Incremental Query Evaluation in a Ring of Databases

Christoph Koch
Dept. of Computer Science
Cornell University
koch@cs.cornell.edu

ABSTRACT

This paper approaches the incremental view maintenance problem from an algebraic perspective. We construct a ring of databases and use it as the foundation of the design of a query calculus that allows to express powerful aggregate queries. The query calculus inherits key properties of the ring, such as having a normal form of polynomials and being closed under computing inverses and delta queries. The k-th delta of a polynomial query of degree k without nesting is purely a function of the update, not of the database. This gives rise to a method of eliminating expensive query operators such as joins 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.

Categories and Subject Descriptors

F.1.2 [**Computation by Abstract Devices**]: Modes of Computation – *Parallelism and concurrency*; H.2.3 [**Database Management**]: Languages – *Query languages*; H.2.4 [**Database Management**]: Systems – *Query processing*

General Terms

Algorithms, Languages, Theory

Keywords

Incremental View Maintenance, Algebra

1. INTRODUCTION

It is widely acknowledged that classical measures of algorithm efficiency do not do database query evaluation justice. In most cases, databases change only incrementally, in updates small compared to the overall database size. This fact has been addressed by the research community by a large amount of work on the incremental view maintenance problem, which aims at cheaply computing an increment to a materialized query result given an update [7, 33, 5, 32, 8, 17, 18, 16, 36, 11, 13, 9, 26, 27, 29, 23].

Most work on incremental view maintenance has aimed at expressing the delta (=change) to the result of a query Q on a database D as follows. Suppose that we denote changes to D by ΔD, which captures both insertions and deletions. Let us denote the updated database by $D' = D + \Delta D$, where $+$ is a way of combining a database with a change to it, made precise later; it is a generalisation of the union operation of relational algebra. We would like to express the change to the result of Q as a query ΔQ that depends on both D and ΔD such that $Q(D') = Q(D) + \Delta Q(D, \Delta D)$. The intuition is that evaluating $\Delta Q(D, \Delta D)$ and using the result to update a materialized representation of $Q(D)$ will often be faster than recomputing Q on the updated database D'. The practical benefits of incremental view maintenance are real and have led to the integration of such techniques into commercial database management systems. But can we also make a complexity-theoretic argument that state-of-the-art incremental query evaluation is more efficient than nonincremental evaluation?

Unfortunately, there is an argument that suggests the answer is no: If we consider a query language such as the conjunctive queries or relational algebra with or without aggregates, the image of the language under taking deltas is the full language: Given an arbitrary query Q, there is another query of the language whose delta is Q. This suggests that incremental view maintenance is not fundamentally easier than nonincremental query evaluation.

In this paper, we show that this intuition is fortunately incorrect if one looks beyond classical delta processing. Recall that the circuit complexity classes $\mathrm{NC0} \subseteq \mathrm{AC0} \subsetneq \mathrm{TC0}$ (cf. [22]) 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. This paper shows that, for a large class of aggregate queries, applying a fixed update to a materialized view is in the complexity class NC0, while non-incremental evaluation takes TC0 and AC0 for queries with and without aggregates, respectively. Since NC0 and TC0 have been separated [34], incremental evaluation is indeed easier than nonincremental evaluation.

This has practical implications: While we cannot assume to have sufficiently many machines available to achieve constant-time processing for very large inputs, the NC0 re-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PODS'10, June 6–11, 2010, Indianapolis, Indiana, USA.
Copyright 2010 ACM 978-1-4503-0033-9/10/06 ...$10.00.

sult asserts *embarassing parallelism* beyond the parallelism exhibited in nonincremental query evaluation (AC0/TC0). Standard schemes such as the Brent scheduling principle [6] can be used to create massively parallel implementations of incremental query evaluation on commodity computers, which require, as a lower bound, only constant time, rather than logarithmic time as for AC0/TC0.

This paper approaches the incremental view maintenance problem from a fresh perspective. We start by identifying the essence of delta processing: A query language needs to be closed under computing an additive inverse (as a generalization of the union operation on relations to support insertions and deletions) and the join operation has to be distributive over this addition to support normalization, factorization, and the taking of deltas of queries. Thus, the algebraic structure of a ring of databases is needed. The first contribution of this paper is to craft such a ring. It is subsequently interconnected with a ring of terms to form an expressive aggregate query calculus, AGCA, which has the algebraic properties needed to define delta processing in a compositional way.

AGCA is closed under taking deltas. Moreover, for an AGCA query Q without nested aggregates, ΔQ is structurally strictly simpler than Q. We formalize this by the notion of the degree of a query, which roughly corresponds to the degree of a polynomial bound on its worst-case data complexity (for a conjunctive query, it is the number of its atoms). 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.

This gives rise to an aggressive *recursive* incremental view maintenance mechanism (cf. [3]) which is ultimately the key to the NC0 result. Given a query Q of degree k, we incrementally maintain a materialized view of Q by adding, on a single-tuple update $\pm\vec{t}$, the result of $\Delta Q(D, \pm\vec{t})$ to the view $Q(D)$. The key idea is now to materialize a map $\pm\vec{t} \mapsto \Delta Q(D, \pm\vec{t})$, for all possible updates $\pm\vec{t}$. This map can be expressed as a single aggregate query with group-by; the different updates $\pm\vec{t}$ form the groups. We incrementally maintain the ΔQ map for changes to D as well, using a delta to ΔQ, and so on. This leads to a recursive query compilation scheme. Overall, Q can be incrementally maintained by a hierarchy of k layers of materialized views, requiring only a simple form of message passing between the views to keep all of them up to date. In most traditional database query processors, the basic building blocks of delta queries are large-grained operators such as joins. We instead compile queries to programs that are not based on classical query operators. The updating of each value in the views only requires a constant number of arithmetic operations on pairs of numbers.

We also show that 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.

The ring construction, delta processing techniques, and compilation algorithms of this paper constitute a streamlined and cleaner way of performing and presenting incremental view maintenance. These contributions are of independent interest, beyond parallel complexity.

The paper 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 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 6 introduces a low-level language, NC0C, to which we later compile queries. NC0C admits massively parallel evaluation; we give a circuit complexity characterization. Section 7 presents algorithms for compiling queries to NC0C, starting with a simple algorithm that we subsequently refine. Section 8 discusses this and related work.

2. ALGEBRAIC FOUNDATIONS

Let us recall some basic definitions from algebra:

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 *ring* over base set A is a tuple $(A, +, *, 0)$ with two operations $+$ and $*$ (called addition and multiplication, respectively) such that $(A, +, 0)$ is a commutative group, $(A, *)$ is a semigroup, 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 *with identity* $(A, +, *, 0, 1)$ is a ring in which $(A, *, 1)$ is a monoid. A 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 $1 + x = 0$. □

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 denote the ring (e.g., $\mathbb{Z}$ for $(\mathbb{Z}, +, *, 0, 1)$). In a monoid, there is a *unique* identity element and 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 [12]). Thus, in particular, a 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).

DEFINITION 2.3. Let A be a commutative ring and let $(G, *^G, 1^G)$ 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 $x \in G$. We define addition and multiplication in $A[G]$ as

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

Then $A[G]$ is called the *monoid algebra* of G over A. □

PROPOSITION 2.4 (CF. P.104F IN [24]). *A monoid algebra is a commutative ring with identity.*

In particular, monoid algebras $A[G]$ are closed under multiplication, i.e., $(\alpha * \beta)(x) = 0$ for all but a finite number of elements $x \in G$. The neutral elements are $0 : x \mapsto 0$ and

$$1 : x \mapsto \left\{ \begin{array}{lll} 1 & \ldots & x = 1^G \\ 0 & \ldots & x \neq 1^G. \end{array} \right.$$

We say that a monoid $(G, *)$ has a *zero* if there is an element $0 \in G$ such that $0 * g = g * 0 = 0$ for all $g \in G$. A *ring homomorphism* is a function $\phi : R \to S$ between two 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 $a, b \in R$.

Lemma 2.5. *Let G be a monoid with 0 and A be a commutative ring. Then the map $\phi : \alpha \mapsto \alpha|_{G-\{0\}}$ is a surjective ring homomorphism from the monoid algebra $A[G]$ to the set of all functions $(G - \{0\}) \to A$.*

Proof. Let $(G, *^G)$ be a monoid with zero 0^G. Consider the monoid algebra $A[G]$ of G over commutative ring A.

Let I be the subset of $A[G]$ consisting of those elements α of $A[G]$ that have $\alpha(x) = 0^A$ for all $x \neq 0^G$. Note that I is closed under $+$, $-$, and $*$: The mapping $I \to A$ with $\alpha \mapsto \alpha(0^G)$ is an isomorphism between I and A. Since A is a ring, I is a sub-ring of $A[G]$.

I is an *ideal* of $A[G]$, that is, $(r * i) \in I$ for all $r \in A[G]$ and all $i \in I$: Consider an arbitrary $x \in G$. By definition, $(r * i)(x) = \sum_{x = y *^G z} r(y) *^A i(z)$. Since $i(z) = 0^A$ unless $z = 0^G$ and $y *^G 0^G = 0^G$, we must have $(r * i)(x) = 0^A$ unless $x = 0^G$, and indeed $(r * i) \in I$.

