DBSP: Incremental Computation on Streams and Its Applications to Databases

Mihai Budiu
Feldera
mbudiu@feldera.com

Tej Chajed
Univ. of Wisconsin-Madison
chajed@wisc.edu

Frank McSherry
Materialize Inc.
mcsherry@materialize.com

Leonid Ryzhyk
Feldera
leonid@feldera.com

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

ABSTRACT

We describe DBSP, a framework for incremental computation. Incremental computations repeatedly evaluate a function on some input values that are "changing". The goal of an efficient implementation is to "reuse" previously computed results. Ideally, when presented with a new change to the input, an incremental computation should only perform work proportional to the size of the changes of the input, rather than to the size of the entire dataset.

In databases "incremental computation" is known as Incremental View Maintenance (IVM); IVM has long been a central problem of database theory and practice. Many solutions have been proposed for restricted classes of computation or of changes, but we are seeking a general solution.

We start by defining incremental computations as computations on data streams, i.e., sequences of data values, by borrowing ideas from Digital Signal Processing.

Using these tools, we give a general solution to the incremental computation problem in 4 steps: (1) we describe a simple but expressive language called DBSP for describing computations over data streams; (2) we give a new mathematical definition of incremental computation for DBSP programs; (3) we give a general algorithm for converting any DBSP program into an incremental program. The algorithm reduces the problem of incrementalizing a complex query to the problem of incrementalizing the primitive operations that compose the query. Finally, (4) we show that practical database query languages, such as SQL and Datalog, can be directly implemented on top of DBSP, using primitives with efficient incremental implementations. As a consequence, we obtain a general recipe for efficient IVM for essentially arbitrary queries written in all these languages.

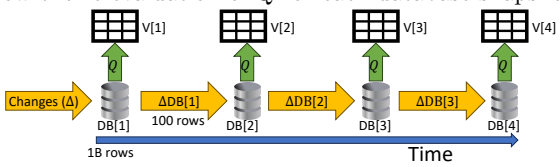
1. INTRODUCTION

1.1 Incremental computation

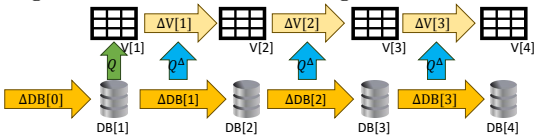
Incremental view maintenance (IVM) is an important and well-studied problem in databases [14]. The IVM problem can be stated as follows: we are given a large database DB (say 1 billion records) and a view V, described by a query Q. The goal of IVM is to keep the contents of V up-to-date in response to changes of the database.

As a concrete example, consider the following view definition statement in SQL: `CREATE VIEW V AS SELECT * FROM T WHERE Age >= 10`. In this example the query Q defining the view V is `SELECT * FROM T WHERE Age >= 10`. The view `V` always contains all the rows of table `T` whose value for the column `Age` is greater than or equal to 10.

In general a query is a function applied to the database state: $V = Q(DB)$. A naïve solution re-executes query Q every time the database changes, illustrated in the following diagram. Time is the horizontal axis; the horizontal arrows labeled with Δ depict changes to the database, which we assume are much smaller than the database itself (e.g., a change could touch perhaps 100 records). The "up" arrows show the re-evaluation of Q for each database snapshot.



The naive solution is expensive. After the first version of the view has been constructed, an ideal algorithm would compute only *changes* to the view ΔV doing work $O(|\Delta DB|)$. Ideally, we want to construct a new query Q^Δ with the property that $\Delta V = Q^\Delta(\Delta DB)$, i.e., Q^Δ can compute the change of the view from the change of the database:



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

In this paper we propose a new way to define Q^Δ, as a form of *computation on streams*. Our model is inspired by Digital Signal Processing DSP [16], applied to databases, hence the name DBSP. Q^Δ can be very efficient. As for traditional database queries, the performance of Q^Δ depends both on the query Q but also on the actual data that the query is applied to. Informally, Q^Δ built by our algorithm, is faster

 This is a simplified version of the paper entitled DBSP: Automatic Incremental View Maintenance for Rich Query Languages, published in PVLDB, Vol. 16, Issue 7, ISSN 2150-8097 [5]. DOI: https://doi.org/10.14778/3587136.3587137

than Q by a factor of $O(|DB|/|\Delta DB|)$. In practice this may be an improvement of several orders of magnitude. For our example above $|DB| \approx 10^9$ and $|\Delta DB| \approx 10^2$, this can make Q^Δ **10 million times** faster!

Instead of treating the database as a large changing object, we model it as a *sequence* or *stream* of database snapshots. Similarly, consecutive view snapshots form a stream. DBSP is a simple programming language computing on streams; inputs and outputs are streams of arbitrary values.

