Incremental Query Evaluation in a Ring of Databases

Christoph Koch

École Polytechnique Fédérale de Lausanne
christoph.koch@epfl.ch

Abstract

This article approaches the incremental view maintenance problem from an algebraic perspective. The algebraic structure of a ring of databases is constructed and extended to form a powerful aggregate query calculus. The query calculus inherits the key properties of rings, such as distributivity and the existence of an additive inverse. As a consequence, the calculus has a normal form of polynomials and is closed under a universal difference operator. This difference operator allows to express the so-called delta queries of the incremental view maintenance literature, but also deltas to the deltas (second deltas), deltas to second deltas (third deltas), and so on. The k-th delta of a query of polynomial degree k is purely a function of the update, not of the database. This gives rise to a multi-layered incremental view maintenance scheme in which a view is maintained using a hierarchy of auxiliary materialized views of k-th deltas. What is gained by this hierarchy is that the work required to keep all views fresh given an update is extremely simple.

The method allows to eliminate expensive query operators such as joins and aggregate sums entirely from programs that perform incremental view maintenance. The main result is that, for non-nested queries, each individual aggregate value can be incrementally maintained using a constant amount of work. This is not possible for nonincremental evaluation and provides a complexity separation between the incremental and nonincremental query evaluation problems.

As a byproduct, we obtain a query language that is significant in its own right. It is an algebraic language in which queries, like in relational calculus, are built up from base objects (generalized relations) using an extremely small set of connectives – addition, multiplication, and aggregation. It is based on a family of algebraic structures developed in this article – called *avalanche (semi)rings* – which algebraizes range-restriction. Thus these structures guarantee finite query results in the presence of inequalities, without making use of an explicit selection operation. The entire language behaves like a polynomial ring of relations and thus makes algebraic manipulation very easy. As a simple algebraic language of interesting expressive power – relational algebra with SQL-style aggregation and unlimited nesting – it is a natural internal representation language for query processors and compilers.

1 Introduction

Most databases evolve incrementally, through changes that are small compared to the overall database size. Memoization of the result of a repeatedly asked query can substantially reduce

the amount of work needed to re-evaluate the query after moderate changes to the database. Using stored query results or auxiliary data structures to this effect is known as *incremental view maintenance*, and has been studied extensively in the past [5, 32, 4, 31, 6, 15, 16, 14, 35, 9, 11, 7, 26, 27, 29, 22].

The notion of a *delta query* is key to incremental view maintenance. Let u denote a change ("update", which captures both insertions and deletions) to database D. Denote the updated database by $D + u$, where $+$ is a way of combining a database with a change to it, made precise later. In the case that u is an insertion into D, $+$ is just the union operation of relational algebra. Denote the result of a query Q on a database D by $Q(D)$. A delta query ΔQ expresses the change to the result of query Q as database D is updated to $D + u$: $Q(D + u) = Q(D) + \Delta Q(D, u)$.

Most work on incremental view maintenance revolves around delta queries and their use according to this formula. The intuition is that evaluating $\Delta Q(D, u)$ and using the result to update a materialized representation of $Q(D)$ is often faster than recomputing $Q(D + u)$ from scratch, because ΔQ is usually a simpler query than Q.

The practical benefits of incremental view maintenance are real and have led to the integration of such techniques into commercial database management systems. Can we also make a complexity-theoretic argument that incremental view maintenance is more efficient than nonincremental query evaluation?

The following argument suggests the answer is no: If we consider a query language L with multiset semantics that is closed under joins (such as SQL), the image of L under taking deltas is the full language L: Given an arbitrary query $Q_0 \in L$, there is another query $Q \in L$ and a single-tuple insertion u such that $\Delta Q(D, u) = Q_0(D)$ on any database D. To see this, just let u be an arbitrary single-tuple insertion into a relation R that does not occur in Q_0. Then query $Q_0 \times \pi_\emptyset R$ in multiset relational algebra is such a Q. This suggests that incremental view maintenance is not fundamentally easier than nonincremental query evaluation.

This article demonstrates that this intuition is fortunately incorrect: *Incremental evaluation is provably easier than nonincremental evaluation.* To obtain this result, we have to approach the problem from a fresh perspective and look beyond classical incremental view maintenance based on the *evaluation* of delta queries. As odd as this may sound, we have to evade any query evaluation in the classical sense of the word whatsoever; even more oddly, *unlimited procrastination and delegation* (to delta queries) does the job. The main result that achieves the complexity separation is in the realm of parallel complexity, but the actual technique is relevant when used on sequential computers as well.

1.1 Recursive Delta Procesing

To get an intuition for the incremental query evaluation technique that allows us to achieve all this, we first consider the core technique abstractly.

Given a function $f : A \times \Theta \to A'$ (where Θ is the product of zero or more sets, so f has one or more arguments overall), binary operations $+$ defined on sets A and A', and a set $U \subseteq A$ (the possible "updates"). To quantify the impact of changing x on $f(x, \theta)$ as the other arguments θ remain unchanged, define functions

$$\Delta^j f : A \times \Theta \times U^j \to A'$$

of j-th deltas of f for each $j \geq 0$ in such a way that $(\Delta^0 f) = f$, that is, the 0-th delta of f is

	$f(x)$						
		$\Delta f(x,-1)$				$\Delta f(x,+1)$	
x	$\Delta^2 f(x,-1,-1)$		$\Delta^2 f(x,-1,+1)$		$\Delta^2 f(x,+1,-1)$		$\Delta^2 f(x,+1,+1)$
-2	2	5	-2	4	-2	-3	2
-1	2	3	-2	1	-2	-1	2
0	2	1	-2	0	-2	1	2
1	2	-1	-2	1	-2	3	2
2	2	-3	-2	4	-2	5	2
3	2	-5	-2	9	-2	7	2
4	2	-7	-2	16	-2	9	2

Figure 1: Recursive memoization of deltas for $f(x) = x^2$: the seven function values to be memoized for various values $x \in \mathbb{Z}$. (See Example 1.1.)

just f, and

$$(\Delta^j f)(x,\theta) + (\Delta^{j+1} f)(x,\theta,u) = (\Delta^j f)(x+u,\theta)$$

for $\theta \in \Theta \times U^j$ and $u \in U$. Think of $(\Delta^{j+1} f)(x,\theta,u)$ as the amount of change that happens to the value of $\Delta^j f(x,\theta)$ as the value of x is changed to $x+u$ ("updated by u") while the other arguments θ remain unchanged. The updated version of the j-th delta is just the old version of the j-th delta plus the $(j+1)$-st delta. To render this less abstract, take the case of a unary function $f(x)$ and $j = 0$: Then this just says that $f(x) + (\Delta f)(x,u) = f(x+u)$, the equation that motivated delta queries above.

If A' has an additive inverse, the functions Δ^j $(j > 0)$ can be defined inductively by

$$\begin{aligned} (\Delta^0 f) &:= f \\ (\Delta^{j+1} f)(x,\theta,u) &:= (\Delta^j f)(x+u,\theta) - (\Delta^j f)(x,\theta). \end{aligned}$$

We require the following: *Function f must have the property that there exists $k \in \mathbb{N}$ such that $\Delta^k f$ is $0^{A'}$ on all inputs, and that we can determine k statically from the definition of f.*

We assume that it is easy to perform additions in A', but expensive to compute f (and generally, $\Delta^j f$) from its arguments. Assume there is a value $x \in A$ that is considered current. We would like to allow for x to be updated with increments from a set of possible updates $U \subseteq A$ and to quickly obtain $f(x,\cdot)$ without having to recompute it "from scratch" after each change to x. Suppose that for all $0 \le j < k$ and $\theta \in \Theta \times U^j$, the values $\Delta^j f(x,\theta)$ for the current x are known (and, as required, $\Delta^k f(\cdot,\cdot) = 0$). On update event $x_{new} := x_{current} + u$, we compute

$$\Delta^j f(x_{new},\theta) := \Delta^j f(x_{current},\theta) + \Delta^{j+1} f(x_{current},\theta,u). \tag{1}$$

This takes only additions of memoized $\Delta^j f$ and $\Delta^{j+1} f$ values. None of these values have to be recomputed from their (function) definitions. If we compute the sums in order of increasing j, we do not need to double-buffer but can update the values in place, i.e., set $\Delta^j f(x,\theta) \mathrel{+}= \Delta^{j+1} f(x,\theta,u)$.

Example 1.1 (The polynomial ring $\mathbb{R}[x]$) Consider the polynomial functions over the real numbers $f : \mathbb{R} \to \mathbb{R}$. Then Δf is a polynomial of degree one less than the degree of f, unless f is constant. It follows that there is a finite k (the degree of polynomial f plus 1) such that the k-th deltas of f are 0. So the technique just sketched is applicable.

Take $f(x) = x^2$.

$$\begin{aligned}
\Delta f(x, u_1) &:= f(x + u_1) - f(x) \\
&= (x + u_1)^2 - x^2 = 2u_1x + u_1^2. \\
\Delta^2 f(x, u_1, u_2) &:= \big(\Delta f(x + u_2, u_1)\big) - \Delta f(x, u_1) \\
&= 2u_1(x + u_2) + u_1^2 - (2u_1x + u_1^2) = 2u_1u_2. \\
\Delta^3 f(x, u_1, u_2, u_3) &:= \big(\Delta^2 f(x + u_3, u_1, u_2)\big) - \Delta^2 f(x, u_1, u_2) \\
&= 2u_1u_2 - 2u_1u_2 = 0.
\end{aligned}$$

In the framework introduced above, $A = A' = \mathbb{R}$ and $\Theta = \{\langle\rangle\}$. Let the set of possible updates be $U = \{+1, -1\}$. Then we need to memoize $|U|^0 + |U|^1 + |U|^2 = 7$ function values[1], and initially compute values for them from the definitions of f, Δf, and $\Delta^2 f$. That is, we compute these seven values from their definitions only once, for the initial x value that we decide to start with (e.g., 0). After that, the function definitions are not used anymore: We only perform additions of the current seven values.

Figure 1 shows these values for $x = -2, -1, 0, \ldots, 4$. At any point in time (for any current x value), we hold one row of this table – with seven values – in memory. This is all we need to increment or decrement x by one. For $x = -1, \ldots, 3$ this corresponds to changing the memoized row to its predecessor or successor in the table of Figure 1. We only need to add, to each memoized value, one other memoized value. For example, let $x = 3$ and we increment x by 1. Then $f(\cdot)$ += $7 = 16$, $\Delta^1 f(\cdot, +1)$ += $2 = 9$, $\Delta^1 f(\cdot, -1)$ += $-2 = -7$, and $\Delta^2 f(\cdot, \cdot, \cdot)$ += 0. □

We can turn this memoization technique into an incremental view maintenance algorithm as follows. The functions f from above are now queries, A is a domain of databases, and A' is the domain of query results. For this to work, the query language L must have two properties,

1. it has to be closed under taking applying deltas: for each query Q of L, ΔQ has to be expressible in L, and L has to support the addition of a query and its delta.

2. for each query, there must be a $k \in \mathbb{N}$ such that the k-th delta query returns constantly $0^{A'}$ (the empty database).

We simply apply the technique sketched above with the set of possible updates U fixed to be the set of single-tuple insertions and deletions. Given a query f, this technique memoizes a hierarchy of materialized views. The j-th view ($0 \leq j < k$) consists of a tuple

$$\langle \theta, u_1, \ldots, u_j, \Delta^j f(x, \theta, u_1, \ldots, u_j) \rangle$$

for $\theta \in \Theta$ and $u_1, \ldots, u_j \in U$. For the view to be finite, U has to be finite.[2]

[1] In fact, since the four Δ^2 values are constant, only three values need to be kept up to date.

[2] If U is the active domain of the database, which may grow through updates, then initial values for view entries that have not been incrementally maintained so far have to be computed. More will be said about this in the body of the article.

Once the values $\Delta^j f(\cdot, \theta, u_1, \ldots, u_j)$ have been initialized and put into the materialized views, the technique does not access the current database x itself. Indeed, $\Delta^j f(x, \theta, u_1, \ldots, u_j)$ is a placeholder for a value that is kept up to date and is identified by $\langle j, \theta, u_1, \ldots, u_j \rangle$ but not x. It is not a function call that is ever evaluated, so the value of x is not needed.

