DBSP: Automatic Incremental View Maintenance for Rich Query Languages

Mihai Budiu
VMware Research
mbudiu@vmware.com

Tej Chajed
VMware Research
tchajed@vmware.com

Frank McSherry
Materialize Inc.
mcsherry@materialize.com

Leonid Ryzhyk
VMware Research
lryzhyk@vmware.com

Val Tannen
University of Pennsylvania
val@seas.upenn.edu

ABSTRACT

Incremental view maintenance (IVM) has long been a central problem in database theory. Many solutions have been proposed for restricted classes of database languages, such as the relational algebra, or Datalog. These techniques do not naturally generalize to richer languages. In this paper we give a general, heuristic-free solution to this problem in 3 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 and a general algorithm for solving IVM for arbitrary DBSP programs, and (3) we show how to model many rich database query languages using DBSP (including the full relational algebra, queries over sets and multisets, arbitrarily nested relations, aggregation, flatmap (unnest), monotonic and non-monotonic recursion, streaming aggregation, and arbitrary compositions of all of these). SQL and Datalog can both be implemented in DBSP. As a consequence, we obtain efficient incremental view maintenance algorithms for queries written in all these languages.

PVLDB Reference Format:
Mihai Budiu, Tej Chajed, Frank McSherry, Leonid Ryzhyk, and Val Tannen. DBSP: Automatic Incremental View Maintenance for Rich Query Languages. PVLDB, 16(7): 1601-1614, 2023.
doi:10.14778/3587136.3587137

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/vmware/database-stream-processor.

1 INTRODUCTION

Incremental view maintenance (IVM) is an important and well-studied problem in databases [25]. The IVM problem can be stated as follows: given a database DB and a view V described by a query Q that is a function of the database, i.e. $V = Q(DB)$, maintain the contents of V in response to changes of the database, ideally more efficiently than by simply reevaluating $Q(DB)$ from scratch. We want an algorithm to evaluate Q over the *changes* ΔDB applied to the database, since often changes are small $|\Delta DB| \ll |DB|$.

This paper provides a new perspective by proposing a new definition of IVM based on a streaming model of computation. Our model is inspired by Digital Signal Processing DSP [44], applied to databases, hence the name DBSP. Whereas previous IVM solutions are based on defining a notion of a (partial) derivative of Q with respect to its inputs, our definition only requires computing *derivatives of streams* as functions of time. Derivatives of streams are always well-defined if the data computed on has a notion of difference that satisfies some simple mathematical properties — specifically, that it forms a commutative group. (Fortunately, relational databases can be modeled in such a way [23, 33].)

DBSP has several attractive properties:

(1) it is **expressive**. (a) It can be used to define precisely multiple concepts: traditional queries, streaming computations, and incremental computations. (b) We have been able to express in DBSP the full relational algebra, computations over sets and bags, nested relations, aggregation, flatmap (unnest), monotonic and nonmonotonic recursion, stratified negation, while-relational programs, window queries, streaming queries, streaming aggregation, and incremental versions of all of the above. In fact, we have built a DBSP implementation of the complete SQL language (§8).

(2) it is **simple**. DBSP has only 4 operators, and it is built entirely on elementary concepts such as functions and algebraic groups.

(3) mathematically **precise**. All the results in this paper have been formalized and checked using the Lean proof assistant [18].

(4) it is **modular**, in the following two ways: (a) the incremental version of a complex query can be reduced recursively to incrementalizing its component subqueries. This gives a simple, syntactic, heuristic-free algorithm (Algorithm 4.6) that converts an arbitrary DBSP query plan to its incremental form. (b) Extending DBSP to support new primitive operators is easy, and they immediately benefit from the rest of the theory of incrementalization. An important consequence of modularity is that the theory can be efficiently implemented, as we briefly discuss in §8.

The core concept of DBSP is the *stream*, which is used to model changes over time. We use $\mathcal{S}_A$ to denote the type of infinite streams with values of type A. If $s \in \mathcal{S}_A$ is a stream, then $s[t] \in A, t \in \mathbb{N}$ is the t-th element of s, also referred to as the *value of the stream at time t*. A streaming operator is a function that consumes one or more streams and produces another stream. We show streaming computations with diagrams, also called "circuits", where boxes are computations and streams are arrows. The following diagram shows a stream operator $T : \mathcal{S}_A \times \mathcal{S}_B \to \mathcal{S}_C$, consuming two input streams s_0 and s_1 and producing one output stream s:

Proceedings of the VLDB Endowment, Vol. 16, No. 7 ISSN 2150-8097.
doi:10.14778/3587136.3587137

$$\begin{tikzcd}
s_0 \arrow[dr] & & \\
& \boxed{T} \arrow[r] & s \\
s_1 \arrow[ur] & &
\end{tikzcd}$$

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 $DB \in \mathcal{S}_{SCH}$, where SCH is the database schema. Time is not wall-clock time, but counts the transactions applied to the database. Since transactions are linearizable, they have a total order. $DB[t]$ is the snapshot of the database contents after t transactions have been applied.

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 is 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: $DB[t] = \sum_{i \leq t} \Delta DB[i]$. The operation $\mathcal{I}$, adding up all changes, is another basic stream operator, is *stream integration*, the inverse of differentiation. The following diagram shows the relationship between the streams ΔDB and DB:

$$\Delta DB \to \boxed{\mathcal{I}} \to DB \to \boxed{\mathcal{D}} \to \Delta DB$$

Suppose we are given a query $Q : SCH \to SCH$ defining a view V. What is a view in a streaming model? It is also a stream! For each snapshot of the database stream we have a snapshot of the view: $V[t] = Q(DB[t])$. In general, given an arbitrary function $f : A \to B$, we define a streaming "version" of f, denoted by $\uparrow\!f$ (read as "f lifted"), which applies f to every element of the input stream independently. We can thus write $V = (\uparrow\!Q)(DB)$.

Armed with these basic definitions, we can precisely define IVM. What does it mean to maintain a view incrementally? An efficient maintenance algorithm needs to compute 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: $Q^{\Delta} \stackrel{\text{def}}{=} D \circ \uparrow\!Q \circ I$. The incremental version of the query maintains the changes to the view $\Delta V \stackrel{\text{def}}{=} \mathcal{D}(V) = \mathcal{D}(\uparrow\!Q(\mathcal{I}(\Delta DB)))$, depicted graphically as:

$$\Delta DB \to \boxed{\mathcal{I}} \xrightarrow{DB} \boxed{\uparrow\!Q} \xrightarrow{V} \boxed{\mathcal{D}} \to \Delta V$$

The incremental version of a query is a stateful *streaming operator* which computes directly on changes and produces changes. The incremental version of a query is thus always well-defined. The above definition shows one way to compute a query incrementally, but applying it naively produces an inefficient execution plan, since it will reconstruct the database at each step. In §3 we show how algebraic properties of the $\cdot^{\Delta}$ transformation are used to optimize the implementation of Q^{Δ}. The first key property is that the incremental composition of subqueries is the composition of incremental subqueries: $(Q_1 \circ Q_2)^{\Delta} = {Q_1}^{\Delta} \circ {Q_2}^{\Delta}$. The second key property is that essentially all primitive database operations have efficient incremental versions. More precisely, they are faster than non-incremental versions by a factor of $O(|DB|/|\Delta DB|)$.

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. §5.1 shows how stratified-monotonic recursive Datalog programs can be implemented in DBSP, and §6 gives *incremental streaming computations for recursive programs*. 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 the graph is changed by adding and deleting edges.

In this paper we omit proofs, they can be found in an extensive companion technical report [12]. We have formalized this theory in the Lean proof assistant [14]; our formalization includes machine-checked proofs for all the theorems in this paper.

This paper makes the following contributions:

(1) DBSP, a **simple** but **expressive** language for streaming computation. DBSP gives an elegant formal foundation unifying the manipulation of streaming and incremental computations.
(2) An algorithm for incrementalizing any streaming computation expressed in DBSP.
(3) An illustration of how DBSP can model various query classes, such as relational algebra, nested relations, aggregations, and stratified-monotonic Datalog.
(4) The first general and machine-checked theory of IVM.
(5) A high-performance open-source implementation of DBSP as a general-purpose streaming query engine in Rust.