The DBSP language has only 4 operators. However, it can express a rich set of computations on streams, including repeated computations (similar to the repeated queries Q above), recursive computations that compute fixed points (like Datalog programs), streaming computations, and incremental computations (which we define shortly). The full paper [5] gives a precise mathematical description of DBSP, this presentation is simplified to convey the main intuitions. We omit the related work section from this presentation.

The central result of this paper is Algorithm 4.1 which, given a DBSP program that computes on a stream of values, mechanically transforms it into an incremental DBSP program that computes on a stream of changes.

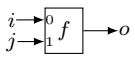
DBSP is not tied to databases in any way; it is in fact a Turing-complete language that can be used for many other purposes. But it works particularly well in the area of databases, for two reasons:

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

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

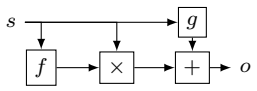
1.2 Circuits and Streams

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

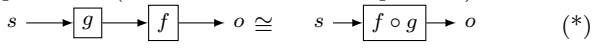


Most of the functions we deal with are commutative, so we can skip inputs label, displaying the circuit above as:

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



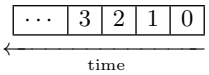
We say that two circuits are **equivalent** if they compute the same function. We use the symbol $\cong$ to indicate circuit equivalence. For example, we have the following circuit equivalence (where $\circ$ is function composition):



(*)

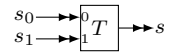
1.3 Streams

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

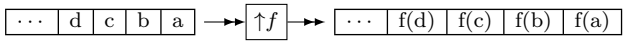


A **stream operator** is a function that computes on streams and produces streams. In general we use "operator" for streams, and "function" for computations on "scalar" values.

We use arrows with a double head to depict streams. The following diagram shows a stream operator T consuming two input streams s_0 and s_1, producing one output stream s:



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



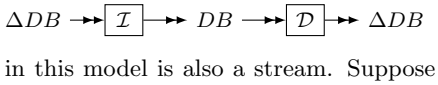
To simplify the notation, we write $a + b$ for streams a, b instead of $a(\uparrow+)b$; we also write $-a$ instead of $(\uparrow-)a$.

1.4 Databases as streams

We generally think of streams as sequences of "small" values, such as insertions or deletions in a database. However, we also treat the whole database as a *stream of database snapshots*. We model a database as a stream DB. Time is not wall-clock time, but counts the transactions 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. This notation is apparent in the diagrams in Section 1.1.

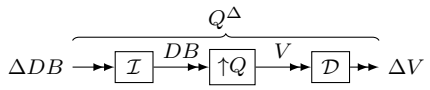
Database transactions also form a stream ΔDB, this time a stream of *changes*, or *deltas*, that are applied to the database. The values of this stream are defined by $(\Delta DB)[t] = DB[t] - DB[t-1]$, where "$-$" stands for the difference between two databases, a notion that we will soon make more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

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



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

Armed with these basic definitions, we can precisely define IVM. What does it mean to maintain a view incrementally? A 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, depicted graphically as:



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

Armed with this general theory of incremental computation, in Section 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. Section 5 shows how programs containing recursion can be implemented and incrementalized in DBSP. For example, given an implementation of transitive closure in the natural recursive way, our algorithm produces a program that efficiently maintains the transitive closure of a graph as nodes and edges are added and deleted.

1.5 Contributions

This work makes the following contributions:

(1) We introduce DBSP, a simple but expressive language for streaming computation. DBSP gives an elegant formal foundation unifying the manipulation of streaming and incremental computations.

(2) An algorithm for incrementalizing any streaming computation expressed in DBSP that handles arbitrary insertions and deletions from any of the data sources.

(3) An illustration of how DBSP can model various classes of practical queries, such as relational algebra, nested relations, aggregations, and Datalog.

(4) The first general and machine-checked theory of IVM. All the theoretical results in the original version of this paper [5] have been checked [7] using the Lean proof assistant [8].

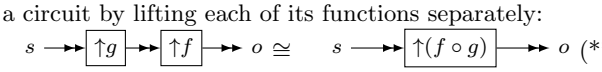
(5) A practical open-source implementation of this theory as a runtime and a SQL compiler.

2. STREAM OPERATORS

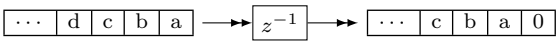
For the rest of this paper we require the set of values A of a stream $\mathcal{S}_A$ to form a commutative group, with operations $+$, $-$, and a 0 (zero) value. The *plus* defines what it means to *add* new data, while the *minus* allows us to compute differences (deltas). We show later that this requirement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

Stream operators are very powerful mathematically, but in DBSP we restrict ourselves to a very small subset. All DBSP computations are *causal* [4]: the output at time t is produced immediately after all inputs up at time t have been received; the output at time t cannot depend on inputs arriving after t.