If all we are interested in is the views, we do not need to store the database, unless it is to check integrity constraints, such as that tuples that do not exist in the database are not attempted to be deleted.

Example 1.2 Consider the following (somewhat artificial) SQL query over unary relation R with schema A and multiset semantics:

$$Q(R) = \text{select count(*) from R r1, R r2 where r1.A = r2.A}$$

The delta queries for single-tuple insertions (denoted $+R(a)$) and deletions (denoted $-R(a)$) are

$$\begin{aligned}
\Delta Q\big(R, \pm R(a)\big) &= 1 \pm 2 \ * (\text{select count(*) from R where R.A=a}) \\
\Delta^2 Q\big(R, \pm_1 R(a_1), \pm_2 R(a_2)\big) &= \begin{cases} \pm_1 \pm_2 2 & \ldots \quad a_1 = a_2 \\ 0 & \ldots \quad \text{otherwise} \end{cases} \\
\Delta^3 Q(R, \cdot, \cdot, \cdot) &= 0.
\end{aligned}$$

The above definitions give $Q(\emptyset) = 0$ and $\Delta Q(\emptyset, \pm R(\cdot)) = 1$. Let us start on the empty database and perform some insertions and deletions. (Multiset relations are denoted by $\{\!\{ \cdot \}\!\}$.)

			$\Delta Q(R, \cdot)$			
Update	R	$Q(R)$	$+R(c)$	$-R(c)$	$+R(d)$	$-R(d)$
	$\emptyset$	0	1	1	1	1
$+R(c)$	$\{\!\{c\}\!\}$	1	3	-1	1	1
$+R(c)$	$\{\!\{c, c\}\!\}$	4	5	-3	1	1
$+R(d)$	$\{\!\{c, c, d\}\!\}$	5	5	-3	3	-1
$+R(c)$	$\{\!\{c, c, c, d\}\!\}$	10	7	-5	3	-1
$-R(d)$	$\{\!\{c, c, c\}\!\}$	9	7	-5	1	1
$+R(c)$	$\{\!\{c, c, c, c\}\!\}$	16	9	-7	1	1
$-R(c)$	$\{\!\{c, c, c\}\!\}$	9	7	-5	1	1

In the above table, $\Delta^2 Q$ is omitted for space reasons. It is constant (does not depend on the database), and $\Delta^2 Q(R, +R(a), +R(a)) = \Delta^2 Q(R, -R(a), -R(a)) = 2$, $\Delta^2 Q(R, +R(a), -R(a)) = \Delta^2 Q(R, -R(a), +R(a)) = -2$, and $\Delta^2 Q(R, \pm_1 R(a_1), \pm_2 R(a_2)) = 0$ for $a_1 \neq a_2$.

We only evaluate queries (and access the database) to initialize the memoized values. After that, we use the update rule from above (Equation (1)) to apply an update u. □

This incremental view maintenance technique uses delta processing more aggressively than classical approaches: Rather than being content with materializing view Q and using ΔQ to update it, delta processing is performed recursively: We also materialize ΔQ, incrementally maintain it using $\Delta^2 Q$, materialize that, and so on.

1.2 Contributions of the Remainder of this Article

We are now left with the problem of finding a query language that has the two properties we have required above. This is achieved in this article by the definition of a query language that is in close analogy with the polynomials of Example 1.1. The language captures a large class of SQL (aggregate) queries.

The first contribution of this article is to craft the algebraic structure of a *ring of databases*. Its addition operation generalizes relational union, and multiplication generalizes the natural join. The elements of the ring are generalizations of multi-set relational databases in which tuples can have negative multiplicities (to model deletions). The article shows that the generalizations are necessary and in a sense the unique solution that admits a full additive inverse (necessary for delta processing) and distributivity (to yield polynomials).

This ring is the basis of the definition of an expressive aggregate query calculus, AGCA, which inherits these key properties. For an AGCA query Q, ΔQ always exists and is structurally strictly simpler than Q. We formalize this by the notion of the degree of a query. The k-th delta of an AGCA query of degree k without nested aggregates has degree 0; a query of degree 0 only depends on the update but not on the database.

Thus, the recursive delta processing technique sketched above is applicable to AGCA: Each query can be incrementally maintained by a hierarchy of materialized views, requiring only basic summations of values in adjacent layers of the view hierarchy to keep all views up to date. As we have seen, only a single arithmetic operation is required per single-tuple update and per value maintained in this hierarchy of views. For the purpose of aggregate query processing, these values are numerical. In practice, such values are usually represented using fixed-size memory words (e.g., 64-bit integers or floating point numbers). An easy argument shows that in this scenario, updating the view hierarchy given a single-tuple update is in NC0.

Recall that the circuit complexity classes NC0 $\subseteq$ AC0 $\subsetneq$ TC0 (cf. [20]) all represent *constant-time parallel computation* using polynomial amounts of hardware. The difference lies in the types of gates used. While NC0 uses only bounded fan-in gates, both AC0 and TC0 use *unbounded* fan-in gates, which are unrealistic. AC0 and TC0 problems cannot be solved in constant parallel time on any amount of bounded fan-in ("real") hardware. In contrast, in an NC0 circuit, the value of an output bit depends only on a constant number of input bits. NC0 problems admit even more extreme parallelism than AC0 or TC0 problems. AC0 and TC0 problems ultimately require a single computation node (or gate of a circuit) to compute a bit whose value depends on, and is an aggregation of, an unbounded number of input bits. NC0 problems do not require such an aggregation.

This article shows that, for a large class of aggregate queries, applying a fixed update to a materialized view is in the complexity class NC0. In comparison, it is known that non-incremental evaluation takes TC0 and AC0 for queries with and without aggregates, respectively (cf. e.g. [24]). Since NC0 and TC0 have been separated [33], incremental evaluation is indeed provably easier than nonincremental evaluation. Moreover, this shows that incremental query evaluation can completely *eliminate both aggregation and joins* from a large class of aggregate queries!

A practical compilation algorithm. Having achieved theoretical success, a second goal is to make this technique also practically useful. To achieve this, we have to address the question of how to maintain finite *active domains* of the memoized functions, and how to extend these when the updates require it.

Moreover, a j-th delta is a function of a j-tuple of update tuples, which means that its domain and the (say, tabular) representation of the memoized function may become large. This is no problem for the parallel complexity result, but it defeats the practical purpose of incremental view maintenance (which is to gain performance).

In the article, we refine the aggressive recursive incremental view maintenance scheme by employing query factorization and simplification of queries, which allows us to create small representations of the materialized views. This, however, requires substantial refinements of the technique. The article presents an algorithm for compiling queries to a simple low-level language, NC0C. This is essentially a small fragment of the programming language C.

Example 1.3 (factorization of delta queries, cf. [3]) Consider the relational database schema $R(A, B)$, $S(C, D)$, $T(E, F)$ and the query

$$Q = \text{select sum(A*F) from R, S, T where B=C and D=E}$$

Then

$$\begin{aligned}\Delta Q(\pm S(c,d)) &= \pm(\text{select sum(A*F) from R, T where B=c and d=E}) \\ &= \pm \underbrace{(\text{select sum(A) from R where B=c})}_{(\Delta Q)_1(c)} * \underbrace{(\text{select sum(F) from T where d=E})}_{(\Delta Q)_2(d)}\end{aligned}$$

This factorization is possible because the aggregate sum operator of SQL (with its multiset semantics) is distributive over multiplication.

Thus, rather than maintaining ΔQ (which in general takes space quadratic in the active domain size), we can maintain two linear size views $(\Delta Q)_1$ and $(\Delta Q)_2$ and update Q on update event $\pm S(c, d)$ as $Q \pm= (\Delta Q)_1(c) * (\Delta Q)_2(d)$ rather than as $Q \pm= \Delta Q(c, d)$. □

The article contains a proof that NC0C update triggers require only a constant number of arithmetic operations ($+$ and $*$) per value updated in the materialized views (Proposition **??**). Moreover, each bit of the materialized representation can be incrementally maintained using an NC0 circuit, which only reads from a constant number of input bits, assuming that numbers are kept in fixed-size registers (Theorem **??**).

In most traditional database query processors that perform incremental view maintenance, the basic building blocks of delta queries are large-grained operators such as joins. By our compilation approach we are instead able to eliminate all joins and evaluate queries without resorting to classical query operators.

The article is structured as follows. After providing some algebraic foundations in Section 2, we define a ring of multiset relations and discuss some of its properties in Section 3. We define the aggregate query language AGCA in Section 4. Section 5 discusses the query language and shows how to turn queries into polynomials and how to factorize monomials. Section 6 studies delta processing. We give a construction for deltas and show that, for a large class of queries, deltas are structurally simpler than the input queries. Section **??** introduces a low-level language, NC0C, to which we later compile queries. NC0C admits massively parallel evaluation; we give a circuit complexity characterization. Section **??** presents algorithms for compiling queries to NC0C, starting with a simple algorithm that we subsequently refine. Section 8 discusses this and related work.

The ring construction of this article allows for a streamlined and cleaner way of presenting incremental view maintenance that clearly exposes the key ideas that render it possible. One may hope and expect that future survey and textbook presentations of the broader topic will profit from the foundational exposition provided below. To this end, portions of this article that cover notions from mainstream mathematics that are rarely seen in the database context (particularly, in Section 2) are presented in a succinct but essentially self-contained survey style.

2 Algebraic Foundations

We first fix some notational conventions in Section 2.1. After that, (1) in Section 2.2, we provide basic definitions from abstract algebra and introduce monoid rings, which are structures that, as we will see later, can accommodate joins in relational databases. (2) In Section 2.3, we introduce an abstraction of monoid rings called *avalanche rings* that can deal with a form of function composition (sideways binding passing in database query languages) and still retains a ring structure. Support for sideways binding passing is needed for technical reasons, to support the mechanics of query languages (operator chaining and range restriction) and to accommodate limited interfaces with binding patterns (conditions and built-in functions). (3) In Section 2.4, we provide a construction for deriving smaller structures from monoid and avalanche rings which retain their key properties. (4) In Section 2.5, we show that there is essentially no other way than monoid rings of obtaining rings whose multiplication operation generalizes joins: If we want joins and rings, we have to go with monoid rings.

While the core problem studied in this paper – incremental view maintenance – calls for ring structures, we generalize our results to semirings where easily possible; this generalization is not needed in later sections, but renders our algebraic framework applicable to database query processing with set semantics (cf. e.g. [13, 12]). The polynomial avalanche semirings, which are at the same time an elegant algebraic structure and a database query language, are an application of this.

In order to avoid cluttered notation at this point, we do not yet apply this framework to databases, but defer this to Section 3. As a consequence, this section is devoid of (database) examples, and the reader is asked for patience.

2.1 Notational Conventions

By convention, we superscript relations and functions with the name of the structure that they belong to. For example, $R^{\mathcal{A}}$ is the relation named R in database $\mathcal{A}$ and $+^{\mathbb{Z}}$ is the addition operation in the ring of integers $\mathbb{Z}$. To avoid confusion with powers, names of structures are never numbers, and are never represented by lower case variable names: thus, if f is a function, f^k denotes the k-th power of (the image of) f. We may omit these superscripts when they are clear from the context.

We use the notations $f : x \mapsto v$, $f(x) := v$, and $f := \{x \mapsto v \mid x \in \mathrm{dom}(f)\}$ interchangeably to define functions, with a preference for the latter when the domain $\mathrm{dom}(f)$ might otherwise remain unclear. We write $f|_D$ to denote the restriction of the domain of f to D, i.e. $f|_D := \{(x \mapsto v) \in f \mid x \in D\}$. A function $f : A \to B$ is called *surjective* if, for all $b \in B$, there is an $a \in A$ such that $f(a) = b$.

