the outside world

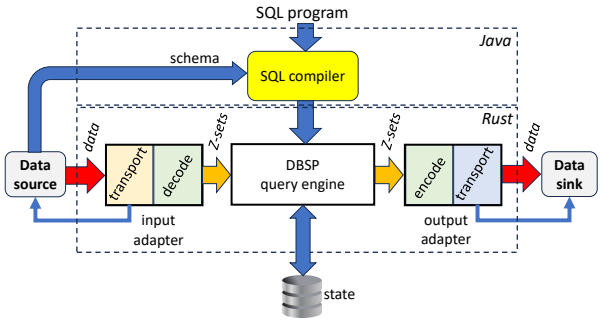


Fig. 5 Communicating with external data sources and sinks.

As noticed many years ago [66], database systems are not designed to interact well with external IVM systems. We provide a variety of adapters for interacting with external data sources, both as inputs and outputs (e.g., Kafka [65], Amazon S3 [81], Google Pub/-Sub [43], DataFrames [92], Delta Lake [16], database CDC streams via Debezium [2], http, etc.), and using many data formats (CSV, JSON, Arrow, Avro, Parquet, etc.). Figure 5 shows how a circuit uses adapters to communicate with the outside world. The input adapters receive data from external sources, buffer it, convert it into $\mathbb{Z}$-sets, and feed it to the circuits. The output adapters receive data from the circuit outputs in the form of $\mathbb{Z}$-sets, and send it to a downstream consumer in a suitable format.

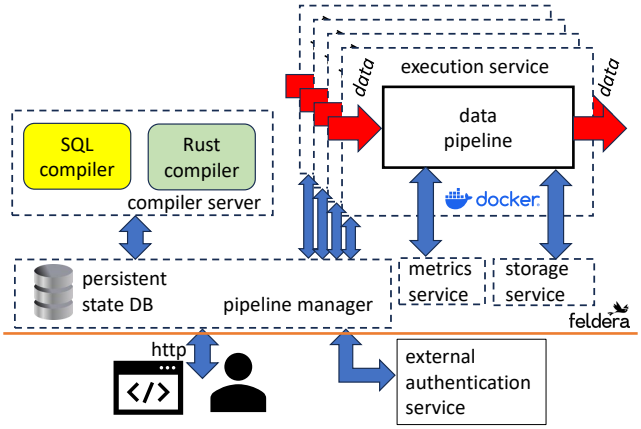


Fig. 6 Feldera Service architecture.

Feldera as a company offers IVM as a service. A version of the service-oriented architecture of the company's cloud offering is shown in Figure 6. The pipeline manager is the centralized control plane, which is responsible for managing the entire life-cycle of the IVM programs. These programs are deployed as *pipelines* that run in isolated Docker containers. Feldera also offers a cloud form factor, which distributes the pipeline manager across several communicating services and uses Kubernetes to run the pipelines.

8 Experimental Evaluation

In this section we quantify some aspects of our implementation. Looking at Figure 5, it is clear that data crosses many layers. In this section we evaluate only the performance of the central block, the DBSP query engine. However, in most real-life application performance will be limited by the adapters and network.

8.1 Latency, throughput, and input change size

Latency is the time between submitting a change and obtaining a result. Observed latency is a function of both query complexity and the size of the internal state, so latency will change as system state grows.

For relatively simple queries, as described in Section §8.2, while running in steady state, the latency of a transaction changing a single input row is in the order of tens to hundreds of microseconds, proving that our engine can be used for very low latency applications.

Throughput is the number of records that can be processed in a time unit. DBSP is synchronous and blocking: for every input change, the pipeline does not accept any other inputs until it has produced the output for all views. This suggests that latency is the inverse of throughput.

There is an additional degree of freedom: the size of an input transaction. In several scenarios there is a choice: (1) when a pipeline is started and is ingesting the initial state of a large database (*backfilling*), or (2) when processing data from streaming sources, without clear transaction boundaries.

As Nikolic has observed before [77], there is a relatively tight relationship between the latency of updating a view, the throughput, and the size of the input changes. Nikolic finds that in DBToaster the optimal value is somewhere between 1K and 10K tuples. Our experiments confirm this. The exact optimum value depends on the query and data distribution. Figure 7 shows some typical measurements for Nexmark query `q5`. Latency grows monotonically with input batch size, but the optimum throughput is obtained for batches of 2K-20K records.

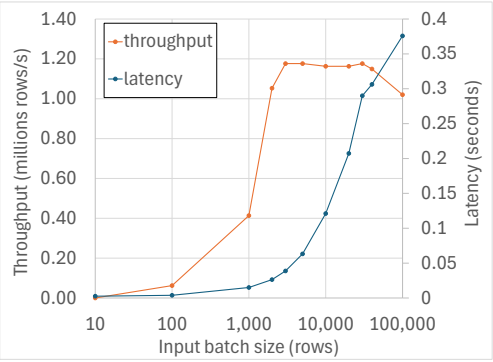


Fig. 7 Latency and throughput as a function of the input batch size. Notice the logarithmic X axis.