2 STREAM COMPUTATIONS

The core notion of our theory of IVM is the **stream**. In this section we introduce formally streams as infinite sequences of values, and define computations on streams. Stream operators (§2.1) are the basic building block of stream computations. Operators can be composed with simple rules (§2.2) into complex computational circuits. In (§2.3) we introduce two essential operations on streams: integration and differentiation.

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

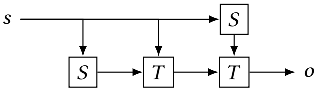
When $s \in \mathcal{S}_A$ and $t \in \mathbb{N}$ we write $s[t]$ for the t-th element of the stream s instead of the usual $s(t)$. We think of the index $t \in \mathbb{N}$ as (discrete) time and of $s[t] \in A$ as the value of the the stream s "at time" t. For example, the stream of natural numbers $id \in \mathcal{S}_{\mathbb{N}}$ given by $id[t] = t$ is the sequence of values $[\ 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad \cdots\ ]$.

Definition 2.2 (stream operator)**:** A **stream operator** with n inputs is a function $T : \mathcal{S}_{A_0} \times \cdots \times \mathcal{S}_{A_{n-1}} \to \mathcal{S}_B$.

In general we use "operator" for functions on streams, and "function" for computations on "scalar" values.

DBSP is an extension of the simply-typed lambda calculus — we will introduce its elements gradually. However, in many cases we find it more readable to use circuit diagrams to depict DBSP programs. (Circuits do hide the *order* of the inputs of an operator; for non-commutative operators we have to distinguish the operator inputs.) In a circuit a rectangle represents an operator application (labeled with the operator name, e.g., T), while an arrow is a stream.

Stream operator *composition* (function composition) is shown as chained circuits. The composition of a binary operator $T : \mathcal{S}_A \times \mathcal{S}_B \rightarrow \mathcal{S}_A$ with the unary operator $S : \mathcal{S}_A \rightarrow \mathcal{S}_B$ into the computation $\lambda s.T(T(s, S(s)), S(s)) : \mathcal{S}_A \rightarrow \mathcal{S}_A$ is:

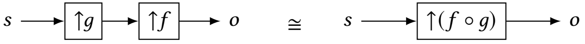


Definition 2.3: (lifting) Given a (scalar) function $f : A \rightarrow B$, we define a stream operator $\uparrow\!f : \mathcal{S}_A \rightarrow \mathcal{S}_B$ by *lifting* the function f pointwise in time: $(\uparrow\!f)(s) \overset{\text{def}}{=} f \circ s$. Equivalently, $((\uparrow\!f)(s))[t] \overset{\text{def}}{=} f(s[t])$. This extends to functions of multiple arguments.

For example, $(\uparrow\!(\lambda x.(2x)))(id) = [\begin{array}{cccccc} 0 & 2 & 4 & 6 & 8 & \cdots \end{array}]$.

Proposition 2.4 (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 2.4 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); the group structure will allow us to reorder insertions and deletions. We show later that this restriction is not a problem for using DBSP with relational data. Now we introduce some useful operators and study their properties.

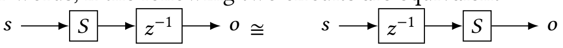
2.2.1 Delays and time-invariance.

Definition 2.5 (Delay)**:** The **delay operator**[1] produces an output stream by delaying its input by one step: $z_A^{-1} : \mathcal{S}_A \rightarrow \mathcal{S}_A$:

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

We often omit the type parameter A, and write just z^{-1}. For example, $z^{-1}(id) = [\begin{array}{cccccc} 0 & 0 & 1 & 2 & 3 & \cdots \end{array}]$.

Definition 2.6 (Time invariance)**:** A stream operator $S : \mathcal{S}_A \rightarrow \mathcal{S}_B$ 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:



This definition extends naturally to operators with multiple inputs.

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.

Definition 2.7 (Causality)**:** A stream operator $S : \mathcal{S}_A \rightarrow \mathcal{S}_B$ is **causal** when for all $s, s' \in \mathcal{S}_A$, and all times t 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 $t' \leq t$. Operators produced by lifting are causal, and z^{-1} is causal. All DBSP operators are causal. The composition of causal operators of any number of inputs is causal.

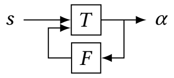
Definition 2.8 (Strictness)**:** A stream operator, $F : \mathcal{S}_A \rightarrow \mathcal{S}_B$ is **strict** if $\forall s, s' \in \mathcal{S}_A, \forall t \in \mathbb{N}$ we have: $(\forall i < t.\ s[i] = s'[i]) \Rightarrow F(s)[t] = F(s')[t]$.

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. Lifted operators in general are *not* strict. z^{-1} is strict.

Proposition 2.9: For a strict $F : \mathcal{S}_A \rightarrow \mathcal{S}_A$ the equation $\alpha = F(\alpha)$ has a unique solution $\alpha \in \mathcal{S}_A$, denoted by $\text{fix}\, \alpha.F(\alpha)$.

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:



This circuit is a well-defined function on streams:

Lemma 2.10: If $F : \mathcal{S}_B \rightarrow \mathcal{S}_B$ is strict and $T : \mathcal{S}_A \times \mathcal{S}_B \rightarrow \mathcal{S}_B$ is causal, the operator $Q(s) = \text{fix}\, \alpha.T(s, F(\alpha))$ is well-defined and causal. If, moreover, F and T are TI then so is Q.

All DBSP computations are built using just lifted functions and delays. We add two more operators in §6.

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 2.11: 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.

Stream addition and negation are causal, TI operators.

Definition 2.12: Given abelian groups A and B we call a stream operator $S : \mathcal{S}_A \rightarrow \mathcal{S}_B$ **linear** if it is a group homomorphism, that is, $S(a + b) = S(a) + S(b)$ (and therefore $S(0) = 0$ and $S(-a) = -S(a)$).

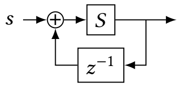
Given a linear function $f : A \rightarrow B$, the stream operator $\uparrow\!f$ is linear and TI (LTI). z^{-1} is also LTI.

Definition 2.13: (bilinear) A function of two arguments $f : A \times B \rightarrow C$ with A, B, C 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: $f : \mathcal{S}_{\mathbb{N}} \times \mathcal{S}_{\mathbb{N}} \rightarrow \mathcal{S}_{\mathbb{N}}, f(a, b)[t] = a[t] \cdot b[t]$.

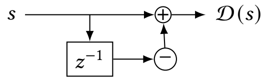
The "feedback loop" of a linear operator is linear:

Proposition 2.14: Let S be a unary, causal, LTI operator. The operator $Q(s) = \text{fix}\, \alpha.S(s + z^{-1}(\alpha))$ is well-defined and LTI:



[1] The name z^{-1} comes from the DSP literature, and is related to the z-transform [44].

Definition 2.15 (Differentiation)**:** The **differentiation operator** $\mathcal{D}_{\mathcal{S}_A} : \mathcal{S}_A \to \mathcal{S}_A$ is defined by: $\mathcal{D}(s) \stackrel{\text{def}}{=} s - z^{-1}(s)$.



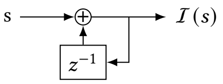
We generally omit the type, and write just $\mathcal{D}$. The value of $\mathcal{D}(s)[t] = s[t] - s[t-1]$ if $t > 0$. As an example, $\mathcal{D}(id) = [\ 0 \quad 1 \quad 1 \quad 1 \quad 1 \quad \cdots\ ]$.

If s is a stream, then $\mathcal{D}(s)$ is the *stream of changes* of s.

Proposition 2.16: $\mathcal{D}$ is causal and LTI.

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

Definition 2.17 (Integration)**:** The **integration operator** $\mathcal{I}_{\mathcal{S}_A} : \mathcal{S}_A \to \mathcal{S}_A$ is defined by $\mathcal{I}(s) \stackrel{\text{def}}{=} \lambda s.\text{fix}\,\alpha.(s + z^{-1}(\alpha))$:



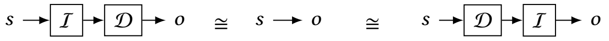
We also generally omit the type, and write just $\mathcal{I}$. This is the construction from Proposition 2.14 using the identity function for S.