We use the converse of the first isomorphism theorem for rings (cf. Theorem 7(2) in Chapter 7 of [12]):

Lemma 2.6. *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.*

The natural projection $\phi : A[G] \to A[G]/I$ is the map $\phi : \alpha \mapsto \alpha|_{G-\{0\}}$. The elements of the quotient ring $A[G]/I$ are precisely the functions $(G - \{0\}) \to A$. $\square$

3. A RING OF DATABASES

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 study a structure $(\mathbb{Z}_{\text{Rel}}, +, *, 0, 1)$ (or just $\mathbb{Z}_{\text{Rel}}$, for short) of generalized multiset relations – collections of tuples with *integer* multiplicities 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 elements of $\mathbb{Z}_{\text{Rel}}$). The schema polymorphism of tuples in our generalized multiset relations just serves the purpose of accommodating such total operator definitions: We have to be able to union together relations containing tuples of different schema.

We use the notations $f : x \mapsto v$, $f(x) := v$, and $f := \{x \mapsto v \mid x \in \text{dom}(f)\}$ interchangeably to define functions, with a preference for the latter when the domain $\text{dom}(f)$ 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 *(typed) tuple* $\vec{t}$ is a partial function from a vocabulary of column names $\text{dom}(\vec{t})$ to data values (that is, $\vec{t}$ is not just a tuple of values but has an associated schema of its own). Throughout Sections 2 to 5, all tuples are typed, and Tup denotes the set of all typed tuples.

For a number of technical reasons, we will also use classical singleton relations (without multiplicities) in what follows. We write $\{\vec{t}\}$ to construct a singleton relation with schema $\text{sch}(\{\vec{t}\}) = \text{dom}(\vec{t})$ from $\vec{t}$ and use the classical natural join operator $\bowtie$ on such singletons. The set of all singletons is denoted by Sng (i.e., $\text{Sng} := \{\{\vec{t}\} \mid \vec{t} \in \text{Tup}\}$).

Definition 3.1. A *generalized multiset relation (gmr)* is a function $R : \text{Tup} \to \mathbb{Z}$ such that $R(\vec{t}) \neq 0$ for at most a finite number of tuples $\vec{t}$. The set of all such functions is denoted by $\mathbb{Z}_{\text{Rel}}$. $\square$

Such a function indicates the multiplicity with which each tuple of Tup occurs in the gmr. Tuples can have negative multiplicities.

The operations $+$ and $*$ of $\mathbb{Z}_{\text{Rel}}$ are defined as follows.

Definition 3.2. For $R, S \in \mathbb{Z}_{\text{Rel}}$,

$$\begin{array}{rcl} R + S & : & \vec{x} \mapsto \big(R(\vec{x}) + S(\vec{x})\big) \\ (-R) & : & \vec{x} \mapsto (-R(\vec{x})) \\ R * S & : & \vec{x} \mapsto \displaystyle\sum_{\{\vec{x}\} = \{\vec{a}\} \bowtie \{\vec{b}\}} R(\vec{a}) * S(\vec{b}) \\ 1 & : & \vec{x} \mapsto \left\{ \begin{array}{lll} 1 & \ldots & \vec{x} = \langle\rangle \\ 0 & \ldots & \vec{x} \neq \langle\rangle \end{array} \right. \\ 0 & : & \vec{x} \mapsto 0 \end{array}$$

$\square$

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 3.2 is similar to the definition of a monoid algebra; this is made precise in the proof of Proposition 3.4.

Example 3.3. Consider the three gmrs of $\mathbb{Z}_{\text{Rel}}$

R	A	B	
	1		$\mapsto -1$
	2	3	$\mapsto 2$

S	C	
	5	$\mapsto 2$

T	B	C	
	3	5	$\mapsto 1$
	4	6	$\mapsto -3$

over column name vocabulary $\Sigma = \{A, B, C\}$ and value domain $\mathbb{N}$. 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 $\Sigma \to \mathbb{N}$: $\{A \mapsto 1\}$ and $\{A \mapsto 2; B \mapsto 3\}$.

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

$S + T$	B	C	
		5	$\mapsto 2$
	3	5	$\mapsto 1$
	4	6	$\mapsto -3$

$R * (S + T)$	A	B	C	
	1		5	$\mapsto -2$
	1	3	5	$\mapsto -1$
	1	4	6	$\mapsto 3$
	2	3	5	$\mapsto 6$

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

Proposition 3.4. *$(\mathbb{Z}_{\text{Rel}}, +, *, 0, 1)$ is a commutative ring with identity.*

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

Consider the monoid algebra $\mathbb{Z}[\text{Sng}_\emptyset]$ of $\text{Sng}_\emptyset$ over $\mathbb{Z}$. By Lemma 2.5, the map $\phi : \alpha \mapsto \alpha|_{\text{Sng}}$ is a surjective ring homomorphism from $\mathbb{Z}[\text{Sng}_\emptyset]$ to the set of all functions $\text{Sng} \to \mathbb{Z}$, thus the image R of ϕ is a commutative ring with identity. The map $\theta : \alpha \mapsto \left(\vec{t} \mapsto \alpha(\{\vec{t}\})\right)$ from R to $\mathbb{Z}_{\text{Rel}}$ is a ring isomorphism. $\Box$

The ring $\mathbb{Z}_{\text{Rel}}$ is not an *integral domain*, however. It has *zero divisors*: The result of joining two nonempty relations may be empty.

Discussion and Justification

The ring $\mathbb{Z}_{\text{Rel}}$ as a $\mathbb{Z}$-module. Let $\mathbb{Z}_{\text{Sng}}$ ($\subseteq \mathbb{Z}_{\text{Rel}}$) be the set of functions of the form

$$\{\vec{t}\} : \vec{x} \mapsto \begin{cases} 1 & \dots \quad \vec{x} = \vec{t} \\ 0 & \dots \quad \vec{x} \neq \vec{t} \end{cases}$$

for tuples $\vec{t}$. We have just overloaded $\{\vec{t}\}$ as an element of both Sng and $\mathbb{Z}_{\text{Sng}}$. No problems will arise from this. Let $k\{\vec{t}\}$, for $k \in \mathbb{N}$, denote $\underbrace{\{\vec{t}\} + \cdots + \{\vec{t}\}}_{k \text{ times}}$, and let $(-k)\{\vec{t}\}$ denote $-(k\{\vec{t}\}) = k(-\{\vec{t}\})$.

Each $\alpha \in \mathbb{Z}_{\text{Rel}}$ can be written as a finite sum

$$v_\alpha = \sum_{\vec{t}} \alpha(\vec{t})\{\vec{t}\}$$

of elements of $\mathbb{Z}_{\text{Sng}}$ and their inverses. Since $+^{\mathbb{Z}_{\text{Rel}}}$ is associative and commutative, this sum is essentially *unique*. There is a bijection between the elements $\alpha \in \mathbb{Z}_{\text{Rel}}$ and the elements v_α defined by these sums.

Thus, $\mathbb{Z}_{\text{Sng}}$ *generates* $\mathbb{Z}_{\text{Rel}}$ and $\mathbb{Z}_{\text{Rel}}$ is a $\mathbb{Z}$-module that is *free* on $\mathbb{Z}_{\text{Sng}}$; $\mathbb{Z}_{\text{Sng}}$ is the *basis* of $\mathbb{Z}_{\text{Rel}}$ (since we will not need the definitions of these notions further, they are not introduced here – cf. e.g. [12], Sections 10.1 and 10.3). If we were to replace $\mathbb{Z}$ in $\mathbb{Z}_{\text{Rel}}$ by a field such as $\mathbb{R}$, we would have an infinite-dimensional vector space.

Viewing $\mathbb{Z}_{\text{Rel}}$ as a $\mathbb{Z}$-module means to ignore its operation $*$, and the fact that it is distributive with $+$, however.

We say that an operation $\circ$ is *conservative over* $\bowtie$ if $R \circ S = R \bowtie S$ on all $R, S \in \mathbb{Z}_{\text{Rel}}$ that are classical relations without duplicates or polymorphic tuples. If we accept the definition of $+$ in $\mathbb{Z}_{\text{Rel}}$ as natural, then the definition of $*$ (which may feel less natural at first) is uniquely determined by distributivity, if we want $*$ to be conservative over $\bowtie$.

Let $\circ$ be an arbitrary multiplication operation on $\mathbb{Z}_{\text{Rel}}$ that is distributive with $+$.

$$\begin{aligned} v_\alpha \circ v_\beta &:= \Big(\sum_{\vec{a}} \alpha(\vec{a})\{\vec{a}\}\Big) \circ \Big(\sum_{\vec{b}} \beta(\vec{b})\{\vec{b}\}\Big) \\ &= \sum_{\vec{a},\vec{b}} \big(\alpha(\vec{a}) *^{\mathbb{Z}} \beta(\vec{b})\big)\big(\{\vec{a}\} \circ \{\vec{b}\}\big) \end{aligned}$$

This follows from the distributivity of $\circ$ and the fact that $\alpha(\vec{a})\{\vec{a}\}$ and $\beta(\vec{b})\{\vec{b}\}$ are actually sums. Since $\circ$ is conservative over $\bowtie$ and $\{\vec{a}\}$ and $\{\vec{b}\}$ are classical singleton relations, $\{\vec{a}\} \circ \{\vec{b}\} = \{\vec{a}\} \bowtie \{\vec{b}\}$, and

$$(\alpha \circ \beta)(\vec{x}) = \sum_{\vec{a},\vec{b}:\{\vec{a}\}\bowtie\{\vec{b}\}=\{\vec{x}\}} \alpha(\vec{a}) * \beta(\vec{b}),$$

as in Definition 3.2, so $\circ$ is $*$ and $v_\alpha * v_\beta = v_{\alpha*\beta}$.