Increasing the batch size further beyond 100K records causes the system to enter an unstable state (the crossover point depends on the available memory size), in which throughput oscillates, as shown in Figure 8. One reason this happens is that some activities, such as merging batches, and garbage collection of useless records, only happen between circuit steps. The right solution to smooth these oscillations is probably a dynamic controller for input sizes, but also for partitioning resources like memory between DBSP operators and their shards.

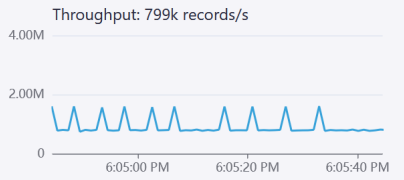


Fig. 8 Unstable throughput for very large input batch sizes.

8.2 Macrobenchmarks

There are no standard benchmark suites for IVM. In this section we use an atypical benchmark suite, Nexmark [89], which was designed for benchmarking streaming systems. The benchmark is driven by a synthetic data generator, modeling an online auction site along with a suite of queries against the streams. Nexmark is already implemented for other streaming database systems, notably for Flink [28,68], a widely used stream processing system. Nexmark is an unusual benchmark, since the data is append-only, and thus grows unbounded. The required internal state would also grow unbounded if the input is assumed to be arbitrary. However, given some weak assumptions about the ordering of the inputs, these queries *can* be implemented using finite state using a garbage-collection mechanism that deletes internal state which cannot influence any future outputs. We leave this subject for a future paper. This kind of benchmarks can only be implemented using a DBSP-like model of computation, where the output is a stream of changes (and not the full views, which would also grow unbounded).

We compare DBSP against Flink on the Nexmark benchmark, which consists of 23 queries. We use the queries that the Flink and DBSP implementations have in common. We omitted `q6` because there was no Flink implementation, and `q10` because we could not make Flink's implementation for it work. We omitted `q11` and

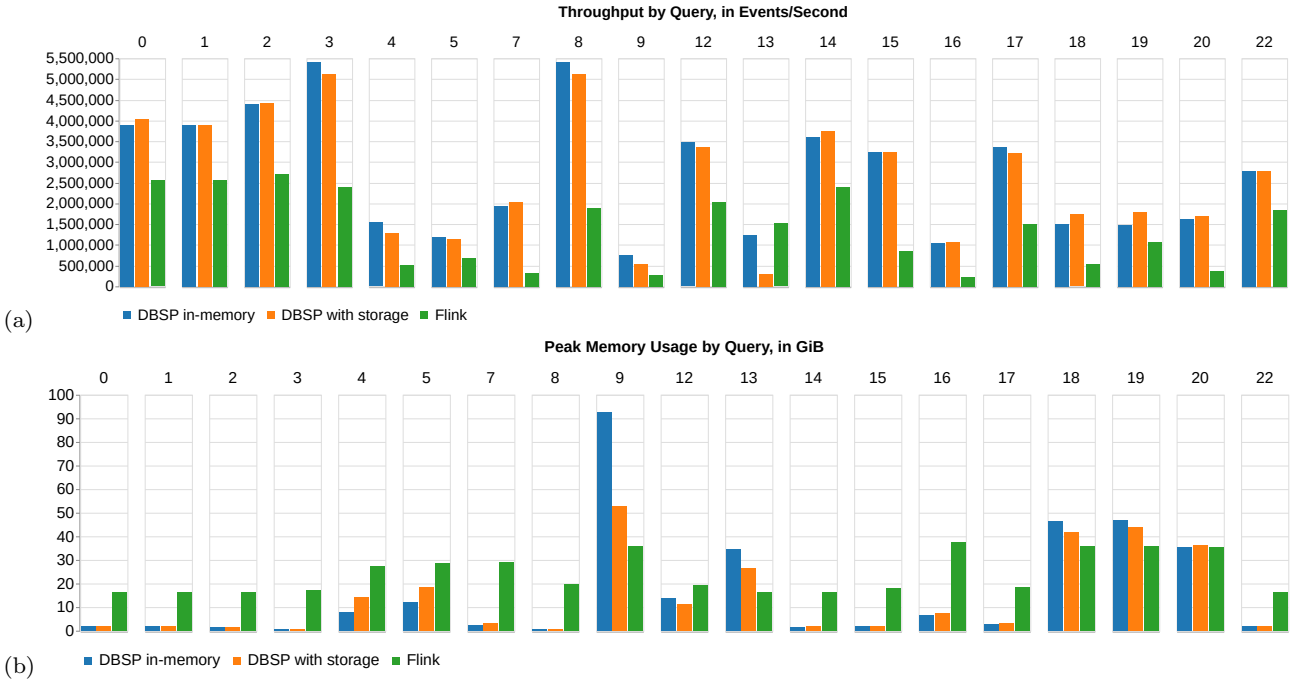


Fig. 9 (a) Average throughput, in events processed per second (higher is better), and (b) peak memory consumption (lower is better), in GiB (2^{30} bytes). The workload comprises the Nexmark queries that DBSP and Flink support in common, over 100,000,000 events.