Proposition 2.18: $\mathcal{I}(s)$ is the discrete (indefinite) integral applied to the stream s: $\mathcal{I}(s)[t] = \sum_{i \leq t} s[i]$.

As an example, $\mathcal{I}(id) = [\ 0 \quad 1 \quad 3 \quad 6 \quad 10 \quad \cdots\ ]$.

Proposition 2.19: $\mathcal{I}$ is causal and LTI.

Theorem 2.20 (Inversion)**:** Integration and differentiation are inverses of each other: $\forall s. \mathcal{I}(\mathcal{D}(s)) = \mathcal{D}(\mathcal{I}(s)) = s$.



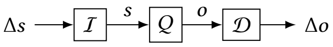
3 INCREMENTAL VIEW MAINTENANCE

Here we define IVM and analyze its properties.

Definition 3.1: 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 $T^{\Delta}(a, b) \stackrel{\text{def}}{=} \mathcal{D}(T(\mathcal{I}(a), \mathcal{I}(b)))$.



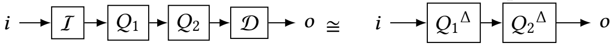
If $Q(s) = o$ is a computation, then Q^{Δ} performs the "same" computation as Q, but between streams of changes Δs and Δo.

Notice that our definition of incremental computation is meaningful only for *streaming* computations; this is in contrast to classic definitions, e.g. [25] which consider only one change. Generalizing the definition to operate on streams gives us additional power, especially when operating with recursive queries.

The following proposition is one of our central results:

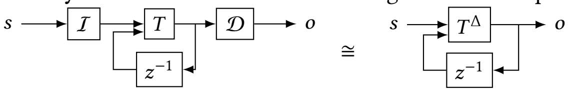
Proposition 3.2: (Properties of the incremental version):
inversion: $Q \mapsto Q^{\Delta}$ is bijective; its inverse is $Q \mapsto \mathcal{I} \circ Q \circ \mathcal{D}$.
invariance: $+^{\Delta} = +, (z^{-1})^{\Delta} = z^{-1}, -^{\Delta} = -, \mathcal{I}^{\Delta} = \mathcal{I}, \mathcal{D}^{\Delta} = \mathcal{D}$
push/pull: $Q \circ \mathcal{I} = \mathcal{I} \circ Q^{\Delta}$; $\mathcal{D} \circ Q = Q^{\Delta} \circ \mathcal{D}$
chain: $(Q_1 \circ Q_2)^{\Delta} = Q_1{}^{\Delta} \circ Q_2{}^{\Delta}$ (Generalizes to multiple inputs.)
add: $(Q_1 + Q_2)^{\Delta} = Q_1{}^{\Delta} + Q_2{}^{\Delta}$
cycle: $(\lambda s.\text{fix}\,\alpha.T(s, z^{-1}(\alpha)))^{\Delta} = \lambda s.\text{fix}\,\alpha.T^{\Delta}(s, z^{-1}(\alpha))$

The **chain rule** states that these two circuits are equivalent:



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:

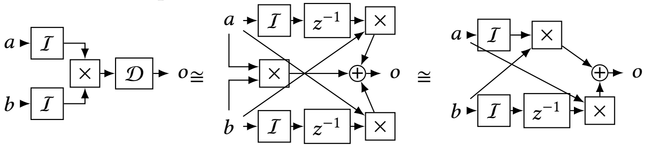


In other words, the incremental version of a feedback loop around a query is just the feedback loop with the incremental query for its body. The significance of this result will be apparent when we implement recursive queries.

To execute incremental queries efficiently, we want to compute directly on streams of changes, without integrating them. The invariance property above shows that stream operators $+$, $-$, and z^{-1} are identical to their incremental versions. The following theorems generalize this to linear and bi-linear operators:

Theorem 3.3 (Linear)**:** For an LTI operator Q we have $Q^{\Delta} = Q$.

Theorem 3.4 (Bilinear)**:** For a bilinear TI operator $\times$ we have $(a \times b)^{\Delta} = a \times b + z^{-1}(\mathcal{I}(a)) \times b + a \times z^{-1}(\mathcal{I}(b)) = \mathcal{I}(a) \times b + a \times z^{-1}(\mathcal{I}(b))$. In pictures:



Rewriting Theorem 3.4 using Δa for the stream of changes to a we get the familiar formula for incremental equi-joins: $\Delta(a \times b) = \Delta a \times \Delta b + a \times (\Delta b) + (\Delta a) \times b$; equi-joins are indeed bilinear.

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 any relational query Q that defines a database 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 [22]) 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. This translation is efficiently incrementalizable because many basic relational queries can be expressed using LTI $\mathbb{Z}$-set operators §4.2.

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

$\mathbb{Z}$-sets generalize database tables: think of a $\mathbb{Z}$-set as a 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 write that $x \in m$ iff $m[x] \neq 0$. We also write $w \cdot x$ for $\{x \mapsto w\}$.

Consider a concrete $\mathbb{Z}$-set $R \in \mathbb{Z}[\texttt{string}]$, defined by $R = \{\texttt{joe} \mapsto 1, \texttt{anne} \mapsto -1\}$. R has two elements in its domain, joe with weight 1 (so $R[\texttt{joe}] = 1$), and anne with weight -1. We say $\texttt{joe} \in R$ and $\texttt{anne} \in R$.

Since $\mathbb{Z}$ is an abelian ring, $\mathbb{Z}[A]$ is also an abelian ring (and thus a group). This group $(\mathbb{Z}[A], +_{\mathbb{Z}[A]}, 0_{\mathbb{Z}[A]}, -_{\mathbb{Z}A})$ has addition and subtraction 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, $R+R = \{\texttt{joe} \mapsto 2, \texttt{anne} \mapsto -2\}$. 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".

Definition 4.1: 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 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 isset(R) = false, since $R[\texttt{anne}] = -1$.

Definition 4.2: 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 ispositive : $\mathbb{Z}[A] \to \mathbb{B}$. given by

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

We have $\forall m \in \mathbb{Z}[A].\text{isset}(m) \Rightarrow \text{ispositive}(m)$. ispositive$(R)$ = false, since $R[\texttt{anne}] = -1$.

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: ispositive(f).

Definition 4.3 (distinct)**:** The function *distinct* : $\mathbb{Z}[A] \to \mathbb{Z}[A]$ "converts" a $\mathbb{Z}$-set into a set:

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

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

Generalizing circuit diagrams. From now on we will use circuits to compute both on scalars ($\mathbb{Z}$-sets in our case) and streams of $\mathbb{Z}$-sets. We use the same graphical representation for functions on streams or scalars: boxes with input and output arrows. For scalar functions the "values" of the arrows are scalars instead of streams; otherwise the interpretation of boxes as function application is unchanged. We will thus use circuits to depict relational query plans.

4.2 Implementing relational operators

The fact that relational algebra can be implemented by computations on $\mathbb{Z}$-sets has been shown before, e.g. [23]. 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 *distinct* **to produce sets**. For example, $a \cup b = \mathit{distinct}(a + b)$, $a \setminus b = \mathit{distinct}(a - b)$, $(a \times b)((x, y)) = a[x] \times b[y]$. Notice that the use of the *distinct* 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 [22]) has shown how some invocations of *distinct* can be eliminated from query plans without changing the query semantics; we will see that incremental versions of *distinct* operators incur significant space costs.

Proposition 4.4: Let Q be one of the following $\mathbb{Z}$-sets operators: filtering σ, join $\bowtie$, or Cartesian product $\times$. Then we have $\forall i \in \mathbb{Z}[I], \text{ispositive}(i) \Rightarrow Q(\mathit{distinct}(i)) = \mathit{distinct}(Q(i))$.

Proposition 4.5: 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 $\text{ispositive}(i) \Rightarrow \mathit{distinct}(Q(\mathit{distinct}(i))) = \mathit{distinct}(Q(i))$.

These properties allow us to "consolidate" distinct operators by performing one *distinct* at the end of a chain of computations.

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:

Algorithm 4.6 (incremental view maintenance)**:**

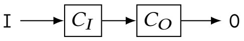
(1) Translate Q into a circuit using the rules in Table 1.
(2) Apply *distinct* elimination rules (4.4, 4.5) until convergence[3].
(3) Lift the whole circuit, by applying Proposition 2.4, converting it to a circuit operating on streams.
(4) Incrementalize the whole circuit "surrounding" it with $\mathcal{I}$ and $\mathcal{D}$.
(5) Apply the chain rule from Proposition 3.2 recursively on the query structure to obtain an incremental implementation.