Musings. It is appealing that our natural choice of $+^{\mathbb{Z}_{\text{Rel}}}$ completely determines $*^{\mathbb{Z}_{\text{Rel}}}$ in any ring, in the way just described, and that the structure of a monoid algebra arises necessarily and naturally.

We have come to expect that query algebras are based on cylindric algebras [19], on which research is rather isolated from mainstream mathematics. In future work, it may be worth looking into extensions of $\mathbb{Z}_{\text{Rel}}$ for applications such as constraint, probabilistic, and scientific databases, where a better integration of query languages with numerical computation and the now standard framework of abstract algebra is desirable.

Relationship to relational algebra. $\mathbb{Z}_{\text{Rel}}$ is no multiset version of relational algebra. Specifically, difference and explicit projection are missing. Observe that, throughout this paper, $R - S = R + (-S)$ does not refer to the difference operation of relational algebra, but to the additive inverse in $\mathbb{Z}_{\text{Rel}}$: for instance, $\emptyset - R = -R$ in $\mathbb{Z}_{\text{Rel}}$, while the syntactically same expression 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*. For the purposes of incremental view maintenance later in the paper, we will use $\mathbb{Z}_{\text{Rel}}$ in a way that we never really delete too much, but the properties of the ring just defined will still be essential.

4. AGGREGATION CALCULUS

In this section, we introduce the query language studied in this paper, AGCA (which stands for *AGgregation CAlculus*). AGCA defines two forms of query expressions, *formulae* and *terms*. Formulae evaluate to elements of the ring $(\mathbb{Z}_{\text{Rel}}, +, *, 0, 1)$ of gmrs and terms evaluate to elements of the ring of rational numbers $(\mathbb{Q}, +, *, 0, 1)$. We connect terms and formulae mutually recursively, creating a powerful language for expressing aggregate queries. Both formulae and terms, and thus the overall query language, inherit the key properties of polynomial rings in that they have an additive inverse, a normal form of polynomial expressions, and admit a form of factorization. These properties will be the basis of delta processing and incremental query evaluation in subsequent sections.

Syntax. AGCA consists of *formulae* and of *terms*. Formulae are expressions built from atomic formulae using $+$, $-$, and $*$. The atomic formulae are *true*, *false*, relational atoms $R(\vec{x})$ where $\vec{x}$ is a tuple of variables, and atomic conditions of the form $t\ \theta\ 0$ comparing term t with 0 using comparison operations θ from $=$, $\neq$, $>$, $\geq$, $<$, and $\leq$. AGCA terms are built from variables, built-in function calls (constants are functions with zero arguments), and aggregate sums (Sum) using addition, its inverse, and multiplication. Built-in functions compute their result entirely based on their input terms, not accessing the database.

The abstract syntax of formulae ϕ and terms t (given variables x, relation names R, comparison operators θ, and constants/builtin functions f) can be given by the EBNF

$$\begin{array}{lll} \phi & ::- & \phi * \phi \mid \phi + \phi \mid -\phi \mid \text{true} \mid \text{false} \mid R(\vec{x}) \mid t\ \theta\ 0 \\ t & ::- & t * t \mid t + t \mid -t \mid \quad f(t^*) \quad \mid \quad x \quad \mid \text{Sum}(t, \phi) \end{array}$$

The atoms *true* and *false* are just syntactic sugar for $0 = 0$ and $0 \neq 0$, respectively.

$$\begin{array}{rcl}
\mathrm{safe}_B(R(x_1,\ldots,x_k)) & := & \{x_1,\ldots,x_k\}\cup B \\
\mathrm{safe}_B(\phi * \psi) & := & \mathrm{safe}_B(\phi)\cup \mathrm{safe}_{\mathrm{safe}_B(\phi)}(\psi) \\
\mathrm{safe}_B(\phi+\psi) & := & \mathrm{safe}_B(\phi)\cap \mathrm{safe}_B(\psi) \\
\mathrm{safe}_B(-\phi) & := & \mathrm{safe}_B(\phi) \\
\mathrm{safe}_B(x=y) & := & \begin{cases} B\cup\{x,y\} & \ldots x \text{ or } y \text{ is in } B \\ B & \ldots \text{otherwise.}\end{cases}
\end{array}$$

Figure 1: Safety of AGCA formulae.

Bound and safe variables. Formulae and terms are evaluated relative to a given set of *bound variables*. Here *bound* is a notion in the spirit of binding patterns [31] – parameters given from the outside – rather than that of variables bound by quantifiers. Given a set of bound variables B, the *safe variables* of a formula are defined as shown in Figure 1. Observe that $\mathrm{safe}_B(\phi) \supseteq B$ for all B and ϕ and that safety of $-\phi$ differs from safety of $\neg\phi$ in relational calculus ($\mathrm{safe}(\neg\phi)=\emptyset$, cf. range-restriction in [1]). A formula is safe if all its variables are safe. Given a term $\mathrm{Sum}(t,\phi)$ with bound variables B, the bound variables of ϕ are B and the bound variables of t are the variables $\mathrm{safe}_B(\phi)$. Term $\mathrm{Sum}(t,\phi)$ is safe if t and ϕ are safe. The bound variables of a subterm are the bound variables of the term. Variables occurring as terms are safe if they are bound. A term is safe if all its variables and Sum atoms are safe.

EXAMPLE 4.1. Given singleton bound variable set $\{y\}$,

$$\mathrm{Sum}(u*f(z), \underbrace{(\underbrace{(\underbrace{R(x,z)}_{x,y,z}+\underbrace{(y=z)}_{y,z})}_{y,z}*(z=w))}_{y,z,w})$$

is unsafe and thus invalid: The safe variables of the formula are $\{y,z,w\}$, so u is not bound in the term $u*f(z)$. The overall term becomes valid for bound variables $\{u,y\}$. □

Semantics. The formal semantics of AGCA is given by mutually recursive functions $[\![\cdot]\!]_F(\cdot,\cdot)$ and $[\![\cdot]\!]_T(\cdot,\cdot)$ for formulae and terms, respectively. We treat variable names as additional column names. Given database $\mathcal{A}$ and a bound variable tuple $\vec{b}$ (i.e., a function that maps each bound variable to a value), $[\![\phi]\!]_F(\mathcal{A},\vec{b})$ evaluates to an element of $\mathbb{Z}_{\mathrm{Rel}}$ and $[\![t]\!]_T(\mathcal{A},\vec{b})$ evaluates to a value from $\mathbb{Q}$. The semantics of AGCA formulae is defined as follows.

$$\begin{array}{rcl}
[\![\phi+\psi]\!]_F(\mathcal{A},\vec{b}) & := & [\![\phi]\!]_F(\mathcal{A},\vec{b}) +^{\mathbb{Z}_{\mathrm{Rel}}} [\![\psi]\!]_F(\mathcal{A},\vec{b}) \\
[\![-\phi]\!]_F(\mathcal{A},\vec{b}) & := & -^{\mathbb{Z}_{\mathrm{Rel}}}[\![\phi]\!]_F(\mathcal{A},\vec{b}) \\
[\![\phi * \psi]\!]_F(\mathcal{A},\vec{b}) & := & \vec{x}\mapsto \displaystyle\sum_{\{\vec{x}\}=\{\vec{y}\}\bowtie\{\vec{z}\}} [\![\phi]\!]_F(\mathcal{A},\vec{b})(\vec{y}) \\
& & \qquad *^{\mathbb{Z}} [\![\psi]\!]_F(\mathcal{A},\vec{y}|_{\mathrm{safe}_{\mathrm{dom}(\vec{b})}(\phi)})(\vec{z})
\end{array}$$

$$[\![t\,\theta\,0]\!]_F(\mathcal{A},\vec{b}) :=$$

$$\vec{x}\mapsto\begin{cases}
1 \ldots & t=y-z, \{y,z\}\cap\mathrm{dom}(\vec{b})\neq\emptyset, \\
& \{y,z\}\cup\mathrm{dom}(\vec{b})=\mathrm{dom}(\vec{x}), \vec{b}=\vec{x}|_{\mathrm{dom}(\vec{b})}, \\
& \text{and } \vec{x}(y)=\vec{x}(z) \\
1 \ldots & \text{otherwise, if } \vec{x}=\vec{b} \text{ and } [\![t]\!]_T(\mathcal{A},\vec{b})\,\theta\,0 \\
0 \ldots & \text{otherwise}
\end{cases}$$

$$[\![R(x_1,\ldots,x_k)]\!]_F(\mathcal{A},\vec{b}) :=$$

$$\vec{x}\mapsto\begin{cases}
R^{\mathcal{A}}(\vec{y}) & \ldots \quad R^{\mathcal{A}} \text{ is defined on } \vec{y}, \{\vec{x}\}\bowtie\{\vec{b}\}\neq\emptyset, \\
& \qquad \mathrm{dom}(\vec{x})=\{x_1,\ldots,x_k\}, \\
& \qquad \mathrm{dom}(\vec{y})=\{A_1,\ldots,A_k\}, \text{ and} \\
& \qquad \vec{x}(x_i)=\vec{y}(A_i) \text{ for all } 1\leq i\leq k \\
0 & \ldots \quad \text{otherwise.}
\end{cases}$$

The definitions of $[\![\phi*\psi]\!]_F$ and $[\![t\,\theta\,0]\!]_F$ are somewhat cumbersome because we need to pass information from relational atoms into condition atoms, just like in the relational calculus. Note that if ϕ and ψ do not contain condition atoms, we can equivalently use the simpler definition $[\![\phi*\psi]\!]_F(\mathcal{A},\vec{b}) := [\![\phi]\!]_F(\mathcal{A},\vec{b}) *^{\mathbb{Z}_{\mathrm{Rel}}} [\![\psi]\!]_F(\mathcal{A},\vec{b})$.