The following circuit equivalence tells us that we can lift a circuit by lifting each of its functions separately:

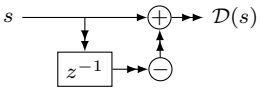


The **delay operator** z^{-1} produces an output stream by delaying its input by one step (and starting with a zero)[1]:

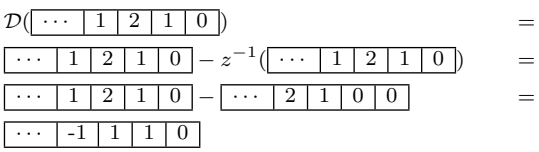


A very important property of the delay operator is that it produces the first output *before* having received the first input, and it produces the second output before having received the second input, etc.

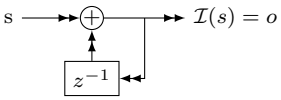
We define the **differentiation operator** as a composition of several other operators: $\mathcal{D}(s) \stackrel{\text{def}}{=} s - z^{-1}(s)$, shown as:



If s is a stream, then $\mathcal{D}(s)$ is the *stream of changes* of s; a value in the output is the difference between two consecutive values in the input. As an example:



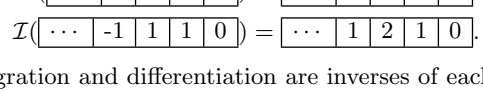
The **integration operator** is given by the following circuit:



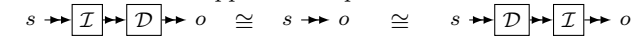
While this definition may seem strange, because the output stream is used to compute itself, the use of the delay in the "feedback" loop ensures that only *previous* values of the output are used in computing the current one. Using the notation $o = \mathcal{I}(s)$ to make formulas more readable, we can see the contents of stream o is produced step by step:

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

In general, $\mathcal{I}(s)[t] = o[t] = \sum_{i \le t} s[i]$. Examples:



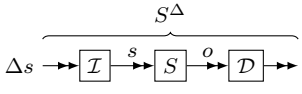
Integration and differentiation are inverses of each other: while $\mathcal{D}$ computes the changes of a stream, $\mathcal{I}$ reconstitutes the original stream given the stream of changes. $\mathcal{I}$ and $\mathcal{D}$ "cancel out" when applied in sequence:



3. INCREMENTAL VIEW MAINTENANCE

The results in this section are not specific to databases, they hold for any stream computations, but we hint about their applicability for databases.

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



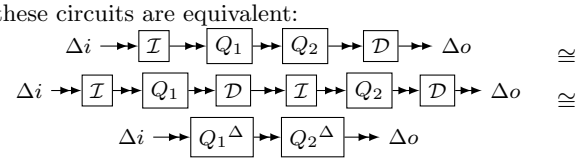
[1]This bizarre name comes from digital signal processing.

If S computes on a stream s, then S^Δ computes on a stream of changes to s. If S produces a stream o, then S^Δ produces the stream of changes to o. Note that this definition does not require S to be a lifted function.

For an operator with multiple inputs and outputs we define the incremental version by applying $\mathcal{I}$ to each input, and $\mathcal{D}$ to each output, e.g.: $T^\Delta(a,b) \stackrel{\text{def}}{=} \mathcal{D}(T(\mathcal{I}(a),\mathcal{I}(b)))$.

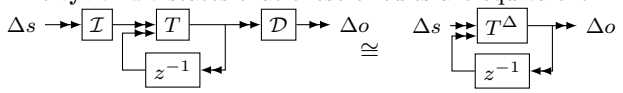
S^Δ has many nice properties:

The **chain rule** states that $(Q_1 \circ Q_2)^\Delta = {Q_1}^\Delta \circ {Q_2}^\Delta$, i.e., these circuits are equivalent:



In the database world, we can read this as: **to incrementalize a composite query you can incrementalize each sub-query independently**. This gives us a simple deterministic recipe reducing the incremental version of an arbitrary query to the incremental version of its primitive operators.

The **cycle rule** states that these circuits are equivalent:



(We have omitted the labels on the inputs of T.) In other words, the incremental version of a feedback loop around a query is just the feedback loop with the incremental query for its body. This result will be useful for recursive queries.

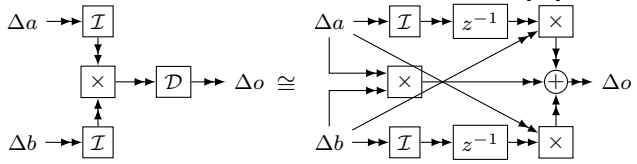
We call an operator S **linear** if it has the property that $S(a+b) = S(a)+S(b)$ (where $+$ is the addition of streams). For a linear operator S we have $S^\Delta = S$. This is very useful because many primitive database operations can be implemented as linear operators: selection, projection, filtering, grouping, parts of aggregation are all linear. Moreover, the following operators are linear: $-$, z^{-1}, $\mathcal{I}$, $\mathcal{D}$, $\uparrow f$ if f is a linear function.