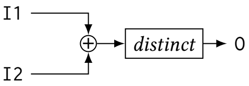
This algorithm is deterministic and its running time is proportional to the number of operators in the query. Step (2) generates an equivalent circuit, with possibly fewer *distinct* operators. Step (3) yields a circuit that consumes a *stream* of complete database snapshots and outputs a stream of complete view snapshots. Step

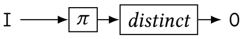
[2] Technically, map (applying a user-defined function to each row) is not relational.
[3] The order in which the rules are applied does not matter, since the algorithm is confluent: it always produces the same final result.

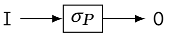
Table 1: Implementation of SQL relational set operators in DBSP. Each query assumes that inputs I, I1, I2, are sets and it produces output sets.

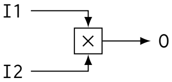
Operation	SQL example	DBSP circuit	Details

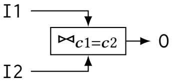
Composition	`SELECT DISTINCT ... FROM`
	`(SELECT ... FROM ...)`
			C_I circuit for inner query,
			C_O circuit for outer query.

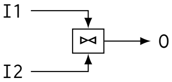
Union	`(SELECT * FROM I1)`
	`UNION`
	`(SELECT * FROM I2)`
			*distinct* eliminates duplicates. An implementation of UNION ALL does not need the *distinct*.

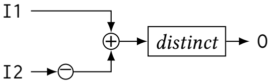
Projection	`SELECT DISTINCT I.c`
	`FROM I`
			$\pi(i)[y] \overset{\text{def}}{=} \sum_{x \in i, x\|_c = y} i[x]$
			$x\|_c$ is projection on column c of the tuple x
			π is linear; ispositive(π)

Filtering	`SELECT * FROM I`
	`WHERE p(I.c)`
			$\sigma_P(m)[x] \overset{\text{def}}{=} \begin{cases} m[x] & \text{if } P(x) \\ 0 & \text{otherwise} \end{cases}$
			$P : A \to \mathbb{B}$ is a predicate.
			σ_P is linear; ispositive(σ_P)

Cartesian product	`SELECT I1.*, I2.*`
	`FROM I1, I2`
			$(a \times b)((x, y)) \overset{\text{def}}{=} a[x] \times b[y]$.
			$\times$ is bilinear, ispositive$(\times)$

Equi-join	`SELECT I1.*, I2.*`
	`FROM I1 JOIN I2`
	`ON I1.c1 = I2.c2`
			$(a \bowtie b)((x, y)) \overset{\text{def}}{=} a[x] \times b[y]$
			if $x\|_{c1} = y\|_{c2}$.
			$\bowtie$ is bilinear, ispositive$(\bowtie)$

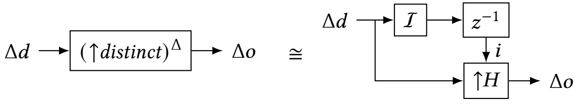
Intersection	`(SELECT * FROM I1)`
	`INTERSECT`
	`(SELECT * FROM I2)`
			Special case of equi-join when both relations have the same schema.

Difference	`SELECT * FROM I1`
	`EXCEPT`
	`SELECT * FROM I2`
			*distinct* removes elements with negative weights from the result.

(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.

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 §4.4). A notable exception is *distinct*. The next proposition shows that the incremental version of *distinct* is also efficient, and it can be computed by doing work proportional to the size of the input change:

Proposition 4.7: The following circuit implements $(\uparrow distinct)^\Delta$:



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

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

Here is the intuition why *distinct* is efficiently incrementalizable: the only elements that can appear in the output of $(\uparrow distinct)^\Delta$ must have changed in the input. So the size of the output change cannot be bigger than the size of the input change. In the diagram above, i is the previous version of the integral of all changes, i.e., the full $\mathbb{Z}$-set whose *distinct* value is being computed. The function H detects whether the weight of an element in i is changing sign (positive to negative or vice-versa) when adding a new delta d.

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

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 a function of $O(|\Delta DB[t]|)$. For example, Theorem 3.3, 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 *distinct* operator is not linear, $(\uparrow \mathit{distinct})^\Delta$ can also be evaluated in $O(|\Delta DB[t]|)$ according to Proposition 4.7. 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 $(\uparrow \mathit{distinct})^\Delta$, $(\uparrow \bowtie)^\Delta$, $\mathcal{I}$, and $\mathcal{D}$ is $O(|DB[t]|)$. They need to store their input or output relations in full.

4.4.1 IVM query plans and optimality.

Let us look again at what we achieved using Algorithm 4.6. A relational algebra query can be implemented by multiple plans, each with a different data-dependent cost[4]. The input of Algorithm 4.6 is a (relational), non-incremental query plan, produced by a query planner. The algorithm produces an incremental plan that is "similar" to the input plan.

Standard query planners use cost-based heuristics and data statistics to optimize plans. A generic IVM planner does not have this luxury, since the plan must be generated *before* any data has been fed to the database. Nevertheless, all standard query optimization techniques, perhaps based on historical statistics, can be used to generate the query plan that is supplied to Algorithm 4.6. The question of optimality in the context of IVM plan is a much more difficult topic than optimization of ad-hoc queries, since the chosen IVM plan will execute for *all future database updates.*

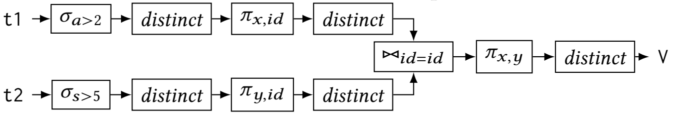
Moreover, since incremental computations maintain internal state, it follows that incremental plans cannot be simply changed in-flight, like we can change ad-hoc queries based on current data statistics: deploying a new plan requires in general constructing its internal state, which is produced by entire history of prior updates. Fortunately, there is a trivial, but somewhat expensive, recipe for installing a new incremental plan: feed the entire current state of the database, as one big change.

4.5 Relational Query Example

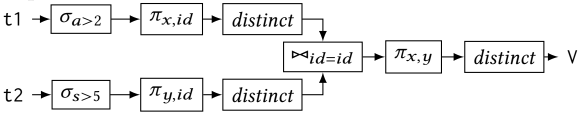
We apply the IVM algorithm 4.6 to a concrete relational SQL query:

```
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
```

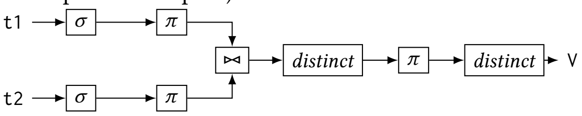
Step 1: Create a DBSP circuit to represent this query using the translation rules from Table 1; notice that this circuit is essentially a dataflow implementation of the query. (Notice that the query asks for SELECT DISTINCT, so there is a *distinct* operator after σ):



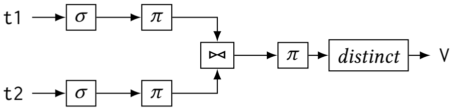
Step 2: apply the rules to eliminate *distinct* operators. First from Proposition 4.5:



The rule from Proposition 4.4 gives (from now on we omit the subscripts to save space):

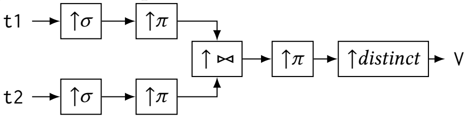


And again 4.5:

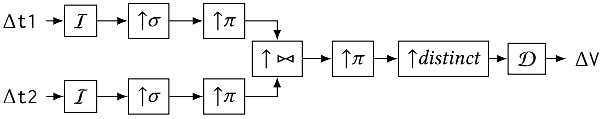


At this point no more *distinct* elimination rules can be applied.