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

The semantics of AGCA terms is defined as

$$\begin{array}{rcl}
[\![x]\!]_T(\mathcal{A},\vec{b}) & := & \vec{b}(x) \\
[\![f(t_1,\ldots,t_k)]\!]_T(\mathcal{A},\vec{b}) & := & f\big([\![t_1]\!]_T(\mathcal{A},\vec{b}),\ldots,[\![t_k]\!]_T(\mathcal{A},\vec{b})\big) \\
[\![\mathrm{Sum}(t,\phi)]\!]_T(\mathcal{A},\vec{b}) & := & \displaystyle\sum_{(\vec{c}\mapsto i)\in[\![\phi]\!]_F(\mathcal{A},\vec{b})} i *^{\mathbb{Q}} [\![t]\!]_T\big(\mathcal{A},\vec{c}|_{\mathrm{safe}_{\mathrm{dom}(\vec{b})}(\phi)}\big)
\end{array}$$

Take $+$, $-$, and $*$ as built-in functions f ($+^{\mathbb{Q}}$, $-^{\mathbb{Q}}$, and $*^{\mathbb{Q}}$) and the definition is complete.

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

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

with *bound variables* $\vec{b}$. While $\mathrm{Sum}(\cdot,\cdot)$ computes exactly one number, we can think of an SQL aggregate query with group by clause as a comprehension

$$\{\langle\vec{b}, [\![\mathrm{Sum}(\cdot,\cdot)]\!](\mathcal{A},\vec{b})\rangle \mid \vec{b}\in\mathrm{Groups}\}.$$

Sections 6 and 7 will answer the question of how to obtain Groups, i.e., the domain of bound variable tuples, in a practical setting.

EXAMPLE 4.2. 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

$$\mathrm{Sum}\big(1, C(c_1,n_1)*C(c_2,n_2)*(n_1=n_2)\big)$$

with bound variable c_1. □

Relational completeness. AGCA captures all of first-order logic (FO) and more. Consider the following function *FO2AG* that maps FO to AGCA formulae:

$$\begin{array}{rcl}
\mathit{FO2AG}(\phi\wedge\psi) & := & \mathit{FO2AG}(\phi) * \mathit{FO2AG}(\psi) \\
\mathit{FO2AG}(\phi\vee\psi) & := & \mathit{FO2AG}(\phi) + \mathit{FO2AG}(\psi) \\
\mathit{FO2AG}(\exists x_1\ldots x_k\ \phi) & := & \mathrm{Sum}(1,\mathit{FO2AG}(\phi))\neq 0 \\
\mathit{FO2AG}(\neg\exists x_1\ldots x_k\ \phi) & := & \mathrm{Sum}(1,\mathit{FO2AG}(\phi))=0
\end{array}$$

FO2AG is the identity on atoms. Here, k may be 0 to support general negation. The key to simulating existential quantification is that Sum performs an implicit projection.

Let $\models$ be the usual satisfaction relation for FO. FO is captured by AGCA in the following sense.

THEOREM 4.3. *For an FO formula ϕ with free variables* $\mathrm{dom}(\vec{b})$, $(\mathcal{A}, \vec{b}) \models \phi \;\Leftrightarrow\; [\![FO2AG(\phi)]\!]_F(\mathcal{A}, \vec{b}) \neq 0^{\mathbb{Z}_{\mathrm{Rel}}}$.

EXAMPLE 4.4. A query asking for students S who have taken T all required courses C is expressed in FO as $S(x) \wedge \neg\exists y\, C(y) \wedge \neg T(x, y)$ with free variable x; *FO2AG* turns this into the AGCA query

$$\mathrm{Sum}\big(1, S(x) * \mathrm{Sum}\big(1, C(y) * \mathrm{Sum}(1, T(x, y)) = 0\big) = 0\big)$$

with bound variable x. (That is, bound in the sense presented above, not in the sense "not free".) □

Recursively monomial terms. Consider a language of expressions constructed from values ("constants") and the operations $+$ and $*$ of a ring A, plus variables. Let these expressions evaluate to elements of a multivariate polynomial ring in the natural way. Turning an expression into a polynomial, that is, a sum of flat products (the products are also known as *monomials*), just means to apply distributivity repeatedly until we end up with a polynomial. This can be combined with simplification operations based on the 1 and 0-elements and the additive inverse, i.e., $\alpha * 1$ maps to α, $\alpha * 0$ maps to 0, $\alpha + 0$ maps to α, $\alpha + (-\alpha)$ maps to 0, $-(-\alpha) = \alpha$, and $(-\alpha) * \beta = -(\alpha * \beta)$.

Such an algorithm for computing and simplifying expressions over a ring is immediately applicable to AGCA formulae and terms. In particular, the operation $*$ is distributive with $+$ despite its sideways information passing semantics. For arbitrary terms s and t and formulae ϕ and ψ, Sum terms can be simplified using the following equations (to be applied by replacing a left by a right hand side expression):

$$\begin{aligned}
\mathrm{Sum}(s + t, \phi) &= \mathrm{Sum}(s, \phi) + \mathrm{Sum}(t, \phi)\\
\mathrm{Sum}(t, \phi + \psi) &= \mathrm{Sum}(t, \phi) + \mathrm{Sum}(t, \psi)\\
\mathrm{Sum}(-t, \phi) &= -\mathrm{Sum}(t, \phi)\\
\mathrm{Sum}(t, -\phi) &= -\mathrm{Sum}(t, \phi)\\
\mathrm{Sum}(t, \mathit{true}) &= t\\
\mathrm{Sum}(t, \mathit{false}) &= 0\\
\mathrm{Sum}(0, \phi) &= 0.
\end{aligned}$$

We call a term that contains neither $+$ nor $-$ anywhere *recursively monomial.* The following result is based on a straightforward algorithm that rewrites an input term in a bottom-up pass using the above rules.

PROPOSITION 4.5. *Each AGCA term is equivalent to a finite sum $\pm t_1 \pm t_2 \cdots \pm t_m$ where $t_1, \ldots, t_m$ are recursively monomial terms.*

EXAMPLE 4.6. $\mathrm{Sum}\big(t, (-\phi) * ((-\psi) + \pi)\big)$ simplifies to the recursively monomial $\mathrm{Sum}(t, \phi * \psi) - \mathrm{Sum}(t, \phi * \pi)$. □

5. DELTA COMPUTATION

Given a ring $(A, +, *, 0, 1)$, let us distinguish between variables $x_1, \ldots, x_n$ and constants ($\in A$) in algebraic expressions over the ring. (Thus, we are now talking about a multivariate polynomial ring $A[x_1, \ldots, x_n]$.) A valuation $\Theta : \{x_1, \ldots, x_n\} \to A$ extends naturally to a valuation of expressions with Θ the identity on elements of A and $\Theta(\alpha \circ \beta) := \Theta(\alpha) \circ \Theta(\beta)$ for $\circ \in \{+, *\}$.

Let us look at a scenario where we change the valuation of the variables from Θ to Θ_{new}. Then an expression $\Delta\alpha$ capturing the change to the value of an expression α can be defined inductively as

$$\begin{aligned}
\Delta(\alpha + \beta) &:= (\Delta\alpha) + \Delta\beta\\
\Delta(\alpha * \beta) &:= ((\Delta\alpha) * \beta) + (\alpha * \Delta\beta) + ((\Delta\alpha) * \Delta\beta)\\
\Delta(-\alpha) &:= -\Delta\alpha\\
\Delta x &:= \Theta_{new}(x) - \Theta(x) \quad (x \text{ is a variable})\\
\Delta c &:= 0 \quad (c \in A)
\end{aligned}$$

where α and β are expressions.

PROPOSITION 5.1. *$\Theta_{new}(\alpha) = \Theta(\alpha) + \Delta\alpha$.*

Proof Sketch. The proof is a straightforward induction: only the two most interesting cases are shown.