2.2 (Semi)rings

Recall some basic definitions from algebra (cf. e.g. [23, 10]):

Definition 2.1 A *semigroup* is a pair $(A, \circ)$ of a base set A and a binary total function $\circ : A \times A \to A$ ("the operation") such that $\circ$ is *associative*, that is, for all $a, b, c \in A$, $(a \circ b) \circ c = a \circ (b \circ c)$. A semigroup is called *commutative* if $a \circ b = b \circ a$ for all $a, b \in A$. A *monoid* $(A, \circ, e)$ is a semigroup that has *neutral element* $e \in A$, that is, $a \circ e = e \circ a = a$ for all $a \in A$. A monoid is called a *group* if for each $a \in A$ there is an *inverse element* $a^{-1} \in A$ such that $a \circ a^{-1} = a^{-1} \circ a = e$.

A *semiring* over base set A is a tuple $(A, +, *, 0, 1)$ with two operations $+$ and $*$ (called addition and multiplication, respectively) such that $(A, +, 0)$ is a commutative monoid, $(A, *, 1)$ is a monoid, $0 * a = a * 0 = 0$ for all $a \in A$, and $+$ and $*$ are *distributive*, that is, $a * (b + c) = a * b + a * c$ and $(a + b) * c = a * c + b * c$ for all $a, b, c \in A$.

A *ring* over base set A is a tuple $(A, +, *, 0)$ such that $(A, +, 0)$ is a commutative group, $(A, *)$ is a semigroup, and $+$ and $*$ are *distributive*. A ring *with identity* $(A, +, *, 0, 1)$ is a ring in which $(A, *, 1)$ is a monoid. A (semi)ring is called commutative if $*$ is commutative.

Example 2.2 The integers $\mathbb{Z}$ and the rational numbers $\mathbb{Q}$ form commutative rings with identity $(\mathbb{Z}, +, *, 0, 1)$ and $(\mathbb{Q}, +, *, 0, 1)$. The natural numbers $\mathbb{N}$ do not form a ring because there is no additive inverse; for example, there is no natural number x such that $1 + x = 0$; but they do form a semiring. The booleans $\mathbb{B} = \{\text{true}, \text{false}\}$ form a semiring $(\mathbb{B}, \vee, \wedge, \text{false}, \text{true})$ in which disjunction $\vee$ takes the role of addition and conjunction $\wedge$ takes the role of multiplication. □

Neutral elements 0 and 1 are named by analogy and are not necessarily numbers. For a group with an operation $+$, we write $-a$ to denote a^{-1} and use the shortcut $a - b$ for $a + (-b)$. When the operations $+$ and $*$ are clear from the context, we will also use the name of the base set to refer to the structure (e.g., $\mathbb{Z}$ for $(\mathbb{Z}, +, *, 0, 1)$).

In a monoid, there is a *unique* identity element. In a group $(A, \circ, e)$, there is a *unique* inverse element a^{-1} for each element $a \in A$ (cf. e.g. Proposition 1 in Chapter 1 of [10]). A (semi)ring is uniquely determined by its base set and its operations $+$ and $*$, and we do not need to explicitly specify 0, 1, or the operation $(\cdot)^{-1}$ (but it will be done for the reader's convenience). In a ring with identity, $0 * a = a * 0 = a$ (thus it is a semiring) and $(-a) * b = -(a * b) = a * (-b)$ for all $a, b \in A$.

Definition 2.3 Let A be a semiring and let G be a monoid. Let $A[G]$ be the set of all functions $\alpha : G \to A$ such that $\alpha(x) = 0$ for all but a finite number of $x \in G$ (that is, α has *finite support*). We define addition and multiplication in $A[G]$ as

$$\begin{aligned} \alpha + \beta &: x \mapsto \alpha(x) +^A \beta(x) \\ \alpha * \beta &: x \mapsto \sum_{x = y *^G z} \alpha(y) *^A \beta(z). \end{aligned}$$

We call $A[G]$ the *monoid semiring* of G over A. If A is a ring with identity, $A[G]$ is called the *monoid ring* of G over A. If A is also commutative, then $A[G]$ is called the *monoid algebra* of G over A. □

The neutral elements are $0 : x \mapsto 0^A$ and

$$1 : x \mapsto \begin{cases} 1^A & \ldots \quad x = 1^G \\ 0^A & \ldots \quad x \neq 1^G. \end{cases}$$

The following is well known (cf. e.g. p.104f of [23]):

Proposition 2.4 *(1) $A[G]$ is a semiring. (2) If A has an additive inverse, then $A[G]$ is a ring with identity. (3) If both A and G are commutative, then $A[G]$ is commutative.*

In particular, monoid semirings $A[G]$ are closed under multiplication, i.e., any product $\alpha * \beta$ has finite support.

2.3 Avalanche semirings

We next define a structure of higher-order functions to accommodate binding passing in query languages.

Definition 2.5 Let $A[G]$ be the monoid semiring of monoid G over semiring A. Let $\overrightarrow{A[G]}$ be the set of all functions $G \to A[G]$. We define addition and multiplication in $\overrightarrow{A[G]}$ as

$$\begin{aligned} f + g \quad &: \quad b \mapsto x \mapsto f(b)(x) +^A g(b)(x) \\ f * g \quad &: \quad b \mapsto x \mapsto \sum_{x = y *^G z} f(b)(y) *^A g(b *^G y)(z). \end{aligned}$$

Then $\overrightarrow{A[G]}$ is called the *avalanche semiring* of G over A. □

Theorem 2.6 *$\overrightarrow{A[G]}$ is a semiring. If A is a ring with identity, then $\overrightarrow{A[G]}$ is a ring with identity.*

Proof. Define $0^{\overrightarrow{A[G]}} : \cdot \mapsto 0^{A[G]}$ and $1^{\overrightarrow{A[G]}} : \cdot \mapsto 1^{A[G]}$; that is, these functions ignore their argument and return $0^{A[G]}$ and $1^{A[G]}$, respectively.

The structure $(\overrightarrow{A[G]}, +^{\overrightarrow{A[G]}}, 0^{\overrightarrow{A[G]}})$ inherits the property of being a commutative monoid resp. commutative group from $(A[G], +^{A[G]}, 0^{A[G]})$ if A is a semiring resp. ring with identity. It is easy to see this: in particular, $(f +^{\overrightarrow{A[G]}} g)(b) = f(b) +^{A[G]} g(b)$, and $+^{A[G]}$ has the desired properties.

The structure $(\overrightarrow{A[G]}, *^{\overrightarrow{A[G]}}, 1^{\overrightarrow{A[G]}})$ is a monoid. $\overrightarrow{A[G]}$ is closed under multiplication, so $*$ is an operation. To verify this, we have to show that a product $f * g$ is a function that returns on any input b an element $(f * g)(b) \in A[G]$. That is, $f(b)$ can be nonzero only on a finite number of inputs from G. If we revisit the definition of $*^{\overrightarrow{A[G]}}$, we see that there are only a finite number of products $x = y * z$ of elements y, z for which $f(b)(y)$ and $g(b * y)(z)$ are nonzero (since $f(\cdot), g(\cdot) \in A[G]$).

Moreover, $*^{\overrightarrow{A[G]}}$ is associative and $1^{\overrightarrow{A[G]}}$ is its identity element:

$$\begin{aligned}
\big(f*(g*h)\big)(x_0)(x_{123}) &:= \sum_{x_{123}=x_1*^G x_{23}} f(x_0)(x_1) \\
&\qquad *^A \underbrace{\sum_{x_{23}=x_2*^G x_3} g(x_0*^G x_1)(x_2) *^A h(x_0*^G x_1*^G x_2)(x_3)}_{(g*h)(x_0*^G x_1)(x_{23})} \\
&= \sum_{x_{123}=x_1*^G x_2*^G x_3} f(x_0)(x_1) *^A g(x_0*^G x_1)(x_2) *^A h(x_0*^G x_1*^G x_2)(x_3) \\
&= \sum_{x_{123}=x_{12}*^G x_3} \underbrace{\Big(\sum_{x_{12}=x_1*^G x_2} f(x_0)(x_1) *^A g(x_0*^G x_1)(x_2)\Big)}_{(f*g)(x_0)(x_{12})} *^A h(x_0*^G x_{12})(x_3) \\
&= \big((f*g)*h\big)(x_0)(x_{123}) \\
(1*f)(b)(x) &:= \sum_{x=y*^G z} 1^{A[G]}(y) *^A f(b*^G y)(z) = \underbrace{1^{A[G]}(1^G)}_{1^A} *^A f(b*^G 1^G)(x) = f(b)(x) \\
(f*1)(b)(x) &:= \sum_{x=y*^G z} f(b)(y) *^A 1^{A[G]}(z) = f(b)(x) *^A 1^{A[G]}(1^G) = f(b)(x)
\end{aligned}$$

Recall: $1^{\overrightarrow{A[G]}}(b)(x) = 1^{A[G]}(x) = \begin{cases} 1^A & \ldots \quad x = 1^G \\ 0^A & \ldots \quad \text{otherwise.} \end{cases}$

It only remains to be shown that distributivity holds:

$$\begin{aligned}
\big(f*(g+h)\big)(b)(x) &:= \sum_{x=y*^G z} f(b)(y) *^A \big(g(b*^G y)(z) +^A h(b*^G y)(z)\big) \\
&= \big((f*g)+(f*h)\big)(b)(x)
\end{aligned}$$

This follows from the distributivity of A and the associativity and commutativity of $+^A$. Analogously, $(f+g)*h = f*h+g*h$. □

Recall the following definitions:

Definition 2.7 A *(semi)ring homomorphism* is a function $\phi : R \to S$ between two (semi)rings R and S that commutes with $+$ and $*$, i.e., $\phi(a \circ^R b) = \phi(a) \circ^S \phi(b)$ for $\circ \in \{+, *\}$ and all $a, b \in R$, where $\phi(0^R) = 0^S$, and where, for a semiring or a ring with identity, $\phi(1^R) = 1^S$. The set $\{r \in R \mid \phi(r) = 0^S\}$ is called the *kernel* of ϕ. A (semi)ring *isomorphism* is a bijective homomorphism.

Let R be a (semi)ring. A nonempty subset S of R is a *sub-(semi)ring* of R if S is a (semi)ring. This is assured if S is closed under the operations $+^R$, $(-^R,)$ and $*^R$. □

Note that $\overrightarrow{A[G]}$ in general is not a monoid semiring. However, the subset

$$\overrightarrow{A[G]}_0 = \{(\cdot \mapsto \alpha) \mid \alpha \in A[G]\} \subseteq \overrightarrow{A[G]}$$

of functions from $\overrightarrow{A[G]}$ that ignore their input is:

Proposition 2.8 *If $A[G]$ is a monoid (semi)ring, then $\overrightarrow{A[G]}_0$ is a sub(semi)ring of $\overrightarrow{A[G]}$ and isomorphic to $A[G]$.*

Proof. We need to show that $\overrightarrow{A[G]}_0$ is closed under application of $+$, $*$, and, for the case that A has an additive inverse, $-$:

$$\begin{aligned}
(\cdot \mapsto \alpha) +^{\overrightarrow{A[G]}} (\cdot \mapsto \beta) &: b \mapsto x \mapsto (\cdot \mapsto \alpha)(b)(x) +^A (\cdot \mapsto \beta)(b)(x) \\
&= b \mapsto x \mapsto \alpha(x) +^A \beta(x) \;=\; \cdot \mapsto \alpha +^{A[G]} \beta \\
-^{\overrightarrow{A[G]}}(\cdot \mapsto \alpha) &: b \mapsto x \mapsto -^A(\cdot \mapsto \alpha)(b)(x) \\
&= b \mapsto x \mapsto -^A\alpha(x) \;=\; \cdot \mapsto -^{A[G]}\alpha \\
(\cdot \mapsto \alpha) *^{\overrightarrow{A[G]}} (\cdot \mapsto \beta) &: b \mapsto x \mapsto \sum_{x=y*^G z} (\cdot \mapsto \alpha)(b)(y) *^A (\cdot \mapsto \beta)(b *^G y)(z) \\
&= b \mapsto x \mapsto \sum_{x=y*^G z} \alpha(y) *^A \beta(z) \;=\; \cdot \mapsto \alpha *^{A[G]} \beta
\end{aligned}$$

Moreover, $0^{\overrightarrow{A[G]}}, 1^{\overrightarrow{A[G]}} \in \overrightarrow{A[G]}_0$. Thus, $\overrightarrow{A[G]}_0$ is closed under the operations of $\overrightarrow{A[G]}$ and the function $(\cdot \mapsto \alpha) \mapsto \alpha$ is an isomorphism from $\overrightarrow{A[G]}_0$ to $A[G]$. □

Proposition 2.4 is a corollary of Theorem 2.6 and Proposition 2.8.