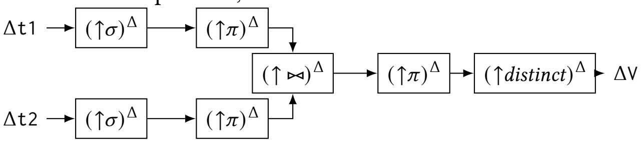
Step 3: we lift the circuit using distributivity of composition over lifting (Proposition 2.4); we obtain a circuit that computes over streams, i.e., for each new input pair of relations t1 and t2 it will produce an output view V:



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:

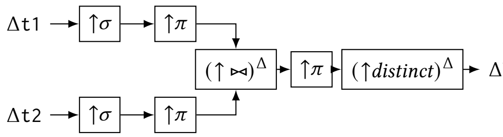


Step 5: apply the chain rule to rewrite the circuit as a composition of incremental operators;

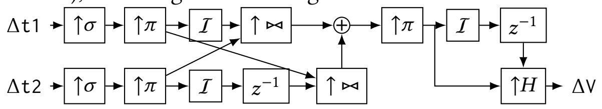


Use the linearity of σ and π to simplify this circuit (notice that all linear operators no longer have a $\cdot^\Delta$):

[4] The optimal plan depends not only on the query, but also on the data.



Finally, replace the incremental join using the formula for bilinear operators (Theorem 3.4), and the incremental *distinct* (Proposition 4.7), obtaining the following circuit:



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

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.

Definition 5.1: 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) $\delta_0 : A \rightarrow \mathcal{S}_A$ produces a stream from a scalar value:

$$\delta_0(v)[t] \stackrel{\text{def}}{=} \begin{cases} v & \text{if } t = 0 \\ 0_A & \text{otherwise} \end{cases}$$

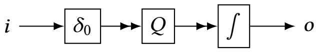
For example, $\delta_0(5)$ is the stream $[\begin{array}{cccccc} 5 & 0 & 0 & 0 & 0 & \cdots \end{array}]$.

Stream elimination. We define the function $\int : \overline{\mathcal{S}_A} \rightarrow A$, over streams that are zero almost everywhere, as $\int(s) \stackrel{\text{def}}{=} \sum_{t \geq 0} s[t]$. $\int$ 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, $\int([\begin{array}{cccccc} 1 & 2 & 3 & 0 & 0 & \cdots \end{array}]) = 6$.

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 the first 0 value encountered.

Proposition 5.2: δ_0 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_0 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_0 and an $\int$ operator:



In this circuit the arrows with double heads denote stream values, while the simple arrow denote scalar values[5]. Q is a streaming operator, but the entire circuit is a scalar function.

Algorithm 5.3 below, which translates recursive queries to DBSP circuits, always produces well-formed circuits.

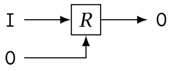
5.1 Implementing recursive queries

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

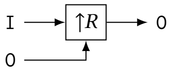
We describe the algorithm for the simpler case of a single-input, single-output query[6]. The input of our algorithm is a Datalog query of the form $O = R(I, O)$, where R is a relational, non-recursive query, producing a set as a result, but whose output O is also an input. The output of the algorithm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

Algorithm 5.3 (recursive queries)**:**

(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 a $\mathbb{Z}$-sets:

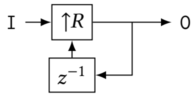


(2) Lift this circuit to operate on streams:

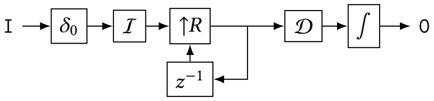


We construct $\uparrow R$ by lifting each operator of the circuit individually according to Proposition 2.4.

(3) Build a cycle, connecting the output to the corresponding recursive input via a delay:



(4) "Bracket" the circuit in $\mathcal{I}$ and $\mathcal{D}$ nodes, and then in δ_0 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 5.4 (Recursion correctness)**:** If isset(I), the output of the circuit above is the relation O as defined by the Datalog semantics of given program R as a function of the input relation I.

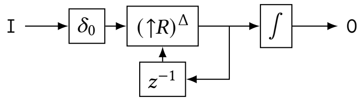
Note that if the query R computes over unbounded data domains (e.g., using integers with arithmetic), this construction does not guarantee that at runtime a fixed point is reached. But if a program

[5] We only use this convention in this diagram; in general the type of an arrow can be inferred from the type of its source node.

[6] The general case in the companion technical report [12] is only slightly more involved.

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 [21]). 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 3.2 we can rewrite this circuit as:



(5.1)

This circuit implements **semi-naïve evaluation** (Algorithm 2 from [21]). We have just proven the correctness of semi-naïve evaluation as an immediate consequence of the cycle rule!

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. Please refer to our companion report for example computations on nested streams [12].

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: $\uparrow\uparrow f : \mathcal{S}_{\mathcal{S}_A} \to \mathcal{S}_{\mathcal{S}_B}$.

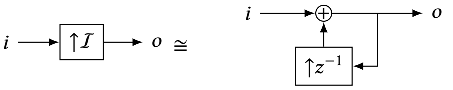
To define recursive nested queries, we need a slightly different definition of strictness. If we think of a nested stream $F : \mathcal{S}_{\mathcal{S}_A} \to \mathcal{S}_{\mathcal{S}_B}$ as a function of timestamps (i, j), then the prior definition of strictness corresponds to strictness in the first 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 2.9 holds for this extended notion of strictness, i.e., the fixed point operator $\text{fix}\,\alpha.F(\alpha)$ is well defined for a strict operator F.

Proposition 6.1: 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 $\uparrow\mathcal{I} : \mathcal{S}_{\mathcal{S}_A} \to \mathcal{S}_{\mathcal{S}_A}$.

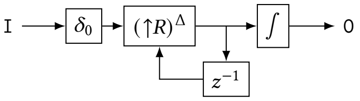
Proposition 6.2 (Lifting cycles)**:** For a binary, causal T we have: $\uparrow(\lambda s.\text{fix}\,\alpha.T(s, z^{-1}(\alpha))) = \lambda s.\text{fix}\,\alpha.(\uparrow T)(s, (\uparrow z^{-1})(\alpha))$ i.e., lifting a circuit containing a "cycle" can be accomplished by lifting all operators independently, including the z^{-1} back-edge.

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



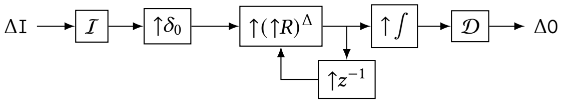
This proposition gives the ability to lift entire circuits, including circuits computing on streams and having feedback edges, which are well-defined, due to Proposition 6.1. With this machinery we can now apply Algorithm 4.6 to arbitrary circuits, even circuits built for recursively-defined relations.

Step 1: Start with the "semi-naive" circuit (5.1):

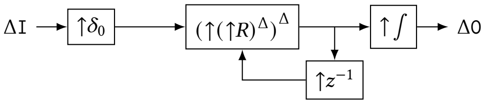


Step 2: nothing to do (for *distinct*).

Steps 3 and 4: Lift the circuit (using 6.2) and incrementalize:



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



(6.1)

This is the incremental version of an arbitrary recursive query. An example for a transitive closure query is in our report [12].

6.1 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 (6.1) never performs more iterations than the non-incremental circuit (5.1): once the non-incremental circuit reaches the fixed point, its output is constant, and the derivative of corresponding value in the incrementalized circuit becomes 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 [12].

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).

7 DBSP AND RICHER QUERY LANGUAGES

The DBSP language can express a richer class of streaming computations (both incremental and non-incremental) than those covered so far. In this section we enumerate several important classes of queries that can be implemented in DBSP, and thus can be incrementalized using Algorithm 4.6.

7.1 Multisets and bags

In §4 we have shown how to implement the relational algebra on sets. Some SQL queries however produce *multisets*, e.g., `UNION ALL`. Since $\mathbb{Z}$-sets generalize multisets and bags, it is trivial to implement query operators on multisets, by just omitting *distinct* operator invocations. For example, SQL `UNION` is $\mathbb{Z}$-set addition followed by *distinct*, whereas `UNION ALL` is just $\mathbb{Z}$-set addition. Indeed, the SQL to DBSP compiler mentioned in §8 handles full standard SQL, including all multiset queries.

7.2 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: $a : 2^A \to B$. In DBSP an aggregation function has a signature $a : \mathbb{Z}[A] \to B$.