We call an operator T with two inputs **bilinear** if it distributes over stream addition: $T(a+b,c) = T(a,c)+T(b,c)$, and $T(a,c+d) = T(a,c)+T(a,d)$. (Similar to multiplication's distributivity over addition.) In databases intersection, joins, and Cartesian products are bilinear.

Using infix notation, for a bilinear operator $\times$ we have:

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

If we ignore the delay operators in this equation we recover the well-known formula for join delta queries, e.g.,[15].



What is the intuition behind this diagram? Let us consider the case of Cartesian product $a \times b$. The incremental product has inputs $\Delta a = \mathcal{D}(a)$ and $\Delta b = \mathcal{D}(b)$. What happens when we add a row x to relation a (i.e., $\Delta a = x$)? The new row x will appear in the output change combined with every row in the *previous version* of the *full* relation b. The operator $\mathcal{I}(\Delta b)$ in fact computes relation b from the stream Δb of changes, and z^{-1} applied to this value gives us its previous version. So the bottom $\times$ operator computes $x \times z^{-1}(b) = \Delta a \times z^{-1}(\mathcal{I}(\Delta b))$, the change produced by the new row x. The top $\times$ operator performs the symmetric operation for the changes of the b relation. The middle $\times$ operator produces the results of changes to both inputs.

4. IVM FOR THE RELATIONAL ALGEBRA

In this section we apply the results on incremental computation to relational databases. As explained in the introduction, our goal is to efficiently compute the incremental version of any relational query Q.

However, we face a technical problem: we said that streams require their values to belong to a commutative group, and relational databases in general are *not* commutative 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 [12]) to represent sets.

4.1 $\mathbb{Z}$-sets

$\mathbb{Z}$-sets generalize database tables: think of a $\mathbb{Z}$-set as a table where each row has an associated integer weight, possibly negative. This weight indicates *how many times* the row belongs to the table.

The following table shows an example $\mathbb{Z}$-set with three rows. The first row has value `joe` and weight 1. We do not show rows with weight 0.

Row	Weight
joe	1
mary	2
anne	-1

$\mathbb{Z}$-sets generalize sets and multisets: a set can be represented as a $\mathbb{Z}$-set by associating a weight of 1 with each element. Multisets (also called "bags" in the database literature) are $\mathbb{Z}$-sets where all weights are positive. Crucially, $\mathbb{Z}$-sets can also represent *changes* to sets and bags. Negative weights represent rows that are being *removed*.

We can define three operations on $\mathbb{Z}$-sets with values of a given type: (1) **zero** (a $\mathbb{Z}$-set with all weights 0) (2) **negation**: just negate all weights; (3) **plus**: add up the weights of the rows that have the same value. Using these operations $\mathbb{Z}$-sets are a commutative group.

We define the function *distinct* on $\mathbb{Z}$-sets. This function's output is a $\mathbb{Z}$-set where all rows of the input with negative weights are removed, and all positive weights are changed to 1. For example, the *distinct* of the above $\mathbb{Z}$-set is:

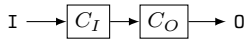
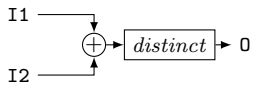
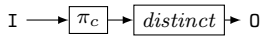
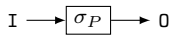
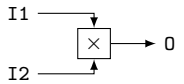
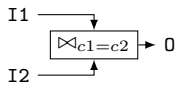
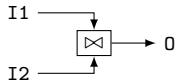
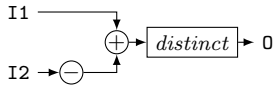
Row	Weight
joe	1
mary	1

Notice that *distinct* "removes" duplicates from multisets, and it also eliminates rows with negative weights.

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. [13]. The translation of the relational operators to functions computing on $\mathbb{Z}$-sets is shown in Table 1. The functions (π, σ, $\bowtie$, $\times$) are the standard relational operators projection, selection, join, Cartesian product. The first row of the table

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

Operation	SQL example	DBSP circuit	Details
Composition	`SELECT ... FROM (SELECT ... FROM I)`		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`		Project each row with its weight unchanged. Add up weights of identical rows.
Filtering	`SELECT * FROM I WHERE P(...)`		P is a predicate applied to each row. Select each row separately. If the row is selected, preserve the weight, else make the weight 0.
Cartesian product	`SELECT I1.*, I2.* FROM I1, I2`		The weight of the pair (a,b) is the product of the the weights of a and b.
Equi-join	`SELECT I1.*, I2.* FROM I1 JOIN I2 ON I1.c1 = I2.c2`		Multiply the weights of the rows that appear in the output.
Intersection	`(SELECT * FROM I1) INTERSECT (SELECT * FROM I2)`		Special case of equi-join when both relations have the same schema.
Difference	`SELECT * FROM I1 EXCEPT SELECT * FROM I2`		*distinct* removes rows with negative weights from the result.

shows that a composite query is translated recursively: implement the sub-queries, and connect them with an arrow. This gives us a recipe for translating an arbitrary relational query plan into a circuit.