2.4 Mutilating the monoids

There are applications of monoid and avalanche semirings in which it is desirable for some elements to be excluded from the monoid. As shown next, we can do this and retain semirings and their desirable properties.

Let $(G, *)$ be a monoid. A subset $G_0 \subseteq G$ is called *downward-closed* if $g * h \in G_0$ implies $g, h \in G_0$ for all $g, h \in G$. In general, G_0 is not (upward-)closed under $*$ and thus not a monoid (i.e., there are $g, h \in G_0$ such that $g * h \notin G_0$), but the complement $G \backslash G_0$ is closed under $*$ and thus a semiring. Note that

$$g * h, h * g \in G \backslash G_0 \quad \text{for all } g \in G \text{ and all } h \in G \backslash G_0. \tag{2}$$

We say that monoid G has a *zero* if there is an element $0 \in G$ such that $0 * g = g * 0 = 0$ for all $g \in G$. $G \backslash \{0\}$ is downward-closed.

Lemma 2.9 *Let G be a monoid, G_0 a downward-closed subset of G, and A a (semi)ring. Then the following are (semi)ring homomorphisms:*

1. *from $A[G]$:*

$$\phi_{A[G],G_0} : \alpha \mapsto \{x \mapsto \alpha(x) \mid x \in G_0\}$$

with kernel $I_{A[G],G_0} = \{\alpha \in A[G] \mid \alpha(x) = 0^A \text{ whenever } x \in G_0\}$.

2. from $\overrightarrow{A[G]}$*:*

$$\phi_{\overrightarrow{A[G]},G_0} : f \mapsto \{b \mapsto \{x \mapsto f(b)(x) \mid b * x \in G_0\} \mid b \in G_0\}$$

with kernel $I_{\overrightarrow{A[G]},G_0} = \{i \in \overrightarrow{A[G]} \mid i(b)(x) = 0^A \text{ whenever } b * x \in G_0\}$.

The semiring $\phi_{A[G],G_0}(A[G])$ consists of functions $G_0 \mapsto A$ and the semiring $\phi_{\overrightarrow{A[G]},G_0}(\overrightarrow{A[G]})$ consists of functions f with domain G_0 such that $f(b)$ is a function $\{x \in G_0 \mid b * x \in G_0\} \to A$. In other words, these semirings simply restrict the domains of their element functions as well as the domains of variables that are introduced in the definitions of their multiplication operations to G_0. The definition of the operations $+$ and $*$ remains syntactically unchanged compared to $A[G]$ and $\overrightarrow{A[G]}$.

Proof. Lemma 2.9 is straightforward to prove by verifying the axioms of (semi)ring homomorphisms. We do not do this here, but prove the case of rings using additional machinery that provides further insight into their structure.

Definition 2.10 A *sub-ring* I of ring R is called a *left* respectively *right ideal* of R if $(r * i) \in I$ respectively $(i * r) \in I$ for all $r \in R$ and all $i \in I$, and an *ideal* if I is both a left and a right ideal. A *quotient ring* R/I is (a ring isomorphic to) the image of a ring homomorphism from R whose kernel is I. □

To verify that I is an (left/right) ideal, we only need to check that I is closed under $+$ and satisfies $r * i \in I$ and/or $i * r \in I$ for and $r \in R, i \in I$. Observe the analogy among the definition of two-sided ideals and the characterization of the complements of downward-closed subsets of monoids given above, as (2). The sets $I_{A[G],G_0}$ and $I_{\overrightarrow{A[G]},G_0}$ defined above (already proclaimed to be kernels of the two ring homomorphisms) are ideals:

Lemma 2.11 *Let G be a monoid, G_0 a downward-closed subset of G, and A a ring with identity. Then (1) $I_{A[G],G_0}$ is an ideal of $A[G]$ and (2) $I_{\overrightarrow{A[G]},G_0}$ is an ideal of $\overrightarrow{A[G]}$.*

Proof. (1) It is easy to see that $I_{A[G],G_0}$ is closed under $+^{A[G]}$ (which is just elementwise addition). Consider an arbitrary $x \in G$, an arbitrary $r \in A[G]$, and an arbitrary $i \in I$. By definition,

$$(r * i)(x) = \sum_{x=y*^G z} r(y) *^A i(z) \qquad \text{and} \qquad (i * r)(x) = \sum_{x=y*^G z} i(y) *^A r(z).$$

Assume that $x = y * z \in G_0$, because otherwise $(r * i)(x)$ and $(i * r)(x)$ are unconstrained and we are done. Then $y, z \in G_0$ because of downward-closure. But then $i(y) = i(z) = 0^A$, and as a consequence, $(r * i)(x) = 0^A$. Thus, $(r * i), (i * r) \in I$, so I is an ideal of ring $A[G]$.

(2) $I_{\overrightarrow{A[G]},G_0}$ is closed under $+^{\overrightarrow{A[G]}}$, which again is just elementwise addition. To show that $r * i, i * r \in I$ for all $r \in \overrightarrow{A[G]}$ and $i \in I$, recall that

$$(r * i)(b)(x) = \sum_{x=y*z} r(b)(y) *^A i(b * y)(z), \qquad (i * r)(b)(x) = \sum_{x=y*z} i(b)(y) *^A r(b * y)(z)$$

The interesting case is when $b * x \in G_0$ (here $(r * i)(b)(x)$ and $(i * r)(b)(x)$ must be 0^A), because otherwise elements of I are unconstrained. But if $b * x = b * y * z \in G_0$, then $b, b * y, z \in G_0$, and thus $i(b)(y) = i(b * y)(z) = 0^A$. □

The following converse of the first isomorphism theorem for rings now assures us that the images of the ring homomorphisms of Lemma 2.9 are just the quotient rings $A[G]/I_{A[G],G_0}$ and $\overrightarrow{A[G]}/I_{\overrightarrow{A[G]},G_0}$, and the ring homomorphisms are the natural projections:

Lemma 2.12 (cf. Theorem 7(2) in Chapter 7 of [10]) *Let R be a commutative ring and I be an ideal of R. Then there is a surjective ring homomorphism from R to the quotient ring R/I, the so-called natural projection of R onto R/I and I is the kernel of that homomorphism.*

This concludes the proof of Lemma 2.9. □

Subsequently, when it is understood what $*^G$ is, $A[G]/I_{A[G],G_0}$ and $\overrightarrow{A[G]}/I_{\overrightarrow{A[G]},G_0}$ will be referred to as $A[G_0]$ and $\overrightarrow{A[G_0]}$, respectively.

The main reason for mutilating monoids in this paper is to exclude the zero. The application for this will be obvious in Section 3.

The type of functions $f \in \overrightarrow{A[G_0]}$ is perhaps a little awkward to handle since the type of $f(b)$ depends on b. We can easily deal with this by extending the type of $f(b)$ uniformly to $A[G_0]$ for all b by requiring $f(b)(x) = 0$ for all $b * x \notin G_0$ and defining multiplication as

$$f * g : b \mapsto x \mapsto \sum_{x = y *^G z,\ b * y \in G_0} f(b)(y) *^A g(b *^G y)(z).$$

One important application of the mutilation construction in the context of databases is to exclude the element 0^G from G, since monoid and avalance semirings would otherwise allow to store information not associated with tuples (admitting the existence of many different "empty" databases). Depending on the application, this may either be slightly undesirable or useful and worthy of further study.[3]

2.5 Monoid rings as modules

Definition 2.13 Let A be a ring. A *(left) A-module* is a set M with a binary operation $+$ on M under which M is a commutative group, and an action $A \times M \to M$, denoted am (the *scalar product*), for all $a \in A$, $m \in M$, which satisfies for all $a, b \in A$ (the *scalars*), $m, n \in M$: $(a +^A b)m = am +^M bm$, $(a *^A b)m = a(bm)$, and $a(m +^M n) = am +^M an$. If A is a ring with identity, we also require $1^A m = m$ for all $m \in M$.

An A-module M is said to be *free* on the subset G of M if for every nonzero element m of M there exist *unique* nonzero elements $a_1, \ldots, a_n \in A$, $g_1, \ldots, g_n \in G$, for some $n \in \mathbb{N}$, such that $m = a_1 g_1 + \cdots + a_n g_n$. In this case, we say that G is a *basis* or *set of free generators* for M.

Let A be a commutative ring. An A-algebra is an A-module M with a binary operation $[\cdot, \cdot] : M \times M \to M$ that is *bilinear*, i.e., which satisfies $[ax + by, z] = a[x, z] + b[y, z]$ and $[z, ax + by] = a[z, x] + b[z, y]$ for all scalars $a, b \in A$ and all elements $x, y, z \in M$. An algebra is called associative if $[\cdot, \cdot]$ is associative. □

[3] It allows to record provenance information for pairs of tuples that fail a join condition.

Proposition 2.14 *(1) Bilinearity implies distributivity, so every associative algebra is a ring. (2) A distributive operation $*$ such that $(ax) * y = a(x * y) = x * (ay)$ for $a \in A$ and $x, y \in M$ is bilinear.*

Proof. (1) Have $a = b = 1^A$. (2) For $a, b \in A$ and $x, y, z \in M$, $(ax+by)*z = (ax)*z+(by)*z = a(x*z)+b(y*z)$ and $z*(ax+by) = z*ax+z*by = a(z*x)+b(z*y)$; in both cases, the first step applies distributivity and the second the precondition of statement (2) of the proposition. □

Note that relational algebra (with union and natural join as addition and multiplication operations) is not an algebra. (Union cannot be applied to every pair of relations.)

A key application of monoid rings is their use in formalizing polynomials. In that case, it is natural to write the functions α of $A[G]$ as sums $\sum_g \alpha(g)g$. This provides us with an alternative viewpoint of monoid rings as A-modules. These are a vehicle for showing that there is essentially only one way[4] of defining a distributive multiplication operation: the convolution product $*^{A[G]}$ of monoid rings.

The following is known (e.g., [23], though in part implicit there), but a proof is given because its notation will be used later:

Proposition 2.15 *Let $A[G]$ be a monoid ring. Then*

1. *$\big(A[G], +^{A[G]}, (a\alpha) \mapsto x \mapsto (a *^A \alpha(x))\big)$ is an A-module that is free on basis G.*
2. *If A is commutative[5], then $*^{A[G]}$ is bilinear, so $A[G]$ is an associative A-algebra.*

Proof. (1) The fact that $\big(A[G], +^{A[G]}, (a\alpha) \mapsto x \mapsto (a *^A \alpha(x))\big)$ is an A-module follows immediately from the fact that A is a ring and from the definition of $+^M$ as $+^{A[G]}$. For instance, $(a +^A b)\alpha = x \mapsto (a +^A b) *^A \alpha(x) = x \mapsto (a *^A \alpha(x)) +^A (b *^A \alpha(x)) = a\alpha +^M b\alpha$ by the distributivity of A.

We write elements

$$x \mapsto \begin{cases} 1 & \dots \quad x = g \\ 0 & \dots \quad \text{otherwise} \end{cases}$$

of $A[G]$ as χ_g and slightly abuse notation to write G for the set $\{\chi_g \mid g \in G\} \subseteq A[G]$. Each $\alpha \in A[G]$ can be written as a sum $\sum_{g \in G} \alpha(g)\chi_g$. This sum is finite because $A[G]$ guarantees that $\alpha(g)$ is nonzero for only a finite number of $g \in G$. Since $+^{A[G]}$ is associative and commutative, this sum is *unique*. Thus G *generates* $A[G]$ and $A[G]$ is an A-module that is *free* on G; G is the *basis* of $A[G]$.