The SQL `COUNT` aggregation function is implemented on $\mathbb{Z}$-sets by $a_{\texttt{COUNT}} : \mathbb{Z}[A] \to \mathbb{Z}$, which computes a *sum* of all the element weights: $a_{\texttt{COUNT}}(s) = \sum_{x \in s} s[x]$. The SQL `SUM` aggregation function is implemented on $\mathbb{Z}$-sets by $a_{\texttt{SUM}} : \mathbb{Z}[\mathbb{R}] \to \mathbb{R}$ which performs a *weighted sum* of all (real) values: $a_{\texttt{SUM}}(s) = \sum_{x \in s} x \times s[x]$. Both these implementations work correctly for sets and multisets.

With this definition the aggregation functions $a_{\texttt{COUNT}}$ and $a_{\texttt{SUM}}$ are in fact linear transformations between the group $\mathbb{Z}[A]$ and the result group ($\mathbb{Z}$, and $\mathbb{R}$ respectively).

If the output of the DBSP circuit is allowed to be such a "scalar" value, then aggregation with a linear function is simply function application, and thus linear. However, in general, composing multiple queries requires the result of an aggregation to be a singleton $\mathbb{Z}$-set (containing a single value), and not a scalar value. In this case the aggregation function is implemented in DBSP as the composition of the actual aggregation and the makeset : $A \to \mathbb{Z}[A]$ function, which converts a scalar value of type A to a singleton $\mathbb{Z}$-set, defined as follows: $\text{makeset}(x) \overset{\text{def}}{=} \{x \mapsto 1\}$.

In conclusion, the following SQL query: `SELECT SUM(c) FROM I` is implemented as the following circuit:[7]

$$\texttt{I} \longrightarrow \boxed{\pi_{\texttt{C}}} \rightarrow \boxed{a_{\texttt{SUM}}} \longrightarrow \boxed{\text{makeset}} \rightarrow \texttt{O}$$

The lifted incremental version of this circuit is interesting: since π and $a_{\texttt{SUM}}$ are linear, they are equivalent to their own incremental versions. Although $(\uparrow\text{makeset})^\Delta = \mathcal{D} \circ \uparrow\text{makeset} \circ \mathcal{I}$ cannot be simplified, it is nevertheless efficient, doing only O(1) work per invocation, since its input and output are singleton values.

Finally, some aggregate functions, such as `MIN`, are *not* linear: for handling deletions they need to track the full set. One way to implement in DBSP the lifted incremental version of such aggregate functions is by "brute force", using the formula $(\uparrow a_{\texttt{MIN}})^\Delta = \mathcal{D} \circ \uparrow a_{\texttt{MIN}} \circ \mathcal{I}$. Such an implementations performs work $O(|DB|)$ at each invocation. However, schemes such as Reactive Aggregator [47] can be implemented as custom DBSP operators to bring the amortized cost per update to $O(\log |DB|)$. This approach is similar to the customized implementation of the *distinct* operator, and it is another facet of the modularity of DBSP, which allows optimized operator implementations to be mixed and matched.

7.3 Grouping; indexed relations

Let K be a set of "key values." Consider the mathematical structure of finite maps from K to $\mathbb{Z}$-sets: $K \to \mathbb{Z}[A] = \mathbb{Z}[A][K]$. We call values i of this structure **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.

We use this structure 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 $G_p(a)[k] \overset{\text{def}}{=} \sum_{x \in a.p(x)=k} a[x] \cdot x$ (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 a this function returns a indexed $\mathbb{Z}$-set, where each element is a **grouping**:[8] for each key k a grouping is a $\mathbb{Z}$-set containing all elements of a that map to k (as in SQL, groupings are multisets). Consider our example $\mathbb{Z}$-set R from §4, and a key function $p(s)$ that returns the first letter of the string s. Then we have that $G_p(R) = \{\texttt{j} \mapsto \{\texttt{joe} \mapsto 1\}, \texttt{a} \mapsto \{\texttt{anne} \mapsto -1\}\}$, i.e., grouping with this key function produces an indexed $\mathbb{Z}$-set with two groupings, each of which contains a $\mathbb{Z}$-set with one element.

The grouping function G_p is linear for any key function p! It follows that the group-by implementation in DBSP is automatically incremental: given some changes to the input relation we can apply the partitioning function to each row changed in the input separately to compute how each grouping changes.

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. Our definition of incremental computation is only concerned with incrementality in the *outermost* structures. We leave it to future work to explore an appropriate definition of incremental computation on the *inner* relations.

7.4 `GROUP BY-AGGREGATE`, flatmap

Grouping in SQL is almost always followed by aggregation. Let us consider an aggregation function $a : (K \times \mathbb{Z}[A]) \to B$ that produces values in some group B, and an indexed relation of type $\mathbb{Z}[A][K]$, as defined above in §7.3. The nested relation aggregation operator $Agg_a : \mathbb{Z}[A][K] \to B$ applies a to the contents of each grouping independently and adds the results: $Agg_a(g) \overset{\text{def}}{=} \sum_{k \in K} a(k, g[k])$. To apply this to our example, let us compute the equivalent of `GROUP BY-COUNT`; we use the following aggregation function *count* : $K \times \mathbb{Z}[A]$, $count(k, s) = \text{makeset}((k, a_{\texttt{COUNT}}(s)))$, using the $\mathbb{Z}$-set counting function $a_{\texttt{COUNT}}$ from §7.2; the notation (a, b) is a pair of values a and b. Then we have $Agg_{count}(G_p(R)) = \{(\texttt{j}, 1) \mapsto$

[7] The actual SQL `SUM` aggregate is even more complicated, because it needs to skip `NULL`s, and it returns `NULL` for an empty input set; this can be implemented in DBSP.

[8] We use "group" for the algebraic structure and "grouping" for the result of `GROUP BY`.

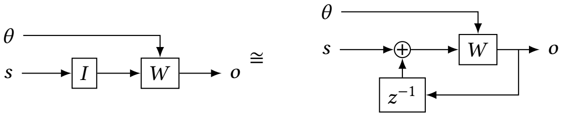
$1, (\mathsf{a}, -1) \mapsto 1\}$.

A very useful operation on nested relations is **flatmap** (or UNNEST in SQL), which is essentially the inverse of grouping, converting an indexed $\mathbb{Z}$-set into a $\mathbb{Z}$-set: flatmap : $\mathbb{Z}[A][K] \to \mathbb{Z}[A \times K]$. flatmap is in fact a particular instance of aggregation, using the aggregation function $a : K \times \mathbb{Z}[A] \to \mathbb{Z}[A \times K]$ defined by $a(k, s) = \sum_{x \in s[k]} s[k][x] \cdot (k, x)$. For our previous example, $\text{flatmap}(G_p(R)) = \{(\mathsf{j}, \mathsf{joe}) \mapsto 1, (\mathsf{a}, \mathsf{anne}) \mapsto -1\}$.

7.5 Streaming joins

Consider a binary query $T(s, t) = \mathcal{I}(s) \uparrow\bowtie t$. This is the *relation-to-stream join* operator supported by streaming databases like Kafka's ksqlDB [31]. Stream s carries changes to a relation, while t carries arbitrary data, e.g., logs or telemetry data points. T discards values from t after matching them against the accumulated contents of the relation $\mathcal{I}(s)$.

Streaming Window queries. Streaming databases often organize the contents of streams into windows, which store a subset of data points with a predefined range of timestamps. In practice, windowing is usually based on physical timestamps attached to stream values rather than logical (transaction) time as in the previous circuit. For instance, the CQL [9] query "SELECT * FROM events [RANGE 1 hour]" returns all events received within the last hour. The corresponding circuit (on the left) takes input stream $s \in \mathcal{S}_{\mathbb{Z}[A]}$ and an additional input $\theta \in \mathcal{S}_{\mathbb{R}}$ that carries the value of the current time.



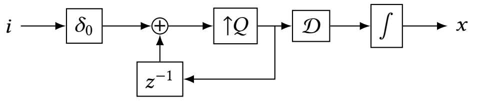
where the *window operator* W prunes input $\mathbb{Z}$-sets, only keeping values with timestamps less than an hour behind $\theta[t]$. Assuming $ts : A \to \mathbb{R}$ returns the physical timestamp of a value, W is defined as $W(v, \theta)[t] \overset{\text{def}}{=} \{x \in v[t] . ts(x) \geq \theta[t] - 1hr\}$. Assuming θ increases monotonically, W can be moved inside integration, resulting in the circuit on the right, which uses bounded memory to compute a window of an unbounded stream. This circuit is a building block of a large family of window queries, including window joins and window aggregation (e.g., SQL OVER queries).