Case $\alpha * \beta$: By the structure-preserving extension of variable valuations to expressions, $\Theta_{new}(\alpha * \beta) = \Theta_{new}(\alpha) * \Theta_{new}(\beta)$. By the induction hypothesis (the proposition), this is $(\Theta(\alpha) + \Delta\alpha) * (\Theta(\beta) + \Delta\beta)$. The claim follows from distributivity.

Case $-\alpha$: It follows immediately from the induction hypothesis that $-\Theta_{new}(\alpha) - (-\Theta(\alpha)) = -\Delta\alpha$. It holds that $-\Theta_{(new)}(\alpha) = \Theta_{(new)}(-\alpha)$. We conclude that $\Theta_{new}(-\alpha) - \Theta(-\alpha) = -\Delta\alpha = \Delta(-\alpha)$. □

Deltas for AGCA queries. We now consider databases containing classical multiset relations, i.e., in which the tuples have coherent schema. Our goal is, given an AGCA term or formula α, to construct a query $\Delta\alpha$ (which is a term if α is a term and a formula if α is a formula) that expresses the change made to the database by the insertion respectively deletion of a single tuple. We write $\pm R(\vec{t})$ to denote the insertion or deletion of a tuple $\vec{t}$ into/from relation R. We write $\Delta_{\pm R(\vec{t})}\alpha$ to denote the delta-query for such such an update. For atomic terms and formulae,

$$\begin{aligned}
\Delta_{\pm R(\vec{t})}\mathrm{Sum}(t, \phi) &:= \mathrm{Sum}((\Delta_{\pm R(\vec{t})}t), \phi)\\
&\quad + \mathrm{Sum}(t, (\Delta_{\pm R(\vec{t})}\phi))\\
&\quad + \mathrm{Sum}((\Delta_{\pm R(\vec{t})}t), (\Delta_{\pm R(\vec{t})}\phi))\\
\Delta_{\pm R(\vec{t})}(t\,\theta\,0) &:= \big(((t + \Delta_{\pm R(\vec{t})}t)\,\theta\,0) * (t\,\bar{\theta}\,0)\big)\\
&\quad - \big(((t + \Delta_{\pm R(\vec{t})}t)\,\bar{\theta}\,0) * (t\,\theta\,0)\big)\\
\Delta_{\pm R(\vec{t})}\big(R(x_1, \ldots, x_{\mathrm{sch}(R)})\big) &:= \pm \bigwedge_{i=1}^{|\mathrm{sch}(R)|} (x_i = t_i)\\
\Delta_{\pm R(\vec{t})}\big(S(x_1, \ldots, x_{\mathrm{sch}(S)})\big) &:= \text{false} \qquad (R \neq S)
\end{aligned}$$

where $\bar{\theta}$ is the complement of the relation θ (i.e., $\neq$ for $=$, $\geq$ for $<$, etc.) and $\mathrm{sch}(R)$ is the schema of R – the set of its column names. For all other atomic terms and formulae α, $\Delta_{\pm R(\vec{t})}\alpha$ is the zero-element of their respective rings (0 and *false*, respectively). The delta rules for nonatomic terms and formulae, i.e., for operations in the rings of terms and formulae $(+, -, *)$, are those provided just above Proposition 5.1.

PROPOSITION 5.2. *Given any database $\mathcal{A}$ and values $\vec{b}$ for the bound variables,*

$$[\![\alpha]\!](\mathcal{A} \pm R(\vec{t}), \vec{b}) = [\![\alpha]\!](\mathcal{A}, \vec{b}) \;+\; [\![\Delta_{\pm R(\vec{t})}\alpha]\!](\mathcal{A}, \vec{b}).$$

Proof Sketch. The proof is by a straightforward bottom-up induction on the syntax tree of AGCA expressions. Only the two most interesting cases are shown.

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 $\phi_{new} = (t + \Delta_{\pm R(\vec{t})} t)\,\theta\,0$, the value of $\Delta_{\pm R(\vec{t})}\phi$ as given in our definition:

(1)	(2)	(3)	(4)	(5)
ϕ	ϕ_{new}	$\neg\phi \wedge \phi_{new}$	$\phi \wedge \neg\phi_{new}$	$\Delta_{\pm R(\vec{t})}\phi = (3) - (4)$
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 $[\![\Delta_{\pm R(\vec{t})}\phi]\!](\mathcal{A}, \vec{b}) = [\![\phi_{new}]\!](\mathcal{A}, \vec{b}) - [\![\phi]\!](\mathcal{A}, \vec{b})$ is always true.

Case $\Delta_{\pm R(\vec{t})} R(\vec{x})$: $\Delta_{\pm R(\vec{t})} R(\vec{x})$ 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 $0^{\mathbb{Z}_{\text{Rel}}}$. □

Example 5.3. Consider the AGCA query

$$q[c_1] = \text{Sum}\big(1, C(c_1, n_1) * C(c_2, n_2) * (n_1 = n_2)\big)$$

from Example 4.2. By the notation $q[c_1]$, we mean that the query has bound variable c_1 (for the group-by column) and defines a map that represents the aggregate query result for each group c_1. We abbreviate $C(c_2, n_2) * (n_1 = n_2)$ as ϕ and $C(c_1, n_1) * \phi$ as ψ to keep the example short.

Let us study the insertion respectively deletion of a single tuple (c, n) to/from C. Since

$$\begin{aligned} \Delta_{\pm C(c,n)}(n_1 = n_2) &= 0 \\ \Delta_{\pm C(c,n)} C(c_i, n_i) &= \pm((c_i = c) * (n_i = n)), \end{aligned}$$

by the delta-rule for $*$,

$$\Delta_{\pm C(c,n)}\phi = \pm((c_2 = c) * (n_2 = n) * (n_1 = n_2)).$$

Again using the delta-rule for $*$, we get

$$\begin{aligned} \Delta_{\pm C(c,n)}\psi = &\big((\pm((c_1 = c) * (n_1 = n))) * \phi\big) + \\ &\big(C(c_1, n_1) * \big(\pm((c_2 = c) * (n_2 = n) * (n_1 = n_2))\big)\big) + \\ &\big(\pm((c_1 = c) * (n_1 = n)) * \\ &\quad (\pm((c_2 = c) * (n_2 = n) * (n_1 = n_2)))\big). \end{aligned}$$

By the delta-rule for Sum, since $\Delta_{\pm C(c,n)} 1 = 0$,

$$\Delta_{\pm C(c,n)} \text{Sum}(1, \psi) = \text{Sum}(1, \Delta_{\pm C(c,n)}\psi),$$

which, following Proposition 4.5, we can turn into the sum

$$\begin{aligned} \pm \quad & \text{Sum}(1, (c_1 = c) * (n_1 = n) * C(c_2, n_2) * (n_1 = n_2)) \\ \pm \quad & \text{Sum}(1, C(c_1, n_1) * (c_2 = c) * (n_2 = n) * (n_1 = n_2)) \\ + \quad & \text{Sum}(1, (c_1 = c) * (n_1 = n) * (c_2 = c) \\ & \qquad * (n_2 = n) * (n_1 = n_2)). \end{aligned}$$

□

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 5.4. Let the (polynomial) *degree* deg of an AGCA term respectively formula be defined inductively as follows (α, β are either terms or formulae):

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

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

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 [35] 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 5.5. *For any AGCA term or formula α with simple conditions only,* $\deg(\Delta\alpha) = \max(0, \deg(\alpha) - 1)$.

The proof is a straightforward induction combining Definition 5.4 with the definition of Δ.

Example 5.6. Consider the query of Example 5.3. We have $\deg q[c_1] = 2$ and $\deg \Delta_{+C(c,n)} q[c_1] = 1$. Computing $q''[c_1] = \Delta_{+C(c',n')}\Delta_{+C(c,n)} q[c_1]$ yields $\text{Sum}(1, (c_1 = c) * (c_2 = c') * (n_2 = n')) + \text{Sum}(1, (c_1 = c') * (n = n'))$, so $\deg q''[c_1] = 0$. 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 5.5 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.

6. NC0C PROGRAMS

This section introduces NC0C, a low-level language for incremental, update-event processing that admits massive parallelization. In an NC0C program, all state is represented by finite map data structures (associative arrays), mapping tuples to numbers. NC0C is a restricted imperative language, similar to C both in syntax and semantics. There are two important differences from C. The first is the initialization of undefined map values, which uses special syntax to account for the fact that NC0C procedures are update triggers for the map data structures. The second is looping, which is performed over a set of values that is intentionally

Time	ΔC	Δq[1]	q[1]	Δq[2]	q[2]	Δq[3]	q[3]	Δq[4]	q[4]
1	insert (1,US)	+1	1						
2	insert (2,UK)		1	+1	1				
3	insert (3,UK)		1	+q2[2,UK]	2	+q1[UK] + 1	2		
4	insert (4,US)	+q2[4,US]	2		2		2	+q1[US] + 1	2
5	delete (3,UK)		2	−q2[3,UK]	1	$\underbrace{-\mathrm{q1[UK]}}_{-2} \underbrace{-\ \mathrm{q2[3,UK]}}_{-1} + 1$	0		2
6	insert (3,US)	+q2[3,US]	3	+q2[3,US]	3	+q1[US] + 1	3	+q2[3,US]	3

Figure 2: Runtime trace of the NC0C program of Example 6.1.

not made explicit in the syntax of NC0C (but is particularly well-behaved – it is a form of constrained structural recursion over the map – and thus admits parallelisation).