(2) Because of the associativity and commutativity of $*^A$, $(a\alpha) *^{A[G]} \beta = a(\alpha *^{A[G]} \beta) = \alpha *^{A[G]} (a\beta)$. By Proposition 2.14 (2), $*^{A[G]}$ is bilinear. □

A quotient ring $A[G_0]$ remains an A-module (an A-algebra if commutative) and is free on basis G_0.

Viewing $A[G]$ as an A-module means to ignore its operation $*$. This is particularly clear for $\mathbb{Z}[G]$. Here the action $a\alpha$, for $a \in \mathbb{Z}$ and $\alpha \in \mathbb{Z}[G]$, can be expressed as a finite sum of elements

[4] For this we need the additive inverse, so A must be a ring.

[5] In this case we have called $A[G]$ is a monoid algebra in Definition 2.3; and indeed, it is an algebra.

of G (with repetitions), i.e., $ag = \underbrace{g + \cdots + g}_{a \text{ times}}$ for $a \geq 0$ and $(-a)g = \underbrace{-g \cdots - g}_{a \text{ times}}$ for $(-a) < 0$. Thus we do not need the action $a\alpha$ or $*^{\mathbb{Z}[G]}$ to implement it, and the additive group of $\mathbb{Z}[G]$ by itself is a $\mathbb{Z}$-module.

We say that an operation $\circ$ over $\mathbb{Z}[G]$ is *conservative over* $*^G$ if $\chi_g \circ \chi_h = \chi_g *^G \chi_h$ for all $g, h \in G$. If we accept the definition of the module $\mathbb{Z}[G]$ as natural, then the definition of $*^{\mathbb{Z}[G]}$ (which may feel less natural at first) is uniquely determined by distributivity, if we want $*^{\mathbb{Z}[G]}$ to be conservative over $*^G$.

Proposition 2.16 *The convolution product $*^{\mathbb{Z}[G]}$ is the unique product operation that extends the additive group $(\mathbb{Z}[G], +^{\mathbb{Z}[G]})$ to a ring and is conservative over $*^G$.*

Proof. We use the sum notation from the proof of Proposition 2.15. Let $\circ$ be an arbitrary multiplication operation on $\mathbb{Z}[G]$ that is distributive with $+$.

$$\alpha \circ \beta := \Big(\sum_g \alpha(g)\chi_g\Big) \circ \Big(\sum_h \beta(h)\chi_h\Big) = \sum_{g,h} \big(\alpha(g) *^A \beta(h)\big)\big(\chi_g \circ \chi_h\big)$$

This follows from the distributivity of $\circ$ and the fact that $\alpha(g)\chi_g$ and $\beta(h)\chi_h$ are actually sums. Since $\circ$ is conservative over $*^G$, $\chi_g \circ \chi_h = \chi_g *^G \chi_h = \chi_{g *^G h}$, and

$$(\alpha \circ \beta)(x) = \sum_{g,h} (\alpha(g) * \beta(h))\chi_{g*h}(x) = \sum_{g,h:g*h=x} \alpha(g) * \beta(h),$$

as in Definition 3.1, so $\circ$ is $*^{\mathbb{Z}[G]}$. □

It is appealing that our natural choice of $+$ completely determines $*$ in any ring that extends the additive group $(\mathbb{Z}[G], +^{\mathbb{Z}[G]})$, and that the structure of a monoid algebra arises necessarily and naturally.

3 Algebraic databases and queries

The goal of this section is to construct an analogon of *multiset* relational algebra – a starting point for building aggregate queries – which has a full *additive inverse*, and so will allow us to compute delta queries in a clean and compositional way. We extend this formalism to a ring structure that allows for sideways binding passing, which is key to building an expressive query language that admits condition checking within an algebraic framework (with key operations $+$ and $*$, but no selection operation, which would have to be higher-order).

3.1 Algebraic stores

Relational algebra, with union considered its addition, is not an algebra because union is not total – two relations can only be combined by union if they have the same schema. Moreover, even union in multiset relational algebra does not have an additive inverse. We now fix these two problems.

We construct a (semi)ring of *generalized multiset relations (gmrs)* – collections of tuples with multiplicities from a (semi)ring and possibly *differing* schemas. The operations $+$ and

$*$ are generalizations of multiset union and natural join, respectively, to *total* functions (i.e., applicable to any pair of gmrs). The schema polymorphism of tuples in gmrs serves the purpose of accommodating such total *operator* definitions.

Let Σ be a vocabulary of column names and Adom an *active domain* of data values. A *record* is a tuple with a schema of its own, and formalized as a *partial* function $\Sigma \to \text{Adom}$. The set of all records is denoted by $\mathbb{T}$. We write $\{\vec{t}\}$ to construct a classical singleton relation (without multiplicities) from record $\vec{t}$. The schema $\text{sch}(\{\vec{t}\})$ of this singleton is $\text{dom}(\vec{t})$. The set of all singletons is denoted by Sng.

Let $\text{Sng}_\emptyset = \text{Sng} \cup \{\emptyset\}$ be the set of singletons plus the empty relation. Then $\text{Sng}_\emptyset$ with the natural join $\bowtie$ forms a commutative monoid with 1-element $\{\langle\rangle\}$ and zero $\emptyset$. In particular, $\text{Sng}_\emptyset$ is closed under $\bowtie$.

Definition 3.1 Fix a semiring (or ring with identity) A. A *generalized multiset relation (gmr)* is a function $R : \mathbb{T} \to A$ such that $R(\vec{t}) \neq 0$ for at most a finite number of tuples $\vec{t}$. Such a function indicates the multiplicity with which each tuple of $\mathbb{T}$ occurs in the gmr. If A is a ring, tuples can have negative multiplicities. The set of all such functions is denoted by $A[\mathbb{T}]$.

The operations $+$ and $*$ of $A[\mathbb{T}]$ are defined as follows. For $R, S \in A[\mathbb{T}]$,

$$\begin{aligned} R + S &: \vec{x} \mapsto \big(R(\vec{x}) + S(\vec{x})\big) \\ R * S &: \vec{x} \mapsto \sum_{\{\vec{x}\}=\{\vec{y}\}\bowtie\{\vec{z}\}} R(\vec{y}) * S(\vec{z}) \\ 1 &: \vec{x} \mapsto \begin{cases} 1 & \ldots \quad \vec{x} = \langle\rangle \\ 0 & \ldots \quad \vec{x} \neq \langle\rangle \end{cases} \\ 0 &: \vec{x} \mapsto 0 \end{aligned}$$

and $(-R) : \vec{x} \mapsto (-R(\vec{x}))$ if A is a ring. □

Example 3.2 Consider the three gmrs of $A[\mathbb{T}]$

R	A	B	
	a_1		$\mapsto r_1$
	a_2	b	$\mapsto r_2$

S	C	
	c	$\mapsto s$

T	B	C	
		c	$\mapsto t_1$
	b	c	$\mapsto t_2$

over column name vocabulary $\Sigma = \{A, B, C\}$ and value domain $\text{Adom} = \{a_1, a_2, b, c\}$. Only entries with nonzero multiplicity are shown. For example, in multiset relation R, two tuples of different schema have a multiplicity other than 0. These two tuples can be specified as partial functions $\Sigma \to \text{Adom}$: $\{A \mapsto a_1\}$ and $\{A \mapsto a_2; B \mapsto b\}$. The values r_1, r_2, s, t_1, t_2 are from A.

Then $S + T$ and $R * (S + T)$ are as follows:

$S+T$	B	C	
		c	$\mapsto s + t_1$
	b	c	$\mapsto t_2$

$R*(S+T)$	A	B	C	
	a_1		c	$\mapsto r_1 * (s + t_1)$
	a_1	b	c	$\mapsto r_1 * t_2$
	a_2	b	c	$\mapsto r_2 * (s + t_1) + r_2 * t_2$

The missing values should not be taken as SQL null values, and $*$ is not an outer join. □

On classical multiset relations (where all multiplicities are ≥ 0 and all tuples with multiplicity > 0 have the same schema), $*$ is exactly the usual multiset natural join operation. Definition 3.1 is isomorphic to the definition of a mutilated monoid algebra $A[\mathrm{Sng}]$ (the mutilation is the removal of the zero from the monoid $\mathrm{Sng}_\emptyset$):

Proposition 3.3 *If A is a (commutative) semiring respectively ring with identity, then so is $A[\mathbb{T}]$.*

Proof. Mutilate the monoid $\mathrm{Sng}_\emptyset$ by removing $\emptyset$. By Lemma 2.9, $A[\mathrm{Sng}]$ is a commutative ring with identity ($\mathrm{Sng} = \mathrm{Sng}_\emptyset \backslash \{\emptyset\}$).

The map $\theta : \alpha \mapsto \left(\vec{t} \mapsto \alpha(\{\vec{t}\})\right)$ from $A[\mathrm{Sng}]$ to $A[\mathbb{T}]$ is a ring isomorphism: The two rings are really the same, with the minor notational difference that the domain of the latter is tuples, while the domain of the former is the same tuples elevated to singleton sets. □

By Proposition 2.15 and the remarks that follow it, $A[\mathbb{T}]$ is an A-module. The proof of Proposition 2.16 also immediately applies to $\mathbb{Z}[\mathbb{T}]$, so $\mathbb{Z}[\mathbb{T}]$ is the smallest generalization of multiset relational algebra to a ring.

3.2 Parametrized gmrs and binding passing

As for monoid (semi)rings, the monoid $\mathrm{Sng}_\emptyset$ allows us to construct avalanche (semi)rings over databases. Below, we continue to work with tuples rather than singletons, like in our choice of $A[\mathbb{T}]$ rather than $A[\mathrm{Sng}_\emptyset]$ above. We construct a structure analogous to $A[\overrightarrow{\mathrm{Sng}_\emptyset}]$ that excludes $\emptyset$ and maps tuples to $A[\mathbb{T}]$.

A *parametrized gmr (pgmr)* is a function $f : \mathbb{T} \to A[\mathbb{T}]$ such that $f(\vec{b})(\vec{x}) = 0^A$ for any inconsistent $\vec{b}, \vec{x} \in \mathbb{T}$ (i.e., $\{\vec{b}\} \bowtie \{\vec{x}\} = \emptyset$). The set of all pgmrs is denoted by $\overrightarrow{A[\mathbb{T}]}$. The operations on $\overrightarrow{A[\mathbb{T}]}$ are

$$\begin{aligned}
f + g &: \vec{b} \mapsto \left(f(\vec{b}) +^{A[\mathbb{T}]} g(\vec{b})\right) \\
f * g &: \vec{b} \mapsto \vec{x} \mapsto \sum_{\{\vec{x}\}=\{\vec{y}\}\bowtie\{\vec{z}\},\ \{\vec{b}\}\bowtie\{\vec{y}\}\neq\emptyset} f(\vec{b})(\vec{y}) *^A g(\vec{b}\vec{y})(\vec{z}) \\
0 &: \cdot \mapsto 0^{A[\mathbb{T}]} \\
1 &: \cdot \mapsto 1^{A[\mathbb{T}]}
\end{aligned}$$

Proposition 3.4 *If A is a semiring respectively ring with identity, then so is $\overrightarrow{A[\mathbb{T}]}$.*

Example 3.5 Let $R \in A[\mathbb{T}]$, $f \in \overrightarrow{A[\mathbb{T}]}$ be the function $\cdot \mapsto R$ that ignores its parameters and returns R, and let $g \in \overrightarrow{A[\mathbb{T}]}$ be the function

$$\vec{b} \mapsto \vec{x} \mapsto \begin{cases} 1^A & \ldots \quad \vec{x} = \langle\rangle,\ A, B \in \mathrm{dom}(\vec{b}),\ \vec{b}(A) < \vec{b}(B) \\ 0^A & \ldots \quad \text{otherwise} \end{cases}$$

By definition of $*^{\overrightarrow{A[\mathbb{T}]}}$, and since $g(\vec{y})(\vec{z}) = 0$ unless $\vec{z} = \langle\rangle$,

$$\begin{aligned}\big(f *^{\overrightarrow{A[\mathbb{T}]}} g\big)(\langle\rangle) \quad &: \quad \vec{x} \mapsto \sum_{\{\vec{x}\}=\{\vec{y}\}\bowtie\{\vec{z}\}} f(\langle\rangle)(\vec{y}) *^A g(\langle\rangle\vec{y})(\vec{z}) \\ &= \quad \vec{x} \mapsto \sum_{\{\vec{x}\}=\{\vec{y}\}\bowtie\{\langle\rangle\}} R(\vec{y}) *^A g(\vec{y})(\langle\rangle) \\ &= \quad \vec{x} \mapsto R(\vec{x}) *^A g(\vec{x})(\langle\rangle).\end{aligned}$$

This "selects" those tuples of R that have A and B in their domain and satisfy the condition $A < B$. The multiplicity of tuples $\vec{x}$ selected is that of R, $R(\vec{x})$. □

4 Query Language

This section introduces the query language AGCA (which stands for *AGgregation CAlculus*). The overall query language inherits the key properties of polynomial rings in that it has an additive inverse, a normal form of polynomial expressions, and aggregates admit a form of factorization. These properties will be the basis of delta processing and incremental query evaluation in subsequent sections.