The translation is fairly straightforward, but many operators require the application of a *distinct* **to produce sets**. For example, $a \cup b = distinct(a + b)$, $a \setminus b = distinct(a - b)$. Filtering on $\mathbb{Z}$-sets works exactly as filtering on sets, but preserves the weight of each value. Selection on $\mathbb{Z}$-sets works similar to selection on sets, but also preserves the weights.

This is a faithful implementation of the relational algebra — the underlying mathematical theory that underlies modern databases — using $\mathbb{Z}$-sets. This implementation produces an abundance of *distinct* operators, but there are known optimizations for removing some of them.

The following functions in Table 1 are linear: $\sigma, \pi, -, +$. The following functions are bilinear: $\times, \bowtie$. In fact, the only non-linear function is *distinct*. In consequence, all these functions (lifted) have very efficient incremental versions.

To explain why these functions are linear, consider the filtering query from the introduction (`WHERE`). What is the change in the output when we add a new row to the input? It is sufficient to check the predicate for the new row. If the predicate returns `true`, the new row is added to the output. So the change in the output only depends on the change in the input, and not on the actual contents of the input. This is what makes the operation linear.

4.3 Incremental view maintenance

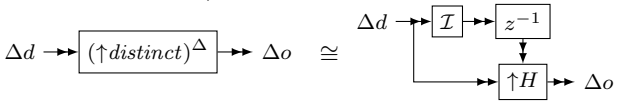
Let us consider a relational query Q defining a view V. To the following algorithm builds a DBSP circuit for Q^Δ:

ALGORITHM 4.1 (INCREMENTAL VIEW MAINTENANCE).

(1) *Translate Q into a circuit using the rules in Table 1.*
(2) *[Optional] Remove some distinct operations.*
(3) *Lift the whole circuit, converting it to a circuit operating on streams, using formula (**) in Section 2.*
(4) *Incrementalize the circuit "surrounding" it with $\mathcal{I}$ and $\mathcal{D}$.*
(5) *Apply the chain rule recursively, producing a circuit using only primitive incremental operations.*

This algorithm is deterministic; the running time is proportional to the number of operators in the query. Step (2) generates an equivalent circuit, with fewer *distinct* operators. Step (3) yields a circuit that consumes a *stream* of complete database snapshots and outputs a stream of view snapshots. Step (4) yields a circuit that consumes a stream of *database changes* and outputs a stream of *view changes*; however, the internal operation of the circuit is non-incremental, as it rebuilds the complete database using integrations. Step (5) optimizes the circuit by replacing each primitive operator with its incremental version. It essentially adds a $\mathcal{I} \circ \mathcal{D}$ pair on every edge in the circuit, and then uses the chain rule to replace each $\mathcal{I} \circ Q \circ \mathcal{D}$ with Q^Δ.

After running this algorithm, all primitives operations are replaced by their incremental versions. The only non-linear operation from Table 1 is *distinct*. However, there is an efficient incremental implementation for *distinct* (this construction has also been known before, but we show it in terms of streaming operations), shown in the following diagram:



The function H has two inputs: the left input is the change Δd, while the top input is the full set, obtained as an integral of the changes. H detects whether the weight of a row in the full set is changing sign (from negative to positive on a row insertion, and from positive to negative on a deletion) when the row appears in a new change. Here is the intuition why *distinct* is efficiently incrementalizable: only tuples that appear in the input change Δd can appear in the output change Δo, so the work performed is $O(|\Delta d|)$. The implementation needs to maintain the *entire input set* (similar to joins) in order to discover whether an item is new or not. That is the purpose of the $\mathcal{I}$ operator.