`q21` because DBSP does not yet fully support session windows and user-defined Rust functions.[8]

We ran both the Flink and DBSP implementations on the same machine, which has a 64-core, 128-thread Threadripper 3990X CPU and 256 GB RAM, with Fedora Core 40 as the operating system. We present results for 100 million Nexmark events (input records), which is a moderate number.

DBSP runs as a single process with 16 worker threads, and otherwise with default settings, matching the number of workers used for Flink. We ran DBSP both with storage disabled, where DBSP keeps all state in RAM, and with storage enabled, where DBSP flushes large batches to secondary storage (see 7.3.2). Enabling storage allows DBSP to work with more state with less memory use, at some cost in throughput.

We configured the Flink implementation of Nexmark with the settings recommended by the upstream project, running 8 Flink task manager containers, each allocated 2 cores, and one Flink job manager container. We tried adjusting Flink and Nexmark settings, but none of these changes improved Flink performance in a significant and reproducible way.

DBSP and Flink support reading input from multiple kinds of data sources. For these measurements, we configured both of them to use their own integrated Nexmark event generators, rather than pulling them from Kafka or HTTP or another source. This eliminated network service performance and configuration as a possible source of variability.

Figure 9 reports our measurements. The two bars for DBSP in each case report results with and without enabling secondary storage.

Throughput.

Figure 9(a) shows throughput of DBSP versus Flink. With storage disabled, DBSP is up to 6.2× faster than Flink, with a geometric mean of 2.2× faster on average. As queries get more complicated, DBSP's advantage over Flink grows by a much larger factor. For example, `q7` is 6.2× faster in DBSP vs. Flink, and q16 and q20 are about 4.5× faster.

`q13` is an outlier that performs slightly slower in our system than Flink; with storage enabled, it is slower than Flink. We are investigating this behavior. With `q13` excluded, every remaining query runs at least 1.4× faster in Feldera (with or without storage).

Storage generally has a small impact on our throughput, except for `q13`, where it has about a 4× penalty. `q0` and other very simple queries are about 1.5× faster.

Peak memory.

Figure 9(b) shows peak memory consumption, as reported as the operating system resident set size (RSS),

[8] The latter feature is expected to be ready soon.

for the Flink or DBSP processes. In our case this is a single process; for Flink, it is the sum of the RSS in the 8 task manager containers. In the measurements we did not include the cost incurred by the control plane in either case (in Feldera's system, the pipeline manager; in Flink, the job manager).

DBSP uses between 0.03× and 2.6× as much memory of Flink, with a geometric mean of 0.24×.

q0 and several other queries use 2 GiB or less memory with our implementation, but over 17 GiB with Flink. These queries are linear, and do not require any state, or only minimal state, so it does not allocate much memory. Flink runs under the Java Virtual Machine, which might cause it to allocate a high minimum amount of memory.

Most queries use less memory in our system than in Flink. q9 and q13 use substantially more, and q18 and q19 use somewhat more, and storage reduces the RAM use significantly.

9 Related work

Incremental view maintenance [27,22,52,32,51,33] is a much studied problem in databases. A survey of results for Datalog queries is present in [73]. The standard approach is as follows: given a query Q, discover a "delta query", a "differential" version ΔQ that satisfies the equation: $Q(d + \Delta d) = Q(d) + \Delta Q(d, \Delta d)$, and which can be used to compute the change for a new input reusing the previous output. DBToaster introduced recursive recursive IVM [9,62,77], where the incrementalization process is repeated for the delta query. Our definition of IVM is subtly different from the above one, as IVM is defined as a stream computation, well-defined for any query.

[20] describes an early implementation in Oracle 8, which handles a limited set of queries. Many custom algorithms were published for various classes of queries: e.g. [48,67] for various classes of joins, [63] for positive nested relational calculus, [53] for relational and aggregate operators; [60] is optimized for triangle queries; DYN [56,57,58] focuses on acyclic conjunctive queries: instead of keeping the output view materialized they build data structures that allow efficiently querying the output views. PAI maps [7] are specially designed for queries with correlated aggregations. [82] discusses non-distributive aggregate functions. [61] uses primary key information to compress the representation of the deltas, and using an "update" operator that is similar to our "upsert" operator. AJU [91] and [86] focus on using foreign key information to optimize query plan generation. These techniques are only sound in the absence of deletions and updates; our implementation uses these optimizations as well. Some algorithms apply to sets, some work for multisets [49]. Many of these formalisms look very complicated because they deal with "insertions", "deletions", and "update" changes separately. $\mathbb{Z}$-sets are a much more compact tool for describing such algorithms. In some sense the DBSP theory, through the chain rule, enables us to reuse all of these results (and any future schemes designed for particular classes of subqueries): given a good implementation strategy for a particular query plan it can be reused as a subplan in any query which uses that particular plan.