Syntax. AGCA expressions are built from constants, variables, relational atoms, aggregate sums (Sum), conditions, and variable assignments (:=) using $+$, $-$, and $*$. The abstract syntax can be given by the EBNF

$$q \;::=\; q * q \mid q + q \mid -q \mid \mathrm{Sum}(q) \mid c \mid x \mid R(\vec{x}) \mid q\;\theta\;0 \mid x := q$$

Here x denotes variables, $\vec{x}$ tuples of variables, R relation names, c constants from A, and θ denotes comparison operations ($=$, $\neq$, $>$, $\geq$, $<$, and $\leq$).

We will also write $q\,\theta\,q'$ for $(q - q')\,\theta\,0$. Note that $x := q$ is a special condition essentially equivalent to $x = q$, with one catch. Remember that in relational calculus, both variables x and y are safe in $\phi \wedge x = y$ if at least one of them is safe in ϕ (the other variable can be *assigned* the value of the safe variable from ϕ). In order to simplify the following semantics definition, we make the distinction between the case where only one of the variables is safe from the left (:=) and the case where both are safe (=) clear by syntax.

Semantics. The formal semantics of AGCA is given by an evaluation function $[\![\cdot]\!]$ that, given a query q and a database $\mathcal{A}$, evaluates to an element $[\![q]\!](\mathcal{A})$ of $\overrightarrow{A[\mathbb{T}]}$:

$$\begin{aligned}[\![q_1 + q_2]\!](\mathcal{A}) &:= [\![q_1]\!](\mathcal{A}) +^{\overrightarrow{A[\mathbb{T}]}} [\![q_2]\!](\mathcal{A}) \\ [\![-q]\!](\mathcal{A}) &:= -^{\overrightarrow{A[\mathbb{T}]}} [\![q]\!](\mathcal{A}) \\ [\![q_1 * q_2]\!](\mathcal{A}) &:= [\![q_1]\!](\mathcal{A}) *^{\overrightarrow{A[\mathbb{T}]}} [\![q_2]\!](\mathcal{A}) \\ [\![\mathrm{Sum}\,q]\!](\mathcal{A}) &:= \vec{b} \mapsto \vec{x} \mapsto \sum_{\vec{x}*\vec{y}=\vec{y}} [\![q]\!](\mathcal{A})(\vec{b})(\vec{y}) \\ [\![c]\!](\,\cdot\,) &:= \cdot \mapsto x \mapsto \begin{cases} c & \ldots \quad x = \langle\rangle \\ 0 & \ldots \quad \text{otherwise} \end{cases} \\ [\![q\,\theta\,0]\!](\mathcal{A}) &:= \vec{b} \mapsto \vec{x} \mapsto \begin{cases} 1 & \ldots \quad [\![q]\!](\mathcal{A})(\vec{b})(\langle\rangle)\,\theta\,0,\ \vec{x} = \langle\rangle \\ 0 & \ldots \quad \text{otherwise} \end{cases}\end{aligned}$$

$$[\![y := t]\!](\mathcal{A}) \;:=\; \vec{b} \mapsto \vec{x} \mapsto \begin{cases} 1 \ldots & \vec{x} = \{y \mapsto [\![t]\!](\mathcal{A})(\vec{b})(\langle\rangle)\} \\ 0 \ldots & \text{otherwise} \end{cases}$$

$$[\![y]\!](\,\cdot\,) \;:=\; \vec{b} \mapsto \vec{x} \mapsto \begin{cases} \textbf{fail} & \ldots \quad y \notin \mathrm{dom}(\vec{b}) \\ \vec{b}(y) & \ldots \quad \text{otherwise, if } \vec{x} = \langle\rangle \\ 0 & \ldots \quad \text{otherwise} \end{cases}$$

$$[\![R(x_1, \ldots, x_k)]\!](\mathcal{A}) \;:=\; \vec{b} \mapsto \vec{x} \mapsto \begin{cases} R^{\mathcal{A}}(\{A_i \mapsto \vec{x}(x_i) \mid A_i \in \mathrm{sch}(R)\}) & \ldots \quad \{\vec{b}\} \bowtie \{\vec{x}\} \neq \emptyset, \\ & \qquad |\mathrm{dom}(\vec{x})| = |\mathrm{sch}(R)| \\ 0 & \ldots \quad \text{otherwise.} \end{cases}$$

Recall that records are functions: $\{\vec{b}\} \bowtie \{\vec{x}\} \neq \emptyset$ asserts that the records $\vec{b}$ and $\vec{x}$ are consistent, that is, for all $y \in \mathrm{dom}(\vec{b}) \cap \mathrm{dom}(\vec{x})$, $\vec{b}(y) = \vec{x}(y)$.

The definition of $[\![R(x_1, \ldots, x_k)]\!]$ supports column renaming, which makes it a little lengthy.

Example 4.1 Let R be a database relation

$R^{\mathcal{A}}$	a	b		
	a_1	b_1	$\mapsto$	r_1
	a_2	b_2	$\mapsto$	r_2

Then

$[\![R(x,y)]\!](\mathcal{A})(\{y \mapsto b_1\})$	x	y		
	a_1	b_1	$\mapsto$	r_1

The query renames the columns (a, b) to (x, y) and selects on y since it is a bound variable. □

Example 4.2 Let

$[\![R(x,y)]\!](\mathcal{A})(\langle\rangle)$	x	y		
	1		$\mapsto$	a_1
		1	$\mapsto$	a_2
	1	1	$\mapsto$	a_3
	1	2	$\mapsto$	a_4

and $a_1, \ldots, a_4 \in \mathbb{Z}$. Then

$[\![R(x,y) * (x < y)]\!](\mathcal{A})(\langle\rangle)$	x	y		
	1	2	$\mapsto$	a_4

$[\![R(x,y) * (x = y)]\!](\mathcal{A})(\langle\rangle)$	x	y		
	1	1	$\mapsto$	$a_1 + a_2 + a_3$

For example, $[\![R(x,y) * (x = y)]\!](\mathcal{A})(\langle\rangle)(\{x \mapsto 1; y \mapsto 1\}) =$

$$\begin{aligned} & \sum_{\{x \mapsto 1; y \mapsto 1\} = \{\vec{y}\} \bowtie \{\vec{z}\}} [\![R(x,y)]\!](\mathcal{A})(\langle\rangle)(\vec{y}) *^A [\![x = y]\!](\mathcal{A})(\vec{y})(\vec{z}) \\ = \;& [\![R(x,y)]\!](\mathcal{A})(\langle\rangle)(\{x \mapsto 1\}) *^A [\![x = y]\!](\mathcal{A})(\{x \mapsto 1\})(\{x \mapsto 1; y \mapsto 1\}) \\ + \;& [\![R(x,y)]\!](\mathcal{A})(\langle\rangle)(\{y \mapsto 1\}) *^A [\![x = y]\!](\mathcal{A})(\{y \mapsto 1\})(\{x \mapsto 1; y \mapsto 1\}) \\ + \;& [\![R(x,y)]\!](\mathcal{A})(\langle\rangle)(\{x \mapsto 1; y \mapsto 1\}) *^A [\![x = y]\!](\mathcal{A})(\{x \mapsto 1; y \mapsto 1\})(\{x \mapsto 1; y \mapsto 1\}) \\ = \;& a_1 + a_2 + a_3. \end{aligned}$$

□

Example 4.3 For gmr R of Example 4.1,

$$\begin{aligned}
[\![\mathrm{Sum}(R(x,y) * 3 * x)]\!](\mathcal{A})(\langle\rangle)(\langle\rangle) &= \sum_{\vec{x},\vec{y},\vec{z}} [\![R(x,y)]\!](\mathcal{A})(\langle\rangle)(\vec{x}) * [\![3]\!](\mathcal{A})(\vec{x})(\vec{y}) * [\![x]\!](\mathcal{A})(\vec{x} * \vec{y})(\vec{z}) \\
&= r_1 * 3 * [\![x]\!](\mathcal{A})(\{x \mapsto a_1, y \mapsto b_1\})(\langle\rangle) \\
&+ r_2 * 3 * [\![x]\!](\mathcal{A})(\{x \mapsto a_2, y \mapsto b_2\})(\langle\rangle) \\
&= r_1 * 3 * a_1 + r_2 * 3 * a_2.
\end{aligned}$$

□

Example 4.4 AGCA is powerful enough to construct gmrs from scratch:

$$[\![(x := x_1) * (y := y_1) * z + (x := x_2) * (-3)]\!](\cdot)(\{x_1 \mapsto a_1; y_1 \mapsto b_1; x_2 \mapsto a_2; z \mapsto 2\})$$

defines the gmr

x	y		
a_1	b_1	$\mapsto$	2
a_2		$\mapsto$	-3

□

Note that $[\![\cdot]\!]$ indeed maps to elements of $\overrightarrow{A[\mathbb{T}]}$; in particular, the image elements are functions $\mathbb{T} \to \mathbb{T} \to A$ whose images are in $A[\mathbb{T}]$, i.e., are nonzero on at most a finite number of inputs.

The evaluation of variables x using $[\![x]\!]$ *fails* if they are not bound at evaluation time. We will consider a query in which this may happen illegal, and will subsequently exclude such queries from AGCA. A definition of *range-restriction* completely analogous to that for relational calculus (cf. [1]), with $\wedge$ and $\vee$ replaced by $*$ and $+$, respectively, allows to statically verify that such runtime failures do not happen.

5 Properties of AGCA

In this section, we first discuss the relationship between AGCA and SQL, relational algebra, and the relational calculus. Then we show how AGCA queries can be turned into polynomials, how monomials admit factorization, and how unnecessary variables can be eliminated from queries.

AGCA on classical and multiset relations. A gmr $R \in \mathbb{Z}[\mathbb{T}]$ is a multiset relation if all of its tuples $\vec{t}$ with nonzero multiplicity have consistent schema $\mathrm{dom}(\vec{t})$ (which is called the schema of R, $\mathrm{sch}(R)$) and no tuple has negative multiplicity.

Let $R, [\![\alpha]\!]_F(\mathcal{A}, \vec{b}), [\![\beta]\!]_F(\mathcal{A}, \vec{b}) \in \mathbb{Z}[\mathbb{T}]$ be multiset relations. Then

$$\begin{aligned}
[\![R(x_1, \ldots, x_{\mathrm{sch}(R)})]\!](\mathcal{A})(\vec{b}) &:= \sigma_{\vec{x} \mapsto (\{\vec{b}\} \bowtie \{\vec{x}\} \neq \emptyset)}(\rho_{\mathrm{sch}(R) \to (x_1,\ldots,x_{\mathrm{sch}(R)})} R^{\mathcal{A}}) \\
[\![\alpha * \beta]\!](\mathcal{A})(\vec{b}) &:= [\![\alpha]\!](\mathcal{A})(\vec{b}) \bowtie [\![\beta]\!](\mathcal{A})(\vec{b}) \\
[\![\alpha + \beta]\!](\mathcal{A})(\vec{b}) &:= [\![\alpha]\!](\mathcal{A})(\vec{b}) \cup [\![\beta]\!](\mathcal{A})(\vec{b}) \\
&\quad (\alpha \text{ and } \beta \text{ have consistent schema}) \\
[\![\phi * (t\,\theta\,0)]\!](\mathcal{A})(\vec{b}) &:= \sigma_{\vec{x} \mapsto ([\![t]\!](\mathcal{A})(\vec{b}\vec{x})(\langle\rangle)\,\theta\,0)}([\![\phi]\!](\mathcal{A})(\vec{b}))
\end{aligned}$$

are multiset relations. Here ρ, σ, $\bowtie$, $\cup$ are the tuple renaming, selection, natural join, and multiset union operators of multiset relational algebra. Selection is of the form $\sigma_{\mathbb{T}\to\text{Boolean}}$, i.e., the subscript is not a condition but a function that evaluates the condition for a given tuple. (This allows for a clean way to specify nested aggregates.)