7.6 Relational while queries

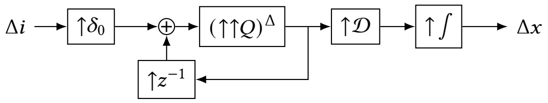
DBSP can express programs that go beyond Datalog: see the non-monotonic semantics for Datalog$^{\neg}$ and Datalog$^{\neg\neg}$[5]. We implement the following "while" program, where Q is an arbitrary query:

```
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 4.6 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.

8 IMPLEMENTATION

The scope of this paper is the DBSP theory of IVM, so we only briefly touch upon the implementation aspects. We defer a full description and evaluation of the system to a future paper.

DBSP Rust library. We have built an implementation of DBSP as part of an open-source project with an MIT license: https://github.com/vmware/database-stream-processor. The implementation consists of a Rust library and a runtime. The library provides APIs for basic algebraic data types: such as groups, finite maps, $\mathbb{Z}$-set, indexed $\mathbb{Z}$-set. A separate circuit construction API allows users to create DBSP circuits by placing operator nodes (corresponding to boxes in our diagrams) and connecting them with streams, which correspond to the arrows in our diagrams. The library provides pre-built generic operators for integration, differentiation, delay, nested integration and differentiation, and a rich library of $\mathbb{Z}$-set basic incremental operators: corresponding to plus, negation, grouping, joining, aggregation, *distinct*, flatmap, window aggregates, etc.

For iterative computations the library provides the δ_0 operator and an operator that approximates $\int$ by terminating iteration of a loop at a user-specified condition (usually the condition is the requirement for a zero to appear in a specified stream). The low level library allows users to construct incremental circuits manually by stitching together incremental versions of primitive operators.

The library supports data-parallel multicore evaluation of circuits using a natural sharding strategy, and a variety of adapters for external data sources (e.g., Kafka, CSV files, etc). The library can also spill internal operator state to persistent storage. Benchmark results (which are very promising) are available in the code repository and will be discussed in future work.

SQL compiler. We have also built a SQL to DBSP compiler, which translates standard SQL queries into DBSP circuits. The compiler implements Algorithm 4.6, to generate a streaming version of any SQL query. The compiler is open-source https://github.com/vmware/sql-to-dbsp-compiler with an MIT license. The compiler front-end parser and optimizer are based on the Apache Calcite [10] infrastructure. The project is mature enough to pass all 7 million SQL Logic Tests [2]. The compiler handles all aspects of SQL, including NULLs, ternary logic, grouping, aggregation, multiset queries, etc. Currently correlated sub-queries and outer joins are essentially converted to equivalent relational plans using multiple joins.

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 [14]. The formalization builds on mathlib [37], 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 [12] and Lean.

9 RELATED WORK

Incremental view maintenance [15, 16, 24–26] is a much studied problem in databases. A survey of results for Datalog queries is present in [40]. 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 [6, 33], where the incrementalization process is repeated for the delta query.

Many custom algorithms were published for various classes of queries: e.g. [34] handles positive nested relational calculus. DYN [28] and IDYN [29, 30] focus on acyclic conjunctive queries. Instead of keeping the output view materialized they build data structures that allow efficiently querying the output views. PAI maps [4] are specially designed for queries with correlated aggregations. AJU [48] focuses on foreign-key joins. It is a matter of future work to evaluate whether custom DBSP operators can match the efficiency of systems specialized for narrow classes of queries.

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 [31] `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 [19, 41] that maintain the number of derivations of each output fact: DRed [26] and its variants [8, 13, 35, 36, 46, 49], the backward-forward algorithm and variants [27, 40, 41]. 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 *distinct* operator shows that precise counting is not necessary.

Picallo et al. [7] provide a general solution to IVM for rich languages. 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 (4.6) 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 [33] and [22] 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) [20]. An intriguing question is "what other interesting group structures could this be applied to besides $\mathbb{Z}$-sets?" Papers such as [43] explore other possibilities, such as matrix algebra, linear ML models, or conjunctive queries.

DBSP does not do anything special for triangle queries [32]. Are there better algorithms for this case?

In §7 we have briefly mentioned that DBSP can easily model window and stream database queries [1, 9]; it is an interesting question whether there are CQL queries that cannot be expressed in DBSP (we conjecture that there aren't any).

Bonifati et al. [11] 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) [39, 42] and its theoretical foundations [3] (and recently [17, 38]). DD's computational model is more powerful than DBSP, since it allows time values to be 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 (↑). DBSP can express both incremental and non-incremental computations. Most importantly, DBSP comes with Algorithm 4.6, 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. Differential Datalog [45] does it for a Datalog dialect.) 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.

10 CONCLUSIONS

We have introduced DBSP, a model of computation based on infinite streams over abelian groups. In this model streams are used for 3 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.

REFERENCES

[1] [n.d.]. The Aurora Project. http://cs.brown.edu/research/aurora/. Last accessed November 2022.

[2] [n.d.]. sqllogictest. https://www.sqlite.org/sqllogictest/doc/trunk/about.wiki. Last accessed March 2023.

[3] Martín Abadi, Frank McSherry, and Gordon Plotkin. 2015. Foundations of Differential Dataflow. In *Foundations of Software Science and Computation Structures (FoSSaCS)*. London, UK. http://homepages.inf.ed.ac.uk/gdp/publications/differentialweb.pdf

[4] Supun Abeysinghe, Qiyang He, and Tiark Rompf. 2022. Efficient Incrementialization of Correlated Nested Aggregate Queries Using Relative Partial Aggregate Indexes (RPAI). In *ACM SIGMOD International conference on Management of data (SIGMOD)* (Philadelphia, PA, USA). 136–149. https://doi.org/10.1145/3514221.3517889

[5] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley. http://webdam.inria.fr/Alice/

[6] Yanif Ahmad and Christoph Koch. 2009. DBToaster: A SQL Compiler for High-Performance Delta Processing in Main-Memory Databases. *Proc. VLDB Endow.* 2, 2 (Aug. 2009), 1566–1569. https://doi.org/10.14778/1687553.1687592

[7] Mario Alvarez-Picallo, Alex Eyers-Taylor, Michael Peyton Jones, and C.-H. Luke Ong. 2019. Fixing Incremental Computation. In *European Symposium on Programming Languages and Systems (ESOP)*. Prague, Czech Republic, 525–552. https://link.springer.com/chapter/10.1007/978-3-030-17184-1_19

[8] Krzysztof R. Apt and Jean-Marc Pugin. 1987. Maintenance of Stratified Databases Viewed as a Belief Revision System. In *ACM SIGMOD International conference on Management of data (SIGMOD)*. San Diego, California, 136–145. https://doi.org/10.1145/28659.28674

[9] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2002. *An Abstract Semantics and Concrete Language for Continuous Queries over Streams and Relations*. Technical Report 2002-57. Stanford InfoLab. http://ilpubs.stanford.edu:8090/563/

[10] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *International Conference on Management of Data (IDMD)* (Houston, TX, USA). 221–230. https://doi.org/10.1145/3183713.3190662

[11] Angela Bonifati, Stefania Dumbrava, and Emilio Jesús Gallego Arias. 2018. Certified Graph View Maintenance with Regular Datalog. *Theory and Practice of Logic Programming* 18, 3-4 (2018), 372–389. https://doi.org/10.1017/S1471068418000224