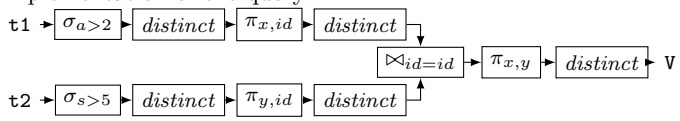
The algorithm reduces the problem of incremental execution of a query plan to the incremental execution of subplans/primitive operators. However, this algorithm works even if we use a primitive P for which no efficient incremental version is known: we can always use the inefficient "brute-force" implementation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

4.4 Relational Query Example

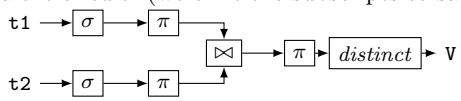
Let's apply the IVM algorithm to the following SQL query:

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

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

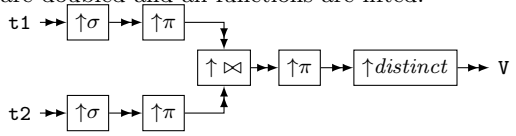


Step 2: eliminate *distinct* operators, producing an equivalent circuit: (we omit the subscripts to save space):

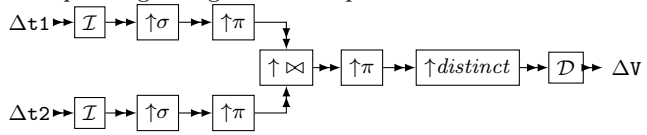


This step is used in some traditional database optimizers. Note that some arrows that represented sets in the original circuit may represent multisets in the optimized circuit.

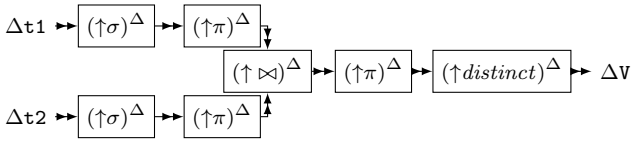
Step 3: lift the circuit to compute over streams; all arrows are doubled and all functions are lifted:



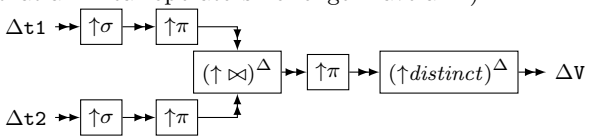
Step 4: incrementalize circuit, obtaining a circuit that computes over changes; this circuit receives changes to relations `t1` and `t2` and for each such change it produces the corresponding change in the output view `V`:



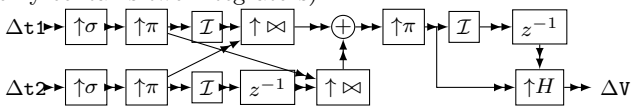
Step 5: apply the chain rule to rewrite the circuit as a composition of incremental operators; notice the use of $\cdot^{\Delta}$ for all operators:



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



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



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

4.5 SQL

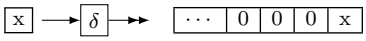
SQL is richer than the relational algebra. It can perform operations on multisets, and it offers operations such as `GROUP BY` and aggregations. All of these can be modeled as operations on $\mathbb{Z}$-set-like structures. Moreover, `GROUP BY` is a linear operation. Some aggregations are "almost" linear, but other, such as `MIN`, require maintaining the full input set, similar to *distinct*, to properly handle deletions. See the full paper and the technical report [6] for more details.

5. RECURSIVE QUERIES

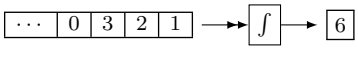
Recursive queries are very useful in many applications. For example, graph algorithms such as graph reachability or transitive closure are naturally expressed using recursive queries. We introduce two simple DBSP stream operators that are used for expressing recursive query evaluation. These operators allow us to build circuits implementing looping constructs, which are used to iterate computations until a fixed-point is reached (i.e., the output of some operator does not change anymore).

5.1 Creating and destroying streams

The **delta function** $\delta : A \to \mathcal{S}_A$ produces a stream from a scalar value. Given an input value x, the output stream is x followed by an infinite number of zeros. The input of δ has a single arrow, while the output has a double arrow.



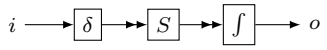
We define the function $\int : \mathcal{S}_A \to A$. Its input stream is required to eventually reach the value 0 and never change afterwards. This function just sums up all the values in the input stream and returns a single result when it encounters the first 0 in the input stream. Notice that the input is a double arrow, while the output is a single arrow. E.g.,:



(This function is also an integrator; its relationship to the $\mathcal{I}$ operator is the same one as the relationship of the definite integral [1] to the indefinite integral [2] in mathematics.)

δ and $\int$ are both linear.

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". However, the δ operator creates a stream with a "new", independent time dimension. We require *well-formed circuits* to "insulate" nested time domains by "bracketing" them between a δ and an $\int$ operator:



S is a streaming operator, but the entire circuit implements a scalar function, as shown by the single arrowheads.

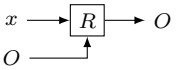
5.2 Implementing recursive queries

We describe the implementation of recursive queries in DBSP. SQL can only express very limited recursive queries, so here we model Datalog queries. In general, a Datalog program defines a set of mutually recursive relations.