Remark 5.1 Throughout this paper, "$-$" does not refer to the difference operation of relational algebra, but to the additive inverse in $A[\mathbb{T}]$: for instance, $\emptyset + (-R) = -R$ in $A[\mathbb{T}]$, while $\emptyset - R$ in relational algebra results in $\emptyset$. It is more appropriate to think of a gmr $-R$ as a deletion, where deleting "too much" results in a database with *negative tuples*.

From SQL to the calculus. A SQL aggregate query

`SELECT` $\vec{b}$, `SUM`(t) `FROM` $R_1\ r_{11}$, $R_1\ r_{12}$, …, $R_2\ r_{21}$, … `WHERE` ϕ `GROUP BY` $\vec{b}$

is expressed in AGCA as

$$\text{Sum}(R_1(\vec{x}_{11}) * R_1(\vec{x}_{12}) * \cdots * R_2(\vec{x}_{21}) * \cdots * \phi * t)$$

with *bound variables* $\vec{b}$. While $\text{Sum}(\cdot)$ computes exactly one number, we can think of an SQL aggregate query with group by clause as a function $\vec{b} \mapsto [\![\text{Sum}(\cdot)]\!](\mathcal{A})(\vec{b})$ that maps each group to its aggregate value.

Example 5.2 Relation C(<u>cid</u>, nation) stores the ids and nationalities of customers. The SQL query

```
SELECT   C1.cid, SUM(1)
FROM     C C1, C C2
WHERE    C1.nation = C2.nation
GROUP BY C1.cid;
```

asks, for each cid, for the number of customers of the same nation (including the customer identified by cid). This query translates to AGCA as

$$\text{Sum}\big(C(c,n) * C(c',n') * (n = n') * 1\big)$$

with bound variable c. Let v be a company id. Since C is a multiset relation,

$$\begin{aligned}
[\![C(c,n)]\!](\mathcal{A})(\{c \mapsto v\}) &= \sigma_{c=v}(\rho_{(\text{cid},\text{nation})\to(c,n)} C^{\mathcal{A}}) \\
[\![C(c',n')]\!](\mathcal{A})(\{c \mapsto v\}) &= \rho_{(\text{cid},\text{nation})\to(c',n')} C^{\mathcal{A}} \\
[\![1]\!](\cdot)(\cdot) &= \{\!\{\langle\rangle\}\!\} \\
R &:= [\![C(c,n) * C(c',n') * (n = n') * 1]\!](\mathcal{A})(\{c \mapsto v\}) \\
&= \sigma_{n=n'}\big(\sigma_{c=v}(\rho_{(\text{cid},\text{nation})\to(c,n)} C^{\mathcal{A}}) \times \rho_{(\text{cid},\text{nation})\to(c',n')} C^{\mathcal{A}}\big) \times \{\!\{\langle\rangle\}\!\}
\end{aligned}$$

are multiset relations too, and R has schema $\{c, n, c', n'\}$. We have

$$[\![\text{Sum}(C(c,n) * C(c',n') * (n = n') * 1)]\!](\mathcal{A})(\{c \mapsto v\}) = \sum_{\vec{x}} R(\vec{x})$$

which is equivalent to
`SELECT SUM(1) from C C1, C C2 WHERE C1.nation = C2.nation AND C1.cid =` v. □

6 Updates and Delta Queries

Our goal is, given an AGCA expression α, to construct a query $\Delta\alpha$ that expresses the change made to the database by the insertion respectively deletion of a single tuple. We write u to denote an update. More explicitly, we write $\pm R(\vec{t})$ to denote the insertion or deletion of a tuple $\vec{t}$ into/from relation R of the database. We write $\Delta_u\alpha$ or $\Delta_{\pm R(\vec{t})}\alpha$ to denote the delta-query for such an update.

$$\begin{aligned}
\Delta_u(\alpha+\beta) &:= (\Delta_u\alpha)+\Delta_u\beta \\
\Delta_u(\alpha * \beta) &:= ((\Delta_u\alpha)*\beta)+(\alpha*\Delta_u\beta)+((\Delta_u\alpha)*\Delta_u\beta) \\
\Delta_u(-\alpha) &:= -\Delta_u\alpha \\
\Delta_u\mathrm{Sum}(\alpha) &:= \mathrm{Sum}(\Delta_u\alpha) \\
\Delta_u(t\,\theta\,0) &:= \big(((t+\Delta_u t)\,\theta\,0)*(t\,\bar\theta\,0)\big)-\big(((t+\Delta_u t)\,\bar\theta\,0)*(t\,\theta\,0)\big) \\
\Delta_u x &:= 0 \\
\Delta_u c &:= 0 \\
\Delta_{\pm R(\vec{t})}\big(R(x_1,\ldots,x_{\mathrm{sch}(R)})\big) &:= \pm \prod_{i=1}^{|\mathrm{sch}(R)|}(x_i := t_i) \\
\Delta_{\pm R(\vec{t})}\big(S(x_1,\ldots,x_{\mathrm{sch}(S)})\big) &:= 0 \qquad (R\neq S)
\end{aligned}$$

where $\bar\theta$ is the complement of the comparison operator θ (i.e., $\neq$ for $=$, $\geq$ for $<$, etc.) and $\mathrm{sch}(R)$ is the schema of R – the set of its column names. We treat an assignment $x := q$ like an equality condition $x = q$.

Proposition 6.1 *Given any database $\mathcal{A}$ and an update $u = \pm R(\vec{t})$,*

$$[\![\alpha]\!](\mathcal{A}+u) = [\![\alpha]\!](\mathcal{A}) + [\![\Delta_u\alpha]\!](\mathcal{A}).$$

Proof. The proof is by a straightforward bottom-up induction on the abstract syntax tree of AGCA expressions.

Case $\alpha\circ\beta$ (for $\circ\in\{+,*\}$): By definition of $[\![\cdot]\!]$, $[\![\alpha\circ\beta]\!](\mathcal{B}) = [\![\alpha]\!](\mathcal{B})\circ[\![\beta]\!](\mathcal{B})$. Let $\mathcal{B}=\mathcal{A}+u$. By the induction hypothesis, $[\![\alpha]\!](\mathcal{B})\circ[\![\beta]\!](\mathcal{B}) = ([\![\alpha]\!](\mathcal{A})+[\![\Delta_u\alpha]\!](\mathcal{A}))\circ([\![\beta]\!](\mathcal{A})+[\![\Delta_u\beta]\!](\mathcal{A})) = [\![(\alpha+\Delta_u\alpha)\circ(\beta+\Delta_u\beta)]\!](\mathcal{A})$. Given our definition of Δ_u, by the associativity and commutativity of $+$ and distributivity of $*$ over $+$, this is $[\![(\alpha\circ\beta)+\Delta_u(\alpha\circ\beta)]\!](\mathcal{A})$.

Case $-\alpha$: Treat as $(-1)*\alpha$, where -1 is a constant, handled below.

Case Sum α:

$$\begin{aligned}
[\![\mathrm{Sum}\ \alpha]\!](\mathcal{A}+u)(\vec{b})(\vec{x}) &= \sum_{\vec{x}*\vec{y}=\vec{y}} [\![\alpha]\!](\mathcal{A}+u)(\vec{b})(\vec{y}) \\
&= \sum_{\vec{x}*\vec{y}=\vec{y}} [\![\alpha]\!](\mathcal{A})(\vec{b})(\vec{y}) + [\![\Delta_u\alpha]\!](\mathcal{A})(\vec{b})(\vec{y}) \\
&= \Big(\sum_{\vec{x}*\vec{y}=\vec{y}} [\![\alpha]\!](\mathcal{A})(\vec{b})(\vec{y})\Big) + \sum_{\vec{x}*\vec{y}=\vec{y}} [\![\Delta_u\alpha]\!](\mathcal{A})(\vec{b})(\vec{y}) \\
&= [\![\mathrm{Sum}\ \alpha]\!](\mathcal{A})(\vec{b})(\vec{x}) + [\![\mathrm{Sum}\ \Delta_u\alpha]\!](\mathcal{A})(\vec{b})(\vec{x}).
\end{aligned}$$

Case $\phi = t\,\theta\,0$. Informally, the delta to ϕ is $+1$ if the condition was previously false and becomes true by the change, -1 if the condition was previosly true and now becomes false, and 0 otherwise. The variables of ϕ can be assumed bound from the outside, thus the multiplicity of the tuple defined by ϕ is either 1 or 0. Consider the following truth table, which shows, for all truth values of ϕ and $\phi_{new} = (t + \Delta_u t)\,\theta\,0$, the value of $\Delta_u\phi$ as given in our definition:

(1) ϕ	(2) ϕ_{new}	(3) $\neg\phi \wedge \phi_{new}$	(4) $\phi \wedge \neg\phi_{new}$	(5) $\Delta_u\phi = (3) - (4)$
1	1	0	0	0
1	0	0	1	-1
0	1	1	0	$+1$
0	0	0	0	0

It is easy to see that for all truth values of $[\![\phi]\!](\mathcal{A})(\vec{b})$ and $[\![\phi_{new}]\!](\mathcal{A})(\vec{b})$, (5) = (2) − (1) in the above table, thus $[\![\Delta_u\phi]\!](\mathcal{A})(\vec{b}) = [\![\phi_{new}]\!](\mathcal{A})(\vec{b}) - [\![\phi]\!](\mathcal{A})(\vec{b})$.

Cases $R(\vec{x})$ and $S(\vec{x})$: $\Delta_{\pm R(\vec{t})}R(\vec{x})$ explicitly constructs the change to R: It evaluates to

$$\pm\{\vec{t}\} : \vec{y} \mapsto \begin{cases} \pm 1 & \ldots \quad \vec{t} = \vec{y} \\ 0 & \ldots \quad \text{otherwise.} \end{cases}$$

$\Delta_{\pm R(\vec{t})}S(\vec{x})$ (for $R \neq S$) evaluates to $0^{A[\mathbb{T}]}$.

Cases of constants and variables: The values of these do not change as the database changes, so their deltas are $0^{A[\mathbb{T}]}$. Note that even a variable term x has this property: it depends on the bound variables, but not on the database. □

Example 6.2 Consider the AGCA query

$$q = \mathrm{Sum}\big(C(c,n) * C(c',n)\big)$$

with bound variable c which is a slightly simplified but equivalent query to the one in Example 5.2. We abbreviate $C(c,n) * C(c',n)$ as α. Let us study the insertion respectively deletion of a single tuple (c_1, n_1) to/from C. Since

$$\begin{aligned} \Delta_{\pm C(c_1,n_1)}C(c,n) &= \pm(c = c_1) * (n = n_1), \\ \Delta_{\pm C(c_1,n_1)}C(c',n) &= \pm(c' = c_1) * (n = n_1), \end{aligned}$$

by the delta-rule for $*$,

$$\begin{aligned} \Delta_{\pm C(c_1,n_1)}\alpha &= (\pm(c = c_1) * (n = n_1)) * C(c',n) \\ &+ C(c,n) * (\pm(c' = c_1) * (n = n_1)) \\ &+ (\pm(c = c_1) * (n = n_1)) * (\pm(c' = c_1) * (n = n_1)). \end{aligned}$$

For the overall query, we get $\Delta_{\pm C(c_1,n_1)}q = \mathrm{Sum}(\Delta_{\pm C(c_1,n_1)}\alpha)$. □

AGCA is closed under taking deltas. Thus we can take deltas as often as we like – our queries are, so to say, *infinitely differentiable*. Next we define a construction to characterize the structural complexity of AGCA expressions and show that for a large class of expressions, taking deltas makes the expressions strictly simpler.