In the next section, we will compile AGCA queries to NC0C programs that perform incremental view maintenance. It is worth noting that these compiled programs only use the maps to represent the view and auxiliary data; no database beyond these is accessed.

Syntax. A *map (read) access*, where m is a map name, is written as $m[\vec{x}]$. An *NC0C term* is an arithmetic expression built from variables, constants, map accesses, functional ifs of the form `if` ϕ `then` t `else` 0 or, using C syntax, (ϕ ? t : 0), and arithmetic operations $+$ and $*$. A condition ϕ is a boolean combination of (in)equalities over variables and constants.

An NC0C trigger is of the form

$$\texttt{on } \pm R(\vec{x}\vec{y}) \texttt{ \{ } s_1\texttt{;} \ldots\texttt{;}\ s_k \texttt{ \}}$$

where $\pm$ indicates insertion respectively deletion, R is a relation name, $\vec{x}\vec{y}$ are variables (the trigger arguments), and $s_1, \ldots, s_k$ are *NC0C statements* of the form

$$\texttt{foreach } \vec{z} \texttt{ do } m[\vec{x}\vec{z}]\langle t_{init}\rangle \ \pm\texttt{=}\ t \qquad (1)$$

where $\vec{z}$ are variables distinct from $\vec{x}\vec{y}$, t is an NC0C term, and t_{init} is an *NC0C* term without map accesses that uses only variables from $\vec{x}\vec{z}$, called the *initializer* of $m[\vec{x}\vec{z}]$. If t_{init} is 0, we may omit it and write $m[\vec{x}\vec{z}]$ rather than $m[\vec{x}\vec{z}]\langle 0\rangle$. Let $m_1[\vec{v}_1], \ldots, m_k[\vec{v}_k]$ be the map accesses in the right-hand side term t. Then m, m_1, $\ldots$, m_k must be pairwise distinct and the variables in $\vec{v}_1, \ldots, \vec{v}_k$ must be a nonoverlapping subsets of the variables in $\vec{x}\vec{y}\vec{z}$. We abbreviate statements of the form (1) with $\vec{z} = \langle\rangle$ as

$$m[\vec{x}]\langle t_{init}\rangle \ \pm\texttt{=}\ t.$$

An *NC0C program* consists of a set of triggers, one for each update event $\pm R$.

Semantics. The semantics of NC0C terms is the same as in C. A statement of form (1) performs the following for each valuation θ of the variables $\vec{z}$ (extending the valuation of variables $\vec{x}\vec{y}$ passed to the trigger via its arguments) such that all map accesses in right-hand side t are defined. If $m[\vec{x}\vec{z}]$ is undefined, initialize it with t_{init}. Then, unconditionally, execute $m[\vec{x}\vec{z}]$ `+=` t. An "on $\pm R(\vec{x}\vec{y})$" trigger fires if the update is of the form $\pm R(\vec{x}\vec{y})$ and executes the statements $s_1; \ldots; s_k$ of its body sequentially (as in C).

The compilation algorithms of the next section may for convenience create multiple triggers for the same update event. However, these NC0C programs do not have cyclic dependencies between triggers: there is no trigger that reads one map that the other updates *and* vice-versa (there is a hierarchy of maps). Assuming without loss of generality that the argument variable tuples of distinct triggers for the same update event are the same, we can perform a suitable topological sort of the triggers that assures that no map is read after it is written, and concatenate their bodies according to this sort to obtain a single trigger per update event.

Example 6.1. Consider the NC0C on-insert trigger

```
on +C(cid, nation) {
  q[cid] += q1[nation];
  foreach cid2 do q[cid2] += q2[cid2, nation];
  q[cid] += 1;
  q1[nation] += 1;
  q2[cid, nation] += 1
}
```

The initializers are all 0, and are thus omitted.

The trigger `on -C` is obtained by changing `+=` to `-=` everywhere in the above trigger except for the third statement (`q[cid] += 1`), which remains unchanged.

These triggers incrementally maintain the query of Example 5.3 as the map $q[\cdot]$; the other maps are auxiliary. They are exactly the NC0C triggers that the compilation algorithm of Section 7.2 will produce (modulo the merging of multiple triggers for the same update event as described above). The ordering of statements in the triggers in not arbitrary: no map value must be read after it is written. So it is important that the first statement precedes the fourth and the second precedes the fifth.

Figure 2 shows a trace of map q as we perform a sequence of insertions and deletions. The $\Delta q[x]$ columns indicate the changes made to $q[x]$ on each update. □

Parallelization

In the following, by the *atomic values maintained* by an NC0C program, we refer to the image values (numbers) $m[\vec{a}]$ stored in the maps maintained by the program. Assume a model of computation in which additions and multiplications of pairs of numbers are performed in unit time.

Proposition 6.2. *Executing NC0C triggers takes a constant amount of work per tuple inserted or deleted and per atomic value maintained.*

Proof Sketch. The syntax of statements of form (1) is misleading in that it suggests a loop – that a nonconstant amount of work is needed to bring an atomic value up to date. Overall, a nonconstant amount of work (evidenced by the loop) is only needed because in general there are many atomic values to be maintained. The loop variables $\vec{z}$ of a statement of form (1) all occur in the lvalue $m[\vec{x}\vec{z}]$, so each such value is written only once. Because a trigger is a constant-length sequence of statements and there are no nested loops or recursion, each atomic value is only written constantly many times in the trigger overall. Since the right-hand side of a statement only performs a constant number of arithmetic operations and comparisons starting from numbers readily available in the maps (consider these to be hash-maps with constant-time access), the overall work done to update each value is a constant number of map accesses, comparisons, additions and multiplications. □

Let us formalize this result more rigidly, using circuit complexity. A *bounded fan-in circuit* is a Boolean circuit (built using AND, OR, and NOT GATES) in which AND and OR gates have only a bounded number of inputs (w.l.o.g., two). The complexity class *NC0* denotes the *LOGSPACE-uniform families* of bounded fan-in Boolean circuits of polynomial size and constant depth [22, 4]. That is, a decision problem P is in NC0 if there is a LOGSPACE algorithm that, given an input size N in unary, outputs a bounded fan-in circuit C_N of polynomial size and constant depth such that C_N outputs true for exactly those problem instances of N-bit length on which P is true.

Obviously, in an NC0 circuit, the output gate only depends on a constant number of input gates. Thus, bits of the result of arithmetics on variable-length integers cannot be computed in NC0. We will consider arithmetics modulo 2^k for a fixed integer k, which covers the real-world case of fixed-precision numbers and fixed-size registers.

Consider a single-tuple update and a representation of a materialized view and auxiliary data consisting of k-bit numbers. We will talk of a (LOGSPACE-uniform) *NC0-interpretation modulo* 2^k of a fixed NC0C update event $\pm R(\vec{t})$ if there is an NC0 circuit for each bit of the representation which, given the old version of the representation, computes the new version of that bit.

THEOREM 6.3. *Update events in NC0C programs have NC0 interpretations modulo* 2^k.

Proof Sketch. Fix an NC0C update event $\pm R(\vec{t})$. Let each map m used in the NC0C program have key domain arity $\mathrm{ar}(m)$ – that is, the keys are $\mathrm{ar}(m)$-tuples. For each map m, let $\mathrm{nst}(m)$ be the number of statements with m the left-hand-side map, from all the $\pm R$ triggers. Let $s^m_1, \ldots, s^m_{\mathrm{nst}(m)}$ be the instantiations of these statements with $\vec{t}$, obtained by substituting all occurrences of the trigger argument variables with their values in $\vec{t}$, in arbitrary order. We define an algorithm that, for an active domain size N given in unary, constructs a circuit that is an NC0-interpretation of the update event. Denote the value of $m[x_1, \ldots, x_{\mathrm{ar}(m)}]$ after adding $s^m_1 + \cdots + s^m_l$, for $0 \leq l \leq \mathrm{nst}(m)$, by $m^{(l)}[x_1, \ldots, x_{\mathrm{ar}(m)}]$. We represent the j-th bit of $m^{(l)}[x_1, \ldots, x_{\mathrm{ar}(m)}]$ by gate $G^{(l)}_{m[x_1,\ldots,x_{\mathrm{ar}(m)}]\cdot j}$ $(1 \leq j \leq k)$. The circuit will have input gates $G^{(0)}_{m[x_1,\ldots,x_{\mathrm{ar}(m)}]\cdot j}$ and output gates $G^{(\mathrm{nst}(m))}_{m[x_1,\ldots,x_{\mathrm{ar}(m)}]\cdot j}$. Since each $x_{(\cdot)}$ has N possible values, there are $\sum_{m \text{ a map}} k * N^{\mathrm{ar}(m)}$ input gates and the same number of output gates.