DBSP as described implies an "eager" execution model: it constantly maintains the entire contents of any number of views, even if no one really wants to inspect the views. In contrast, "lazy" models [54] only build part of the views when the views are inspected. Such models have the potential to be more efficient. A simple way to implement a "lazy" model using DBSP is to essentially accumulate all input changes as $\mathbb{Z}$-sets and apply the incremental algorithm only when the output view is queried. Between "lazy" and "eager" one can place "snapshot" views, which are updated periodically [35]. Snowflake offers all these models [10].

DBSP is a bottom-up system, which always produces eagerly the *changes* to the output views. Instead of maintaining the output view entirely, DBSP proposes generating deltas as the output of the computation (similar to the kSQL [59] `EMIT CHANGES` queries). The idea that both inputs and outputs to an IVM system are streams of changes seems trivial, but this is key to the symmetry of our solution: both in our definition of IVM (3.1), and the fundamental reason that the chain rule exists — the chain rule is the one that makes our structural induction IVM algorithm possible.

Several IVM algorithms for Datalog-like languages use counting based approaches [38,74] that maintain the number of derivations of each output fact: DRed [52] and its variants [29,93,85,64,69,14], the backward-forward algorithm and variants [74,55,73]. DBSP is more general, and our incrementalization algorithm handles arbitrary recursive queries and generates more efficient plans for recursive queries in the presence of arbitrary updates (especially deletions, where competing approaches may over-delete). Interestingly, the $\mathbb{Z}$-sets weights in DBSP are related to the counting-number-of-derivations approaches, but our use of the *dist* operator shows that precise counting is not necessary.

Picallo et al. [13] provide a general solution to IVM for rich languages. Unlike their proposal, DBSP requires a group structure on the values operated on; this assumption has two major practical benefits: it simplifies the mathematics considerably (e.g., Picallo uses monoid actions to model changes), and it provides a general, simple algorithm for incrementalizing arbitrary

programs. The downside of DBSP is that one has to find a suitable group structure (e.g., $\mathbb{Z}$-sets for sets) to "embed" the computation. Picallo's notion of "derivative" is not unique: they need creativity to choose the right derivative definition, we need creativity to find the right group structure.

Finding a suitable group structure has proven easy for relations (both [62] and [46] use $\mathbb{Z}$-sets to uniformly model data and insertions/deletions), but it is not obvious how to do it for other data types, such as sorted collections, or tree-shaped collections (e.g., XML or JSON documents) [42]. An intriguing question is "what other interesting group structures could this be applied to besides $\mathbb{Z}$-sets?" Papers such as [78] explore other possibilities, such as matrix algebra, linear ML models, or conjunctive queries.

DBSP can also model window and stream database queries [15,5] such as CQL queries. [18] proposes using SQL to express both standard database queries and streaming queries; it also proposes some extensions to SQL specific to streaming systems. The DBSP theory allows us to more precisely understand the classes of queries that cannot be expressed in SQL. A SQL query is a function of the state of the database; in other words, a SQL query cannot provide different results based on the order of insertions of tuples in a table. Streaming systems however can. DBSP also enables us to handle generalizations of the proposed concepts – for example, we believe that the "timestamps" attached by streaming systems to "events" do not need any special treatment.

[23] implemented a verified IVM algorithm for a particular class of graph queries called Regular Datalog, with an implementation machine-checked in the Coq proof assistant. Their focus is on a particular algorithm and the approach does not consider other SQL operators, general recursion, or custom operators (although it is modular in the sense that it works on any query by incrementalizing it recursively). Furthermore, for all queries a deletion in the input change stream requires running the non-incremental query to recover. We formally verify the theorems in our paper, which are much broader in scope, but not our implementations.

DBSP is also related to Differential Dataflow (DD) [72, 76,34] and its theoretical foundations [6]. DD's computational model is more powerful than DBSP, since it models time values as part of an arbitrary lattice. In fact, DD is the only other framework which we are aware of that can incrementalize recursive queries as efficiently as DBSP does. In contrast, our model uses either "linear" times, or nested time dimensions via the modular lifting transformer ($\uparrow$). DBSP can express both incremental and non-incremental computations. Most importantly, DBSP comes with Algorithm 4, a syntax-directed translation that can convert any expressible query into an incremental version — in DD users have to assemble incremental queries manually using incremental operators. materialize.com offers a product that automates incrementalization for SQL queries based on DD. The Differential Datalog [84] project compiles Datalog to DD. Unlike DD, DBSP is a modular theory, which easily accommodates the addition of new operators: as long as we can express a new operator as a DBSP circuit, we can (1) define its incremental version, (2) apply the incrementalization algorithm to obtain an efficient incremental implementation, and (3) be confident that it composes with any other operators.