We describe the algorithm to build DBSP circuits for the simple case of a single-input, single-output recursive query. The input of our algorithm is a Datalog query of the form $O = R(x, O)$, 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 x. In this section we build a non-incremental circuit, which evaluates the Datalog query; in Section 5.3 we derive the incremental version of this circuit.

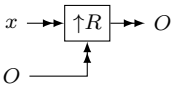
Algorithm 5.1 (recursive queries).

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



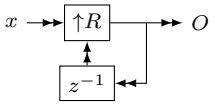
In all these diagrams we show input 0 of operator R on the left, and input 1 on the bottom.

(2) *Lift this circuit to operate on streams:*

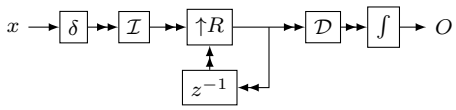


Construct $\uparrow R$ by lifting each operator individually, using equation (**) in Section 2.

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



(4) *"Bracket" the circuit as follows:*



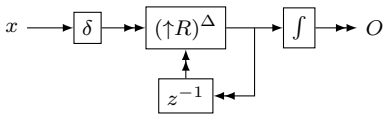
The left input of $\uparrow R$ is an infinite stream of identical values $\boxed{\cdots \mid x \mid x \mid x \mid x}$. The feedback cycle in this circuit is a `while` loop that iterates until no changes are observed (i.e., a fixed-point of R is reached); the outputs produced by $\uparrow R$ will be: in sequence $R(x, 0)$, $R(x, R(x, 0))$, $R(x, R(x, R(x, 0)))$, etc.. The $\mathcal{D}$ operator yields the set of *new changes* computed by each iteration of the loop. When the set of new changes becomes zero, the fixed point has been reached.

Please note that this is **not** a streaming circuit: the input and output arrows are both simple. This is a circuit which receives a single input value and produces a single corresponding output. The circuit uses streams internally to implement the fixed point iteration.

A concrete example for a transitive closure query is Section 8.2 of our technical report [6].

When R, the body of the loop, implements a Datalog programs computing on a finite data domain, this program can be proven to always terminate and compute the least fixed point that contains x. For an arbirary function R, the resulting circuit may loop forever for some inputs.

In fact, this circuit implements the standard Datalog **naïve evaluation** algorithm (e.g., see Algorithm 1 in [11]). 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 we can rewrite this circuit:



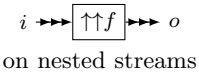
This circuit implements **semi-naïve evaluation** (Algorithm 2 in [11]). We have just proven the correctness of semi-naïve evaluation as an immediate consequence of the cycle rule!

5.3 Incremental recursive programs

In Section 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 Section 5 we showed how to compile a recursive query into a circuit that employs incremental computation internally, 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*.

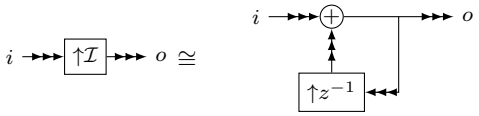
In the same way streams are infinite vectors, streams of streams are infinite matrices. We denote streams of streams with triple arrows in our diagrams.

The same way we lift functions to produce stream operators, we can lift stream operators to produce operators on streams of streams. A scalar function f can be lifted twice to produce an operator between streams of streams:

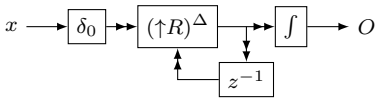


The operator z^{-1} on nested streams delays "rows" of the matrix, while $\uparrow z^{-1}$ delays "columns".

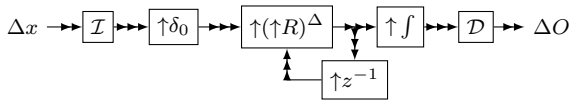
We have seen in equation (**) that lifting a graph entails lifting all operators. This extends to graphs with cycles, e.g:



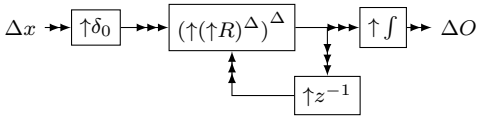
This gives us the ability to lift entire circuits, including circuits computing on streams and having feedback edges. With this machinery we can now apply Algorithm 4.1 to arbitrary circuits, even circuits for recursive relations.
Step 1: Start with the "semi-naive" circuit:



Step 2: nothing to do for *distinct*.
Steps 3 and 4: Lift the circuit and incrementalize:



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



This is the incremental version of a recursive query. A concrete example for a transitive closure query is Section 9.1 of our technical report [6].

6. IMPLEMENTATION

DBSP does not make any simplifying assumptions that would make it impractical. In fact, Feldera Inc. has built an open-source implementation of DBSP as a query engine in Rust [9]; and also a compiler from SQL to DBSP [10]. This compiler handles essentially the entire SQL language. The compiler generates execution plans for incrementally maintaining any number of views defined in SQL.