For each map m and each $1 \leq l \leq \mathrm{nst}(m)$, we proceed as follows. Let s^m_l be statement `foreach` $\vec{z}$ `do` $m[\vec{a}\vec{z}]$ `±=` t such that $\vec{a}$ are constants (a projection of $\vec{t}$). The definition of NC0C guarantees that no other variables than those of $\vec{z}$ occur in right-hand side t. Loop over each $\vec{x}\vec{z}$ in $N^{\mathrm{ar}(m)}$. If $\vec{x} \neq \vec{a}$, forward the value of $m^{(l-1)}[\vec{x}\vec{z}]$ to $m^{(l)}[\vec{x}\vec{z}]$ by connecting the gates. Otherwise, build a circuit for $m^{(l-1)}[\vec{x}\vec{z}] \pm t$ with the variables in t substituted, connect the inputs to the gates representing $m^{(l-1)}[\vec{x}\vec{z}]$ and $m'^{(0)}[\cdot]$ for each of the maps m' accessed in t, and connect the output to the gates representing $m^{(l)}[\vec{x}\vec{z}]$. The construction of NC0 circuits for adding and multiplying two k-bit numbers modulo 2^k (k fixed) is straightforward, and so is the wiring together of these circuits into circuits that compute fixed arithmetic expressions over k-bit numbers (using $+$, $*$, k-bit constants, and comparisons).

The algorithm needs a fixed number ($\max_m \mathrm{ar}(m)$) of registers of $\lceil \log_2 N \rceil$ bits (which is logarithmic in the input, which was N given in unary) and runs in LOGSPACE. □

The result extends to bulk updates involving a constant number of tuples by simple composition of circuits.

By fixing the update events in Theorem 6.3, we avoid the need to perform lookups of map items by "address" (key). This would require unbounded fan-in gates to encode in a constant-depth circuit. But note that this does not defeat our aim: Incremental evaluation eliminates the need to compute a sum of an unbounded number of terms, which is qualitatively different from a map lookup, which on real computers can be done in constant time (using a bus, which performs lookups but does not compute sums).

7. QUERY COMPILATION

This section describes algorithms for compiling AGCA queries to NC0C. Throughout the section, we will consider AGCA Sum terms excluding nested aggregates and inequality join conditions (i.e., involving two variables, e.g. $x < y$; non-join conditions such as $x > 5$ are permitted). These terms are called *primitive AGCA* terms.

The second requirement guarantees that NC0C initializers are constants and do not require us to go back to the database to compute. We will discuss relaxing this restriction at the end of the section.

7.1 Basic Compilation Algorithm

The following lemma allows to eliminate unneeded variables – variables that are made safe by condition atoms equating them to other safe variables – from *AGCA* queries.

LEMMA 7.1. *Given a safe term Sum*(t, ϕ) *and a set of bound variables* B*. Then there is an equivalent safe term Sum*(t', ψ) *such that each variable in* ψ *either occurs in a relational atom* $R(\vec{x})$ *of* ψ *or in* B*.*

An AGCA term $t = \mathrm{Sum}(t_0, \phi)$ is called *constraints-only* if ϕ does not contain relational atoms $R(\vec{x})$. When ϕ contains only bound variables, we can think of t as a (functional) if-statement "if ϕ then t_0 else 0" or, using C syntax, "ϕ ? t_0 : 0". Let $\mathrm{MakeC}(t, B)$ be a function that turns t into the corresponding functional if-statement after performing variable elimination using bound variables B.

We present a simple compilation algorithm for AGCA terms $\mathrm{Sum}(t, \phi)$ that do not contain nested aggregates, i.e., neither t not ϕ contain Sum terms.

THEOREM 7.2. *There is an algorithm that compiles any primitive AGCA term into an NC0C program that incrementally maintains it.*

Proof Sketch. To create on-insert ($+R$) and on-delete ($-R$) triggers that incrementally maintain map $q[\vec{b}]$ of Sum term t, we execute the following algorithm as Compile0(q, $\vec{b}$, t):

```
algorithm Compile0($q$, $\vec{b}$, $t$)
outputs an NC0C program
begin
for each relation $R$ in the schema, $\pm$ in $\{+,-\}$ do
  $\vec{a}$ := turn sch($R$) into a list of new variable names;
  $t'$ := $\Delta_{\pm R(\vec{a})} t$;
  if $t'$ is constraints-only then $t''$ := MakeC($t'$, $\vec{a}\vec{b}$)
  else $t''$ := $q_{\pm R}[\vec{a}\vec{b}]$; Compile0($q_{\pm R}$, $\vec{a}\vec{b}$, $t'$) end if;
  $t_{init}$ := $[\![t]\!]_F(\emptyset, \vec{a}\vec{b})$;
  output on $\pm R(\vec{a})$ {foreach $\vec{b}$ do $q[\vec{b}]\langle t_{init}\rangle$ += $t''$}
end
```

Here, $q_{\pm R}$ is a new map name for an auxiliary materialized view. When $\vec{b}$ is the empty tuple, we can omit `foreach` $\vec{b}$ `do` from the NC0C statement created.

The algorithm takes an aggregate query $t = \mathrm{Sum}(t_0, \phi)$ with bound variables $\vec{b}$ and defines a map $q[\vec{b}]$ for it, representing a materialized view of the query. It creates a trigger for each possible update event $\pm R$ which specifies how to update $q[\vec{b}]$ when this event occurs. To do this, it computes the delta t' of the query, and creates a new map $q_{\pm R}$ representing a materialized view of the delta. The statement increments $q[\vec{b}]$ by $q_{\pm R}[\cdot]$, and uses the result of evaluating term t on the empty database as the initializer for $q[\vec{b}]$. The function $[\![\cdot]\!]_F$ was defined in Section 4. In particular, on Sum terms t that are not constraints-only, $t_{init} = 0$. The new map $q_{\pm R}$ is incrementally maintained as well. To do this, the algorithm recursively calls itself. The algorithm terminates because by Theorem 5.5 the delta t' eventually reaches degree 0 (i.e., is constraints-only). In this case no new map is created for it but t' (turned into an NC0C term using MakeC) itself is used as the right-hand side of the NC0C statement.

This is precisely the recursive incremental view maintenance mechanism sketched in the introduction, using the notation introduced in the past sections. □

EXAMPLE 7.3. Let $q[] = \mathrm{Sum}(1, R(x) * S(x))$. Then

$$\begin{aligned}
\Delta_{\pm R(u)} q &= \pm\mathrm{Sum}(1, (x = u) * S(x)) &&=: q_R[u] \\
\Delta_{\pm S(v)} q &= \pm\mathrm{Sum}(1, R(x) * (x = v)) &&=: q_S[v] \\
\Delta_{\pm S(v)} q_R[u] &= \pm\mathrm{Sum}(1, (x = u) * (x = v)) \\
\Delta_{\pm R(u)} q_S[v] &= \pm\mathrm{Sum}(1, (x = u) * (x = v))
\end{aligned}$$

Moreover, MakeC(Sum$(1, (x = u) * (x = v))$) = `if (u=v) then 1 else 0`. We will see later that all the initializers for this code are 0. Compile0 produces the NC0C insert triggers

```
on +R(u) { q[] += qR[u] }
on +R(u) { foreach v do
              qS[v] += if (u=v) then 1 else 0 }
on +S(v) { q[] += qS[v] }
on +S(v) { foreach u do
              qR[u] += if (u=v) then 1 else 0 }
```

The delete triggers are obtained from the insert triggers by replacing all occurrences of `+` by `-`.

7.2 Eliminating Loop Variables

We now improve algorithm Compile0 from the proof of Theorem 7.2 to loop over fewer variables.

Extraction of aggregates. For a term t and its set B of bound variables, the function Extract(t, B) replaces each maximal subterm s of t that is of the form Sum$(\cdot,\cdot)$ but is not constraints-only by a map access $m[\vec{x}]$. Here m is a new name and $\vec{x}$ are those variables of B that occur in s, turned into an arbitrarily ordered tuple. The result of Extract thus is a pair (t', Θ) of the remainder term t' and a mapping Θ from map accesses $m[\vec{x}]$ to extracted subterms s (which could be used to undo the extraction). That is, t' is constraints-only, and t' with its map accesses substituted using Θ is t.

EXAMPLE 7.4. Let t be the term $\mathrm{Sum}\big(x*\mathrm{Sum}(v, R(v, z)), y = z\big) * \mathrm{Sum}\big(u, R(x, u)\big)$. Extract($t$, $\{x, y\}$) returns the pair (t', θ) consisting of term $t' = \mathrm{Sum}(x * m_1[z]\langle\theta(m_1[z])\rangle, y = z) * m_2[x]\langle\theta(m_2[x])\rangle$ and the mapping

$$\theta = \{m_1[z] \mapsto \mathrm{Sum}(v, R(v, z));\ m_2[x] \mapsto \mathrm{Sum}(u, R(x, u))\}.$$

Factorization of monomial aggregate terms. For e either a formula or a term, let vars(e) be the set of all variables occurring in e. Factorization employs the equivalence

$$\mathrm{Sum}(s * t, \phi * \psi) = \mathrm{Sum}(s, \phi) * \mathrm{Sum}(t, \psi)$$

which is true if $(\mathrm{vars}(s) \cup \mathrm{vars}(\phi)) \cap (\mathrm{vars}(t) \cup \mathrm{vars}(\psi)) = \emptyset$.