Definition 6.3 Let the (polynomial) *degree* deg of an AGCA expression be defined inductively as follows:

$$\begin{aligned}
\deg(\alpha * \beta) &:= \deg(\alpha) + \deg(\beta) \\
\deg(\alpha + \beta) &:= \max(\deg(\alpha), \deg(\beta)) \\
\deg(-\alpha) &:= \deg(\alpha) \\
\deg(\mathrm{Sum}(\alpha)) &:= \deg(\alpha) \\
\deg(\alpha\,\theta\,0) &:= \deg(\alpha) \\
\deg(R(\vec{x})) &:= 1.
\end{aligned}$$

For all other kinds of expressions, $\deg(\cdot) := 0$. □

The degree of a conjunctive query is the number of relation atoms joined together. In absence of further knowledge about the structure of the query (e.g., tree-width) or the data, the data complexity [34] of an AGCA query q given values for the bound variables (in other words, evaluating the query for one group) is $O(n^{\deg(q)})$.

An AGCA condition $t\,\theta\,0$ is *simple* if $\Delta t = 0$ for all update events. This is in particular true if t does not contain Sum subterms.

Theorem 6.4 *For any AGCA expression α with simple conditions only,*

$$\deg(\Delta\alpha) = \max(0, \deg(\alpha) - 1).$$

Proof Sketch. The proof is a straightforward induction combining Definition 6.3 with the definition of Δ. In particular, $\Delta R(\vec{x})$ is a constant expression containing *simple* condition atoms but no relational atom, so $\deg(\Delta R(\vec{x})) = 0$. □

Example 6.5 Consider the query of Example 6.2 and its delta.

$$\begin{aligned}
\Delta_{\pm C(c_1,n_1)} q = \mathrm{Sum}\Big(& (\pm(c = c_1) * (n = n_1)) * C(c', n) \\
+ \; & C(c, n) * (\pm(c' = c_1) * (n = n_1)) \\
+ \; & (\pm(c = c_1) * (n = n_1)) * (\pm(c' = c_1) * (n = n_1))\Big).
\end{aligned}$$

We have $\deg q[c] = 2$, $\deg \Delta_{\pm C(c_1,n_1)} q[c] = 1$, and

$$\begin{aligned}
\Delta_{\pm C(c_2,n_2)} \Delta_{\pm C(c_1,n_1)} q = \mathrm{Sum}\Big(& (\pm(c = c_1) * (n = n_1)) * (\pm(c' = c_2) * (n = n_2)) \\
+ \; & (\pm(c = c_2) * (n = n_2)) * (\pm(c' = c_1) * (n = n_1))\Big),
\end{aligned}$$

so $\deg q''[c] = 0$. The definition of Δ ensures that the delta of any query of degree 0 is 0, so the value as well as the degree of any derivative of q higher than q'' are 0, too. □

Theorem 6.4 guarantees that, for any AGCA expression with simple conditions only and a fixed k, the k-th delta-derivative has degree zero. Such an expression does not access the database: it only depends on the update. This will be key in the compilation result presented in Section **??**.

7 Back to Recursive Incremental View Maintenance

Given the development of Section 1.1, we now only need to tie together some lose ends to obtain the result that incremental maintenance of AGCA is in NC0.

We study data complexity, that is, we assume the query to be fixed. We consider AGCA queries with *simple conditions*, for which Theorem 6.4 holds.

We have seen in Section 1.1 that for a fixed query and a kind of single-tuple update (the insertion or deletion of a tuple into/from a relation), there is a trigger that consists of a sequence of statements that increment a GMR by the result of an AGCA query over views that does not use aggregation. Assuming multiplicities are stored as machine words and arithmetics is modulo maximum word size, each output bit depends only on a fixed number of input bits, and thus evaluating such a statement and the entire trigger given a single-tuple update is in NC.

Theorem 7.1 *The incremental evaluation of AGCA using triggers obtained by recursive incremental view maintenance is in NC0.*

This separates incremental view maintenance from nonincremental evaluation of AGCA, where output bits in general are the result of the aggregation of information from many input tuples; nonrecursive AGCA– just like relational algebra on bag relations – takes TC0 complexity.

A formal version of this result is obtained along the lines sketched in [21].

8 Related Work

There is a large literature on the incremental view maintenance problem (e.g. [5, 32, 4, 31, 6, 15, 16, 14, 35, 9, 11, 7, 26, 27, 22]). Work in this tradition expresses the delta to a query as a query itself, which is evaluated mostly using classical operator-based query evaluation techniques. The ring $A[\mathbb{T}]$ of this paper adds to the state of the art in this area by simplifying and generalizing the machinery for obtaining delta queries. Previous work in this area generally does not address aggregates nested into conditions, while this paper does. The work closest to ours is that of [15, 16], where a multiset semantics and a counting-based model of incremental computation are proposed. None of the previous work, however, applies delta processing to deltas recursively as done here.

Treatments of bag semantics based on an algebraic connection to counting also appear in [25, 13, 12]. The goals of this paper are different from those of [25, 13, 12]. $\mathbb{Z}$-relations [12] are relations in which the tuples have integer (including negative) multiplicities and they are used to study the equivalence, rewriting, and optimization of certain queries with negation, with an application to incremental view maintenance. There is a fundamental technical difference between the algebraic modeling of [13, 12] and the one in this paper, in that we consider untyped relations which allows us to define union and join as total operations, yielding a ring structure.

There is a considerable body of work on incremental computation by the programming languages research community [8, 30, 2]. This work is different in spirit since it has the objective to speed up Turing-complete programming languages, which is substantially harder. The restriction to query languages with strong algebraic properties allows for delta processing in a form that is not possible for general-purpose programming languages.

Classical complexity classes are not well suited for characterizing the complexity of incremental query evaluation in databases. In the database theory literature, there is some work on

dynamic complexity classes such as DynFO [29, 9, 26, 27, 19, 18, 28], which fills this gap. DynFO essentially captures the expressive power that relational calculus yields for incremental computation (for tuple insertions and deletes). DynFO is more powerful than FO used nonincrementally. For example, it is well known that graph reachability cannot be expressed in FO; however, there are representations of reachability in undirected graphs that can be incrementally maintained using first-order interpretations (i.e., in DynFO). Graph reachability on *directed* graphs can be incrementally maintained using TC0 – in DynTC0 [18]. In short, this thread of work studies how much additional expressive power one obtains by using a classical query language (such as FO) to compute increments. In a sense, we do the opposite – we ask with how much less power (and cost) we can make do by incremental computation for the evaluation of queries in practical languages. The main result of this paper suggests that DynFO relates to FO similarly as the complexity class TC0 relates to NC0. No claim is made that such a relationship holds strictly, i.e. that TC0 = Dyn-NC0, but it is plausible that a result in this spirit could be achieved.

9 Future work

We have come to expect that query algebras are based on cylindric algebras [17], on which research is rather isolated from mainstream mathematics. The ring $A[\mathbb{T}]$ and the query language AGCA, which is based on it, connect database queries with mainstream algebra. In future work, it may be worth looking into applications of $A[\mathbb{T}]$ in the context of constraint, probabilistic, and scientific databases, where a better integration of query languages with numerical computation and the standard framework of abstract algebra is desirable.

Acknowledgments

This article creates a foundation for the author's current project on practical query compilation, DBToaster [3]. The author thanks Yanif Ahmad and Oliver Kennedy for many insightful discussions and Val Tannen and the anonymous reviewers of PODS for their advice.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. *ACM Trans. Program. Lang. Syst.*, 28(6):990–1034, 2006.

[3] Y. Ahmad and C. Koch. DBToaster: A SQL compiler for high-performance delta processing in main-memory databases. *PVLDB*, 2(2):1566–1569, 2009.

[4] J. A. Blakeley, P.-Å. Larson, and F. W. Tompa. Efficiently updating materialized views. In *Proc. SIGMOD Conference*, pages 61–71, 1986.

[5] P. Buneman and E. K. Clemons. Efficient monitoring relational databases. *ACM Trans. Database Syst.*, 4(3):368–382, 1979.

[6] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proc. VLDB*, pages 577–589, 1991.

[7] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *Proc. SIGMOD*, pages 469–480, 1996.

[8] A. J. Demers, T. W. Reps, and T. Teitelbaum. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *Proc. POPL*, pages 105–116, 1981.

[9] G. Dong and J. Su. Incremental and decremental evaluation of transitive closure by first-order queries. *Inf. Comput.*, 120(1):101–106, 1995.

[10] D. S. Dummit and R. M. Foote. *Abstract Algebra*. John Wiley & Sons, 3rd edition, 2004.

[11] S. Ghandeharizadeh, R. Hull, and D. Jacobs. Heraclitus: Elevating deltas to be first-class citizens in a database programming language. *ACM Transactions on Database Systems*, 21(3):370–426, Sept. 1996.

[12] T. J. Green, Z. Ives, and V. Tannen. Reconcilable differences. In *Proc. ICDT*, St. Petersburg, Russia, 2009.

[13] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *Proc. PODS*, pages 31–40, 2007.

[14] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proc. SIGMOD*, 1995.

[15] A. Gupta, D. Katiyar, and I. S. Mumick. Counting solutions to the view maintenance problem. In *Proc. Workshop on Deductive Databases, JICSLP*, 1992.

[16] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proc. SIGMOD Conference*, pages 157–166, 1993.

[17] L. Henkin, J. Monk, and A. Tarski. *Cylindric Algebras, Part I.* North-Holland, 1971.

[18] W. Hesse. The dynamic complexity of transitive closure is in DynTC^0. *Theor. Comput. Sci.*, 296(3):473–485, 2003.

[19] W. Hesse and N. Immerman. Complete problems for dynamic complexity classes. In *Proc. LICS*, 2002.

[20] D. S. Johnson. A catalog of complexity classes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume 1, chapter 2, pages 67–161. Elsevier Science Publishers B.V., 1990.

[21] C. Koch. Incremental query evaluation in a ring of databases. In *Proc. PODS*, pages 87–98, 2010.

[22] Y. Kotidis and N. Roussopoulos. A case for dynamic view management. *ACM Transactions on Database Systems*, 26(4):388–423, 2001.

[23] S. Lang. *Algebra*. Graduate Texts in Mathematics. Springer-Verlag, revised 3rd edition, 2002.

[24] L. Libkin. *Elements of Finite Model Theory*. Springer, 2004.

[25] L. Libkin and L. Wong. Some properties of query languages for bags. In *Proc. DBPL*, pages 97–114, 1993.

[26] L. Libkin and L. Wong. Incremental recomputation of recursive queries with nested sets and aggregate functions. In *Proc. DBPL*, pages 222–238, 1997.

[27] L. Libkin and L. Wong. On the power of incremental evaluation in SQL-like languages. In *Proc. DBPL*, pages 17–30, 1999.

[28] P. B. Miltersen, S. Subramanian, J. S. Vitter, and R. Tamassia. Complexity models for incremental computation. *Theor. Comput. Sci.*, 130(1):203–236, 1994.

[29] S. Patnaik and N. Immerman. Dyn-FO: A parallel, dynamic complexity class. *J. Comput. Syst. Sci.*, 55(2):199–209, 1997.

[30] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Proc. POPL*, 1989.

[31] N. Roussopoulos. An incremental access method for viewcache: Concept, algorithms, and cost analysis. *ACM Transactions on Database Systems*, 16(3):535–563, 1991.

[32] O. Shmueli and A. Itai. Maintenance of views. In B. Yormark, editor, *Proc. SIGMOD*, pages 240–255. ACM Press, 1984.

[33] R. Smolenski. Algebraic methods in the theory of lower bounds for boolean circuit complexity. In *Proc. STOC*, pages 77–82, 1987.

[34] M. Y. Vardi. The complexity of relational query languages. In *Proc. STOC*, pages 137–146, May 1982.

[35] W. P. Yan and P.-Å. Larson. Eager aggregation and lazy aggregation. In *Proc. VLDB*, pages 345–357, 1995.