[12] Mihai Budiu, Frank McSherry, Leonid Ryzhyk, and Val Tannen. 2022. DBSP: A Language for Expressing Incremental View Maintenance for Rich Query Languages. https://github.com/vmware/database-stream-processor/blob/main/doc/spec.pdf.

[13] Stefano Ceri and Jennifer Widom. 1991. Deriving Production Rules for Incremental View Maintenance. In *International Conference of Very Large Data Bases (VLDB)*. Barcelona, Spain, 577–589. http://www.vldb.org/conf/1991/P577.PDF

[14] Tej Chajed. 2022. DBSP formalization. https://github.com/tchajed/dbsp-theory

[15] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. 1995. Optimizing Queries with Materialized Views. In *International Conference on Data Engineering (ICDE)*. 190–200.

[16] Rada Chirkova and Jun Yang. 2012. *Materialized Views*. Now Publishers Inc., Hanover, MA, USA.

[17] Zaheer Chothia, John Liagouris, Frank McSherry, and Timothy Roscoe. 2016. Explaining Outputs in Modern Data Analytics. *Proc. VLDB Endow.* 9, 12 (Aug. 2016), 1137–1148. https://doi.org/10.14778/2994509.2994530

[18] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover. In *International Conference on Automated Deduction (CADE-25)*. Berlin, Germany.

[19] Hasanat M. Dewan, David Ohsie, Salvatore J. Stolfo, Ouri Wolfson, and Sushil Da Silva. 1992. Incremental Database Rule Processing In PARADISER. *J. Intell. Inf. Syst.* 1, 2 (1992), 177–209. https://doi.org/10.1007/BF00962282

[20] J. Nathan Foster, Ravi Konuru, Jerome Simeon, and Lionel Villard. 2008. An Algebraic Approach to XQuery View Maintenance. In *ACM SIGPLAN Workshop on Programming Languages Technologies for XML*. San Francisco, CA.

[21] Sergio Greco and Cristian Molinaro. 2015. Datalog and Logic Databases. *Synthesis Lectures on Data Management* 7, 2 (2015), 1–169. https://doi.org/10.2200/S00648ED1V01Y201505DTM041

[22] Todd J Green, Zachary G Ives, and Val Tannen. 2011. Reconcilable differences. *Theory of Computing Systems* 49, 2 (2011), 460–488. https://web.cs.ucdavis.edu/~green/papers/tocs11_differences.pdf

[23] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance Semirings. In *Symposium on Principles of Database Systems (PODS)*. Beijing, China, 31–40. https://doi.org/10.1145/1265530.1265535

[24] Timothy Griffin and Leonid Libkin. 1995. Incremental Maintenance of Views with Duplicates. In *ACM SIGMOD International conference on Management of data (SIGMOD)* (San Jose, California, USA). 328–339. https://doi.org/10.1145/223784.223849

[25] Ashish Gupta, Inderpal Singh Mumick, et al. 1995. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.* 18, 2 (1995), 3–18.

[26] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. 1993. Maintaining Views Incrementally. In *ACM SIGMOD International Conference on Management of Data*. Washington, D.C., USA, 157–166. https://doi.org/10.1145/170035.170066

[27] John V. Harrison and Suzanne W. Dietrich. 1992. Maintenance of Materialized Views in a Deductive Database: An Update Propagation Approach. In *Workshop on Deductive Databases (Technical Report)*. Washington, D.C., 56–65.

[28] Muhammad Idris, Martin Ugarte, and Stijn Vansummeren. 2017. The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates. In *ACM SIGMOD International conference on Management of data (SIGMOD)* (Chicago, Illinois, USA). 1259–1274. https://doi.org/10.1145/3035918.3064027

[29] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. 2018. Conjunctive Queries with Inequalities under Updates. *Proc. VLDB Endow.* 11, 7 (mar 2018), 733–745. https://doi.org/10.14778/3192965.3192966

[30] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. 2019. Efficient Query Processing for Dynamically Changing Datasets. *SIGMOD Rec.* 48, 1 (November 2019), 33–40. https://doi.org/10.1145/3371316.3371325

[31] Hojjat Jafarpour, Rohan Desai, and Damian Guy. 2019. KSQL: Streaming SQL Engine for Apache Kafka. In *International Conference on Extending Database Technology (EDBT)*. Lisbon, Portugal, 524–533. http://openproceedings.org/2019/conf/edbt/EDBT19_paper_329.pdf

[32] Ahmet Kara, Hung Q. Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. 2020. Maintaining Triangle Queries under Updates. *ACM Trans. Database Syst.* 45, 3, Article 11 (aug 2020), 46 pages. https://doi.org/10.1145/3396375

[33] Christoph Koch. 2010. Incremental Query Evaluation in a Ring of Databases. In *Symposium on Principles of Database Systems (PODS)*. Indianapolis, Indiana, USA, 87–98. https://doi.org/10.1145/1807085.1807100

[34] Christoph Koch, Daniel Lupei, and Val Tannen. 2016. Incremental View Maintenance For Collection Programming. In *Symposium on Principles of Database Systems (PODS)*. San Francisco, California, USA, 75–90. https://doi.org/10.1145/2902251.2902286

[35] Jakub Kotowski, François Bry, and Simon Brodt. 2011. Reasoning as Axioms Change - Incremental View Maintenance Reconsidered. In *Web Reasoning and Rule Systems RR (Lecture Notes in Computer Science, Vol. 6902)*. Springer, Galway, Ireland, 139–154. https://doi.org/10.1007/978-3-642-23580-1_11

[36] James J. Lu, Guido Moerkotte, Joachim Schü, and V. S. Subrahmanian. 1995. Efficient Maintenance of Materialized Mediated Views. In *ACM SIGMOD International conference on Management of data (SIGMOD)*. San Jose, California, 340–351. https://doi.org/10.1145/223784.223850

[37] The mathlib Community. 2020. The Lean Mathematical Library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New Orleans, LA, USA) *(CPP 2020)*. Association for Computing Machinery, New York, NY, USA, 367–381. https://doi.org/10.1145/3372885.3373824

[38] Frank McSherry, Andrea Lattuada, Malte Schwarzkopf, and Timothy Roscoe. 2020. Shared Arrangements: Practical Inter-Query Sharing for Streaming Dataflows. *Proc. VLDB Endow.* 13, 10 (June 2020), 1793–1806. https://doi.org/10.14778/3401960.3401974

[39] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow. In *Conference on Innovative Data Systems Research (CIDR)*. Asilomar, CA, 12 pages. https://www.cidrdb.org/cidr2013/Papers/CIDR13_Paper111.pdf

[40] Boris Motik, Yavor Nenov, Robert Piro, and Ian Horrocks. 2019. Maintenance of Datalog materialisations revisited. *Artif. Intell.* 269 (2019), 76–136. https://doi.org/10.1016/j.artint.2018.12.004

[41] Boris Motik, Yavor Nenov, Robert Edgar Felix Piro, and Ian Horrocks. 2015. Incremental Update of Datalog Materialisation: the Backward/Forward Algorithm. In *Conference on Artificial Intelligence (AAAI)*. Austin, Texas, 1560–1568. http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9660

[42] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System. In *ACM Symposium on Operating Systems Principles (SOSP)*. Farminton, Pennsylvania, 439–455. https://doi.org/10.1145/2517349.2522738

[43] Milos Nikolic and Dan Olteanu. 2018. Incremental View Maintenance with Triple Lock Factorization Benefits. In *International Conference on Management of Data (ICMD)* (Houston, TX, USA). 365–380. https://doi.org/10.1145/3183713.3183758

[44] L. R. Rabiner and B. Gold (Eds.). 1975. *Theory and Application of Digital Signal Processing*. Prentice-Hall.

[45] Leonid Ryzhyk and Mihai Budiu. 2019. Differential Datalog. In *Datalog 2.0*. Philadelphia, PA, 12 pages. http://budiu.info/work/ddlog.pdf

[46] Martin Staudt and Matthias Jarke. 1996. Incremental Maintenance of Externally Materialized Views. In *International Conference of Very Large Data Bases (VLDB)*. Mumbai (Bombay), India, 75–86. http://www.vldb.org/conf/1996/P075.PDF

[47] Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. 2015. General Incremental Sliding-Window Aggregation. *Proc. VLDB Endow.* 8, 7

(February 2015), 702–713. https://doi.org/10.14778/2752939.2752940

[48] Qichen Wang and Ke Yi. 2020. Maintaining Acyclic Foreign-Key Joins under Updates. In *ACM SIGMOD International conference on Management of data (SIGMOD)*. Portland, OR, USA, 1225–1239. https://doi.org/10.1145/3318464.3380586

[49] Ouri Wolfson, Hasanat M. Dewan, Salvatore J. Stolfo, and Yechiam Yemini. 1991. Incremental Evaluation of Rules and its Relationship to Parallelism. In *ACM SIGMOD International conference on Management of data (SIGMOD)*. ACM Press, Denver, Colorado, 78–87. https://doi.org/10.1145/115790.115799