PROPOSITION 7.5. *A monomial aggregate term can be maximally factorized in linear time in its size.*

EXAMPLE 7.6. The term $\mathrm{Sum}(5 * x * \mathrm{Sum}(1, R(y, z)) * w, R(x, y) * R(v, w))$ factorizes as $\mathrm{Sum}(5, \mathit{true}) * \mathrm{Sum}(x * \mathrm{Sum}(1,\ R(y, z)), R(x, y)) * \mathrm{Sum}(w, R(v, w))$. □

Recursive factorization, given term Sum(t, ϕ), recursively – bottom-up – factorizes the aggregate terms in t before applying factorization as just described to Sum(t, ϕ) itself.

Lifting ifs. Observe that if ψ is a constraints-only term in which all variables are bound and t_0 is a term in which all variables are bound, then

$$\begin{aligned}
\mathrm{Sum}(t, \phi * \psi) &= \mathrm{Sum}(\mathrm{Sum}(t, \phi), \psi) \\
t_0 * \mathrm{Sum}(t, \psi) &= \mathrm{Sum}(t_0 * t, \psi)
\end{aligned}$$

Thus, given a recursively monomial term, we can lift ψ to the top. Let function LiftIfs$(\cdot, B)$, given bound variables B, do exactly this.

EXAMPLE 7.7. This will be used in Example 7.10:

$$\begin{aligned}
&\mathrm{LiftIfs}(\mathrm{Sum}(1, C(c_2, n) * (c_1 = c)), \{c_1, c, n\}) = \\
&\qquad\qquad \mathrm{Sum}(\mathrm{Sum}(1, C(c_2, n)), c_1 = c).
\end{aligned}$$

Further auxiliary functions. Simplify(t, B), given an aggregate term t and a set of bound variables B, (1) turns t into an equivalent sum of (inverses of) recursively monomials using Proposition 4.5, (2) recursively factorizes each of the result monomials, (3) eliminates all variables other than B, and finally (4) performs LiftIfs$(\cdot, B)$. The result $t \pm_1 t_1 \cdots \pm_n t_n$ is equivalent to t and the t_i are recursively monomials involving only variables in B.

ElimLV is a function that takes an NC0C statement

foreach $\vec{x}\vec{y}$ do $q[\vec{x}\vec{y}]$ += if $\vec{x} = \vec{z}$ then t else 0

and simplifies it to the equivalent statement `foreach` $\vec{y}$ `do` $q[\vec{z}\vec{y}]$ `+=` t. We lift ifs to be able to apply this optimization.

```
algorithm Compile($m$, $\vec{b}$, $t$)
outputs an NC0C program
begin
for each relation $R$ in the schema, $\pm_0$ in $\{+,-\}$ do
    $\vec{a}$ := turn sch($R$) into a list of new variable names;
    $t' := \Delta_{\pm_0 R(\vec{a})} t$;
    $(\pm_1 t_1 \cdots \pm_n t_n, \Theta)$ := Extract(Simplify($t'$, $\vec{a}\vec{b}$), $\vec{a}\vec{b}$);
    for each $i$ from 1 to $n$ do
        $t_{init} := [\![t]\!]_F(\emptyset, \vec{a}\vec{b})$;
        $s_i$ := (foreach $\vec{b}$ do $m[\vec{b}]\langle t_{init}\rangle$ ($\pm_i$)= MakeC($t_i$, $\vec{a}\vec{b}$));
        output on $\pm_0 R(\vec{a})$ { ElimLV($s_i$) };
    for each $(m'[\vec{x}] \mapsto t'')$ in $\Theta$ do Compile($m'$, $\vec{x}$, $t''$);
end
```

Figure 3: The algorithm Compile.

The algorithm. The algorithm Compile is given in Figure 3. It is invoked like Compile0 and closely follows its structure. The difference is that we first Simplify the delta of the query and extract the non-constraints-only aggregates. The result is a sum of constraints-only recursively monomials with map accesses. We turn each of the recursively monomials into a separate statement. The reason for this is that using LiftIfs and ElimLV, we remove loop variables, and each of the statements (monomials) may loop over a different subset of the argument variables of the map representing the query (the remaining variables are substituted by constants).

THEOREM 7.8. *Given a primitive term t and bound variables $\vec{x}$ by which results are to be grouped, the output of Compile(m, $\vec{x}$, t) is an NC0C program that correctly maintains query t in map $m[\vec{x}]$ under inserts and deletes.*

EXAMPLE 7.9. Algorithm Compile simplifies the two foreach-loop statements of Example 7.3 to `qS[u] += 1` and `qR[v] +=1`. The resulting triggers for this example have no loops and run in *constant sequential time*. □

EXAMPLE 7.10. Consider the query $q[c_1]$ of Example 5.3 and the sum of (inverses of) recursively monomials equivalent to $\Delta_{\pm C(c,n)} q[c_1]$ computed there. Factorization on this query is the identity. Eliminating variables according to Lemma 7.1 with variables $\{c_1, c, n\}$ bound yields $\pm t_1 \pm t_2 + t_3$ with $t_1 = \text{Sum}(1, (c_1 = c) * C(c_2, n))$, $t_2 = \text{Sum}(1, C(c_1, n))$, and $t_3 = \text{Sum}(1, c_1 = c)$. The result of if-lifting for t_1 is shown in Example 7.7. For t_2 and t_3 it is the identity.

$$\text{Extract}(\text{Simplify}(\Delta_{\pm C(c,n)} q[c_1], \{c_1, c, n\}), \{c_1, c, n\})$$

yields $(\pm t_1' \pm t_2' + t_3', \Theta)$ where $t_1' = \text{Sum}(\text{q1}[n], c_1 = c)$, $t_2' = \text{q2}[c_1, n]$, $t_3' = t_3$, and

$$\Theta = \left\{ \begin{array}{rcl} \text{q1}[n] & \mapsto & \text{Sum}(1, C(c_2, n)) \\ \text{q2}[c_1, n] & \mapsto & \text{Sum}(1, C(c_1, n)). \end{array} \right\}$$

Using ElimLV, we get the three trigger statements

$$q[c] \mathrel{\texttt{±=}} \text{q1}[n]; \qquad \texttt{foreach } c_1 \texttt{ do } q[c_1] \mathrel{\texttt{±=}} \text{q2}[c_1, n]; \qquad q[c] \mathrel{\texttt{+=}} 1.$$

Without ElimLV, the first statement would be

`foreach` c_1 `do` $q[c_1]$ `±= if` $c_1 = c$ `then q1`$[n]$ `else` 0

and the third would be more complicated, too. We further have to compile q1 and q2. Since

$$\Delta_{\pm C(c',n')} \text{q1}[n] = \Delta_{\pm C(c',n')} \text{q2}[c, n] = \pm 1,$$

the compiled NC0C program is exactly as shown in Example 6.1.

7.3 Initializers

We defined the primitive (=compilable) AGCA terms to be those without nested aggregates or inequality join conditions. The second requirement of that definition can be replaced by the requirement to exclude terms that are unsafe if the set of bound variables is set to $\emptyset$ or where this condition can become true for a k-th delta.

For example, the query $q[] = \text{Sum}(1, R(x) * S(y) * (x < y))$ is excluded because its delta $m[y] = \text{Sum}(1, R(x) * (x < y))$ is safe for bound variable y, but unsafe for the empty set of bound variables. The problem is that on an insertion into R, we do not know for which y values the map m should be updated – the query does not provide use with a method of bounding the domain of y.

However, if we assume a global, immutable active domain given (cf. the proof of Theorem 6.3), we never have to compute initial values, and the compilation algorithms are applicable to all Sum terms with simple conditions only (i.e., for which taking deltas simplifies the query structure).

For primitive queries, it is true that for the initialization of a map value $m[\vec{a}]\langle t_{init}\rangle$, we can evaluate t_{init} on the empty database; any contents of the database are not visible to query t_{init} or would otherwise have caused initialization of $m[\vec{a}]$ earlier. This can be proved by induction.

8. RELATED WORK

There is a large literature on the incremental view maintenance problem (e.g. [7, 33, 5, 32, 8, 17, 18, 16, 36, 11, 13, 9, 26, 27, 23]). 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 $\mathbb{Z}_{\text{Rel}}$ 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 [17, 18], 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, 15, 14]. The goals of this paper are different from those of [25, 15, 14]. $\mathbb{Z}$-relations [14] 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 [15, 14] 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 [10, 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, 11, 26, 27, 21, 20, 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 non-incrementally. 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 [20]. 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.

Acknowledgments

This paper creates a foundation for a current project on practical query compilation at Cornell, DBToaster. A predecessor of the compilation algorithm of Figure 3 was developed jointly with Yanif Ahmad and is described in [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.

9. 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] D. A. M. Barrington, N. Immerman, and H. Straubing. On uniformity within NC^1. *J. Comput. Syst. Sci.*, 41(3):274–306, 1990.

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

[6] R. P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21:201–206, 1974.

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

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

[9] 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.

[10] 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.

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

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

[13] 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.

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

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

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

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

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

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

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

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

[22] 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.

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

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

[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] A. Rajaraman, Y. Sagiv, and J. D. Ullman. Answering queries using templates with binding patterns. In *Proc. PODS*, pages 105–112, 1995.

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

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

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

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

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