Many custom streaming systems have been implemented: Storm [88], Spark Streaming [94], Flink [28], Samza [79], Beam [11], Kafka Streams [90], and many have adopted SQL dialectsL Spark Structured Streaming [17], Spark SQL, KSQL [59]. These sytems usually sacrifice some of the nice properties of database systems in order to compute efficiently over unbounded streams. These systems may only support restricted classes of queries. We believe that in the future databases will incorporate the best features of streaming systems, and that DBSP shows one way this can be achieved.

[10,12] describe the Snowflake incremental and streaming capabilities. In Showflake "streams" are database table that store the history of changes to a table. Dynamic tables are views which are periodically refreshed, at user-specified intervals. These can be updated either incrementally or using batch recomputation; the system chooses a stragegy based on the refres period. The views provide snapshot isolation, which is similar to the DBSP consistency model.

The `pg_ivm` [50] project ofers an open-source Postgres module which adds IVM capabilities. The supported set of queries has some significant restrictions.

The DBSP model is simple enough so it can be implemented in a few hundred lines of Python [36].

10 Conclusions

10.1 Adoption

Traditional databases could in principle be retrofitted to use the algorithms in this paper, but the existing query engines are not built around structures that can represent negative changes (like $\mathbb{Z}$-sets), so this effort will require a significant redesign.

Moreover, we argue that databases should not only compute views incrementally, but should use "changes" as the fundamental data structure to communicate with their environment: a database service should offer the following API: users register to receive notifications for

changes in one or more views. Then, for any transaction committed, each user receives a notification containing the list of changes for the all the views they registered. Databases today do not have convenient mechanism for reporting changes to the outside world. In fact, entire industries have sprung up around the concept of Change Data Capture [1], which is building ad-hoc solutions for extracting changes from databases, usually by inspecting the write-ahead transaction log.

10.2 Summary

We have introduced DBSP, a model of computation based on infinite streams over commutative groups. In this model streams are used for 3 different purposes: (1) to model consecutive snapshots of a database, (2) to model consecutive changes (deltas, or transactions) applied to a database and changes of a maintained view, (3) to model consecutive values of loop-carried variables in recursive computations.

We have defined an abstract notion of incremental computation over streams, and defined the incrementalization operator $\cdot^{\Delta}$, which transforms an *arbitrary* stream computation Q into its incremental version Q^{Δ}. The incrementalization operator has some very nice algebraic properties, which gave us a general algorithm for incrementalizing many classes of complex queries, including arbitrary recursive queries.

We believe that DBSP can form a solid foundation for a theory and practice of streaming incremental computation. As a proof, we have built a SQL compiler that can essentially incrementalize arbitrary queries.

References

1. https://en.wikipedia.orag/wiki/Change_data_capture. Retrieved March 2024.
2. Debezium. https://debezium.io/. Retrieved September 2024.
3. Signed-digit representation. https://en.wikipedia.org/wiki/Signed-digit_representation.
4. sqllogictest. https://www.sqlite.org/sqllogictest/doc/trunk/about.wiki. Last accessed March 2023.
5. The Aurora project. http://cs.brown.edu/research/aurora/, 2004. Retrieved September 2024.
6. Martín Abadi, Frank McSherry, and Gordon Plotkin. Foundations of Differential Dataflow. In *Foundations of Software Science and Computation Structures (FoSSaCS)*, London, UK, April 11-18 2015.
7. Supun Abeysinghe, Qiyang He, and Tiark Rompf. Efficient incrementialization of correlated nested aggregate queries using relative partial aggregate indexes (RPAI). In *ACM SIGMOD International conference on Management of data (SIGMOD)*, page 136–149, 2022.
8. Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
9. Yanif Ahmad and Christoph Koch. DBToaster: A SQL compiler for high-performance delta processing in main-memory databases. *Proc. VLDB Endow.*, 2(2):1566–1569, August 2009.
10. Tyler Akidau, Paul Barbier, Istvan Cseri, Fabian Hueske, Tyler Jones, Sasha Lionheart, Daniel Mills, Dzmitry Pauliukevich, Lukas Probst, Niklas Semmler, Dan Sotolongo, and Boyuan Zhang. What's the difference? Incremental processing with change queries in Snowflake. *Proc. ACM Manag. Data*, 1(2), jun 2023.
11. Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, aug 2015.
12. Tyler Akidau, Fabian Hueske, Konstantinos Kloudas, Leon Papke, Niklas Semmler, and Jan Sommerfeld. Continuous data ingestion and transformation in Snowflake. In *Proceedings of the 18th ACM International Conference on Distributed and Event-Based Systems*, DEBS '24, page 195–198, 2024.
13. Mario Alvarez-Picallo, Alex Eyers-Taylor, Michael Peyton Jones, and C.-H. Luke Ong. Fixing incremental computation. In *European Symposium on Programming Languages and Systems (ESOP)*, pages 525–552, Prague, Czech Republic, April 6–11 2019.
14. Krzysztof R. Apt and Jean-Marc Pugin. Maintenance of stratified databases viewed as a belief revision system. In *ACM SIGMOD International conference on Management of data (SIGMOD)*, pages 136–145, San Diego, California, March 23-25 1987.
15. Arvind Arasu, Shivnath Babu, and Jennifer Widom. An abstract semantics and concrete language for continuous queries over streams and relations. Technical Report 2002-57, Stanford InfoLab, 2002.
16. Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, Michał undefinedwitakowski, Michał Szafrański, Xiao Li, Takuya Ueshin, Mostafa Mokhtar, Peter Boncz, Ali Ghodsi, Sameer Paranjpye, Pieter Senster, Reynold Xin, and Matei Zaharia. Delta lake: high-performance ACID table storage over cloud object stores. *Proc. VLDB Endow.*, 13(12):3411–3424, aug 2020.
17. Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. Structured Streaming: A declarative API for real-time applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 601–613, 2018.
18. Edmon Begoli, Tyler Akidau, Fabian Hueske, Julian Hyde, Kathryn Knight, and Kenneth Knowles. One SQL to rule them all — an efficient and syntactically idiomatic approach to management of streams and tables. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1757–1772, 2019.
19. Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. Apache Calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *International Conference on Management of Data (IDMD)*, page 221–230, 2018.
20. Randall G. Bello, Karl Dias, Alan Downing, James J. Feenan Jr., James L. Finnerty, William D. Norcott, Harry Sun, Andrew Witkowski, and Mohamed Ziauddin. Materialized views in Oracle. In *Proceedings of 24rd International Conference on Very Large Data Bases (VLDB'98)*, pages 659–664, August 24-27 1998.

21. Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82–87, 1976.
22. Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. Efficiently updating materialized views. In *ACM SIGMOD International conference on Management of data (SIGMOD)*, SIGMOD '86, page 61–71, 1986.
23. Angela Bonifati, Stefania Dumbrava, and Emilio Jesús Gallego Arias. Certified graph view maintenance with regular Datalog. *Theory and Practice of Logic Programming*, 18(3-4):372–389, 2018.
24. Mihai Budiu, Tej Chajed, Frank McSherry, Leonid Ryzhyk, and Val Tannen. DBSP: Automatic incremental view maintenance for rich query languages. In *Proceedings of the VLDB Endowment (VLDB)*, volume 16, pages 1601–1614, Vancouver, Canada, August 2023.
25. Mihai Budiu, Tej Chajed, Frank McSherry, Leonid Ryzhyk, and Val Tannen. DBSP: Incremental computation on streams and its applications to databases. *SIGMOD Research Highlights*, 53, March 2024. A shorter and simpler version of the VLDB 2023 DBSP paper.
26. Mihai Budiu, Frank McSherry, Leonid Ryzhyk, and Val Tannen. DBSP: A language for expressing incremental view maintenance for rich query languages. https://github.com/feldera/feldera/blob/main/doc/spec.pdf, December 2022.
27. O. Peter Buneman and Eric K. Clemons. Efficiently monitoring relational databases. *ACM Trans. Database Syst.*, 4(3):368–382, sep 1979.
28. Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.
29. Stefano Ceri and Jennifer Widom. Deriving production rules for incremental view maintenance. In *International Conference of Very Large Data Bases (VLDB)*, pages 577–589, Barcelona, Spain, 1991.
30. Tej Chajed. DBSP formalization, December 2022.
31. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), jun 2008.
32. Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. Optimizing queries with materialized views. In *International Conference on Data Engineering (ICDE)*, page 190–200, 1995.
33. Rada Chirkova and Jun Yang. *Materialized Views*. Now Publishers Inc., Hanover, MA, USA, 2012.
34. Zaheer Chothia, John Liagouris, Frank McSherry, and Timothy Roscoe. Explaining outputs in modern data analytics. *Proc. VLDB Endow.*, 9(12):1137–1148, August 2016.
35. Latha S. Colby, Akira Kawaguchi, Daniel F. Lieuwen, Inderpal Singh Mumick, and Kenneth A. Ross. Supporting multiple view maintenance policies. *SIGMOD Rec.*, 26(2):405–416, jun 1997.
36. Bruno Rucy Carneiro Alves de Lima. PyDBSP. https://github.com/brurucy/pydbsp, September 2024.
37. Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover. In *International Conference on Automated Deduction (CADE-25)*, Berlin, Germany, 2015.
38. Hasanat M. Dewan, David Ohsie, Salvatore J. Stolfo, Ouri Wolfson, and Sushil Da Silva. Incremental database rule processing in PARADISER. *J. Intell. Inf. Syst.*, 1(2):177–209, 1992.
39. Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. RocksDB: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Trans. Storage*, 17(4), oct 2021.
40. Feldera Inc. DBSP Rust crate. https://crates.com/crates/dbsp. Retrieved March 2024.
41. Feldera Inc. Feldera repository. https://github.com/feldera/feldera. Retrieved August 2024.
42. J. Nathan Foster, Ravi Konuru, Jerome Simeon, and Lionel Villard. An algebraic approach to XQuery view maintenance. In *ACM SIGPLAN Workshop on Programming Languages Technologies for XML*, San Francisco, CA, January 9 2008.
43. Google. Google pub/sub. https://cloud.google.com/pubsub?hl=en. Retrieved September 2024.
44. Goetz Graefe. Encapsulation of parallelism in the Volcano query processing system. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, SIGMOD '90, page 102–111, 1990.
45. Sergio Greco and Cristian Molinaro. Datalog and logic databases. *Synthesis Lectures on Data Management*, 7(2):1–169, 2015.
46. Todd J Green, Zachary G Ives, and Val Tannen. Reconcilable differences. *Theory of Computing Systems*, 49(2):460–488, 2011.
47. Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Symposium on Principles of Database Systems (PODS)*, page 31–40, Beijing, China, June 11-14 2007.
48. Timothy Griffin and Bharat Kumar. Algebraic change propagation for semijoin and outerjoin queries. *SIGMOD Rec.*, 27(3):22–27, sep 1998.
49. Timothy Griffin and Leonid Libkin. Incremental maintenance of views with duplicates. In *ACM SIGMOD International conference on Management of data (SIGMOD)*, page 328–339, 1995.
50. IVM Development Group. pg_ivm. https://github.com/sraoss/pg_ivm. Retrieved September 2024.
51. Ashish Gupta, Inderpal Singh Mumick, et al. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.
52. Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *ACM SIGMOD International Conference on Management of Data*, page 157–166, Washington, D.C., USA, 1993.
53. Himanshu Gupta and Inderpal Singh Mumick. Incremental maintenance of aggregate and outerjoin expressions. *Inf. Syst.*, 31(6):435–464, sep 2006.
54. Eric N. Hanson. A performance analysis of view materialization strategies. *SIGMOD Rec.*, 16(3):440–453, dec 1987.
55. John V. Harrison and Suzanne W. Dietrich. Maintenance of materialized views in a deductive database: An update propagation approach. In *Workshop on Deductive Databases*, Technical Report, pages 56–65, Washington, D.C., November 14 1992.
56. Muhammad Idris, Martin Ugarte, and Stijn Vansummeren. The dynamic Yannakakis algorithm: Compact and efficient query processing under updates. In *ACM SIGMOD International conference on Management of data (SIGMOD)*, page 1259–1274, 2017.
57. Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. Conjunctive queries with inequalities under updates. *Proc. VLDB Endow.*, 11(7):733–745, mar 2018.

58. Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. Efficient query processing for dynamically changing datasets. *SIGMOD Rec.*, 48(1):33–40, November 2019.
59. Hojjat Jafarpour, Rohan Desai, and Damian Guy. KSQL: Streaming SQL engine for Apache Kafka. In *International Conference on Extending Database Technology (EDBT)*, pages 524–533, Lisbon, Portugal, March 26-29 2019.
60. Ahmet Kara, Hung Q. Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Maintaining triangle queries under updates. *ACM Trans. Database Syst.*, 45(3), aug 2020.
61. Yannis Katsis, Kian Win Ong, Yannis Papakonstantinou, and Kevin Keliang Zhao. Utilizing ids to accelerate incremental view maintenance. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1985–2000, 2015.
62. Christoph Koch. Incremental query evaluation in a ring of databases. In *Symposium on Principles of Database Systems (PODS)*, page 87–98, Indianapolis, Indiana, USA, 2010.
63. Christoph Koch, Daniel Lupei, and Val Tannen. Incremental view maintenance for collection programming. In *Symposium on Principles of Database Systems (PODS)*, page 75–90, San Francisco, California, USA, 2016.
64. Jakub Kotowski, François Bry, and Simon Brodt. Reasoning as axioms change — incremental view maintenance reconsidered. In *Web Reasoning and Rule Systems RR*, volume 6902 of *Lecture Notes in Computer Science*, pages 139–154, Galway, Ireland, August 29-30 2011. Springer.
65. Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of NetDB*, number 2011, pages 1–7, Athens, Greece, 2011.
66. Wilburt Labio, Jun Yang, Yingwei Cui, Hector Garcia-Molina, and Jennifer Widom. Performance issues in incremental warehouse maintenance. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, page 461–472, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.
67. Per-Ake Larson and Jingren Zhou. Efficient maintenance of materialized outer-join views. In *International Conference on Data Engineering (ICDE)*, pages 56–65, 2007.
68. Jingsong Lee, Jark Wu, et al. Nexmark benchmark. https://github.com/nexmark/nexmark, August 2024.
69. James J. Lu, Guido Moerkotte, Joachim Schü, and V. S. Subrahmanian. Efficient maintenance of materialized mediated views. In *ACM SIGMOD International conference on Management of data (SIGMOD)*, pages 340–351, San Jose, California, May 22-25 1995.
70. The mathlib Community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pages 367–381, New York, NY, USA, 2020. Association for Computing Machinery.
71. Frank McSherry. Differential Dataflow rust library. https://github.com/TimelyDataflow/differential-dataflow. Retrieved September 2024.
72. Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential Dataflow. In *Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, January 6–9 2013.
73. Boris Motik, Yavor Nenov, Robert Piro, and Ian Horrocks. Maintenance of Datalog materialisations revisited. *Artif. Intell.*, 269:76–136, 2019.
74. Boris Motik, Yavor Nenov, Robert Edgar Felix Piro, and Ian Horrocks. Incremental update of Datalog materialisation: the backward/forward algorithm. In *Conference on Artificial Intelligence (AAAI)*, pages 1560–1568, Austin, Texas, January 25-30 2015.
75. Inderpal Singh Mumick, Dallan Quass, and Barinderpal Singh Mumick. Maintenance of data cubes and summary tables in a warehouse. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, SIGMOD '97, page 100–111, 1997.
76. Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *ACM Symposium on Operating Systems Principles (SOSP)*, page 439–455, Farminton, Pennsylvania, 2013.
77. Milos Nikolic, Mohammad Dashti, and Christoph Koch. How to win a hot dog eating contest: Distributed incremental view maintenance with batch updates. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 511–526, 2016.
78. Milos Nikolic and Dan Olteanu. Incremental view maintenance with triple lock factorization benefits. In *International Conference on Management of Data (ICMD)*, page 365–380, 2018.
79. Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. Samza: stateful scalable stream processing at LinkedIn. *Proc. VLDB Endow.*, 10(12):1634–1645, aug 2017.
80. Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33:351–385, 1996.
81. Mayur R. Palankar, Adriana Iamnitchi, Matei Ripeanu, and Simson Garfinkel. Amazon S3 for science grids: a viable solution? In *Proceedings of the 2008 International Workshop on Data-Aware Distributed Computing*, DADC '08, page 55–64, 2008.
82. Themistoklis Palpanas, Richard Sidle, Roberta Cochrane, and Hamid Pirahesh. Incremental maintenance for non-distributive aggregate functions. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, page 802–813. VLDB Endowment, 2002.
83. L. R. Rabiner and B. Gold, editors. *Theory and Application of Digital Signal Processing*. Prentice-Hall, 1975.
84. Leonid Ryzhyk and Mihai Budiu. Differential Datalog. In *Datalog 2.0*, Philadelphia, PA, June 4-5 2019.
85. Martin Staudt and Matthias Jarke. Incremental maintenance of externally materialized views. In *International Conference of Very Large Data Bases (VLDB)*, pages 75–86, Mumbai (Bombay), India, September 3-6 1996.
86. Christoforos Svingos, Andre Hernich, Hinnerk Gildhoff, Yannis Papakonstantinou, and Yannis Ioannidis. Foreign keys open the door for faster incremental view maintenance. *Proc. ACM Manag. Data*, 1(1), may 2023.
87. Viraj Thakkar, Madhumitha Sukumar, Jiaxin Dai, Kaushiki Singh, and Zhichao Cao. Can modern LLMs tune and configure LSM-based key-value stores? In *Proceedings of the 16th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '24, page 116–123, 2024.
88. Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 147–156, 2014.

89. Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. Nexmark–a benchmark for queries over data streams (draft). *Technical report*, 2008.
90. Guozhang Wang, Lei Chen, Ayusman Dikshit, Jason Gustafson, Boyang Chen, Matthias J. Sax, John Roesler, Sophie Blee-Goldman, Bruno Cadonna, Apurva Mehta, Varun Madan, and Jun Rao. Consistency and completeness: Rethinking distributed stream processing in Apache Kafka. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 2602–2613, 2021.
91. Qichen Wang and Ke Yi. Maintaining acyclic foreign-key joins under updates. In *ACM SIGMOD International conference on Management of data (SIGMOD)*, page 1225–1239, Portland, OR, USA, 2020.
92. McKinney Wes. *Python for data analysis*. O'Reilly Media, Inc., 1 edition, 2012.
93. Ouri Wolfson, Hasanat M. Dewan, Salvatore J. Stolfo, and Yechiam Yemini. Incremental evaluation of rules and its relationship to parallelism. In *ACM SIGMOD International conference on Management of data (SIGMOD)*, pages 78–87, Denver, Colorado, May 29-31 1991. ACM Press.
94. Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *ACM Symposium on Operating Systems Principles (SOSP)*, page 423–438, Farminton, Pennsylvania, 2013.