Plan quality. A relational algebra query can be implemented by multiple plans, each with a different data-dependent cost. The input of Algorithm 4.1 is a 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 query. Nevertheless, all standard query optimization techniques, perhaps based on historical statistics, can be used to generate the query plan that is supplied to our Algorithm. 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*.

Tradeoffs. Incremental computation is not free. It is in fact a trade-off between time and space. In the cost analysis we have to consider both the time and the space used by each operator. While many incremental database operations can be implemented using work proportional to the size of the changes, and no storage overhead, several classes of database operations, such as joins, "distinct", and aggregates can be implemented efficiently only using additional storage in the form of *indexes*. The size of these indexes is proportional to the size of the total data in the database (and not just to the size of the changes) — and since some indexes are over intermediate relations, they can even exceed the size of the original database. In DBSP the indexes are represented by delay operators z^{-1}. In fact, the delay operator (and its lifted variant $\uparrow z^{-1}$) are the only operators that maintains state. This is also the only state that needs to be persisted, checkpointed, or migrated to make DBSP computations fault-tolerant.

DBSP is an "eager" or "top-down" execution model: it constantly maintains the entire contents of any number of views, even if no one really wants to inspect the views. In contrast, "lazy" or "bottom-up" models only build part of the views when the views are inspected. Such models have the potential to be more efficient. Eager models can be converted into lazy ones if something is known about the class of operations that will be executed against the views.

Start-up costs. When a new view is installed, the IVM system must compute the first change, which is the same as the initial contents of the view. This computation is in proportional to the size of the whole database. This is known as the "backfill" problem. Likewise, changes to the definition of a view or the data schema require recomputing the affected queries from scratch.

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

Moreover, we argue that databases should not only compute views incrementally, but should use "changes" as the fundamental data structure to communicate with their environment: a database service should offer the following API: users register to receive notifications for changes in one or more views. Then, for any transaction committed, each user receives a notification containing the list of changes for the all the views they registered. Databases today do not have convenient mechanism for reporting changes to the outside world. In fact, entire industries have sprung up around the concept of Change Data Capture [3], which is building ad-hoc solutions for extracting changes from databases, usually by inspecting the write-ahead transaction log.

7. CONCLUSIONS

We have introduced DBSP, a model of computation based on infinite streams over commutative 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.

8. REFERENCES

[1] `https://en.wikipedia.org/wiki/Integral`. Retrieved March 2024.

[2] `https://en.wikipedia.org/wiki/Antiderivative`. Retrieved March 2024.

[3] `https://en.wikipedia.orag/wiki/Change_data_capture`. Retrieved March 2024.

[4] Causal system. `https://en.wikipedia.org/wiki/Causal_system`. Retrieved March 2024.

[5] M. Budiu, T. Chajed, F. McSherry, L. Ryzhyk, and V. Tannen. DBSP: Automatic incremental view maintenance for rich query languages. In *Proceedings of the VLDB Endowment (VLDB)*, volume 16, pages 1601–1614, Vancouver, Canada, August 2023. Best paper award.

[6] M. Budiu, F. McSherry, L. Ryzhyk, and V. Tannen. DBSP: A language for expressing incremental view maintenance for rich query languages. https://github.com/feldera/feldera/blob/main/papers/spec.pdf, December 2022.

[7] T. Chajed. DBSP formalization. `https://github.com/tchajed/dbsp-theory`, Dec. 2022.

[8] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. The Lean theorem prover. In *International Conference on Automated Deduction (CADE-25)*, Berlin, Germany, 2015.

[9] Feldera Inc. DBSP Rust crate. `https://crates.com/crates/dbsp`. Retrieved March 2024.

[10] Feldera Inc. SQL to DBSP compiler. `https://github.com/feldera/feldera/tree/main/sql-to-dbsp-compiler`. Retrieved March 2024.

[11] S. Greco and C. Molinaro. Datalog and logic databases. *Synthesis Lectures on Data Management*, 7(2):1–169, 2015.

[12] T. J. Green, Z. G. Ives, and V. Tannen. Reconcilable differences. *Theory of Computing Systems*, 49(2):460–488, 2011.

[13] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *Symposium on Principles of Database Systems (PODS)*, page 31–40, Beijing, China, June 11-14 2007.

[14] A. Gupta, I. S. Mumick, et al. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[15] C. Koch. Incremental query evaluation in a ring of databases. In *Symposium on Principles of Database Systems (PODS)*, page 87–98, Indianapolis, Indiana, USA, 2010.

[16] L. R. Rabiner and B. Gold, editors. *Theory and Application of Digital Signal Processing*. Prentice-Hall, 1975.