The VLDB Journal (2014) 23:253–278
DOI 10.1007/s00778-013-0348-4

SPECIAL ISSUE PAPER

DBToaster: higher-order delta processing for dynamic, frequently fresh views

Christoph Koch · Yanif Ahmad · Oliver Kennedy · Milos Nikolic · Andres Nötzli · Daniel Lupei · Amir Shaikhha

Received: 11 February 2013 / Revised: 14 November 2013 / Accepted: 20 November 2013 / Published online: 9 January 2014

Abstract Applications ranging from algorithmic trading to scientific data analysis require real-time analytics based on views over databases receiving thousands of updates each second. Such views have to be kept fresh at millisecond latencies. At the same time, these views have to support classical SQL, rather than window semantics, to enable applications that combine current with aged or historical data. In this article, we present the DBToaster system, which keeps materialized views of standard SQL queries continuously fresh as data changes very rapidly. This is achieved by a combination of aggressive compilation techniques and DBToaster's original recursive finite differencing technique which materializes a query and a set of its higher-order deltas as views. These views support each other's incremental maintenance, leading to a reduced overall view maintenance cost. DBToaster supports tens of thousands of complete view refreshes per second for a wide range of queries.

Keywords Database queries · Materialized views · Incremental view maintenance · Compilation

C. Koch (✉) · M. Nikolic · A. Nötzli · D. Lupei · A. Shaikhha
École Polytechnique Fédérale de Lausanne (EPFL) IC DATA, Station 14, 1015 Lausanne, Switzerland
e-mail: christoph.koch@epfl.ch

Y. Ahmad
The Johns Hopkins University, 3400 North Charles Street, Baltimore, MD 21218, USA
e-mail: yanif@jhu.edu

O. Kennedy
SUNY Buffalo, 338 Davis Hall, Buffalo, NY 14260, USA
e-mail: okennedy@buffalo.edu

1 Introduction

Data analytics has been dominated by after-the-fact exploration in classical data warehouses for decades. This is now beginning to change: Today, businesses, engineers, and scientists are increasingly placing data analytics engines earlier in their workflows to react to signals in fresh data. These dynamic datasets exhibit a wide range of update rates, volumes, anomalies, and trends. Responsive analytics is an essential component of computing in finance, telecommunications, intelligence, and critical infrastructure management and is gaining adoption in operations, logistics, scientific computing, and Web and social media analysis.

Developing suitable responsive analytics engines is challenging. The combination of frequent updates, long-running queries and a large stateful working set precludes the exclusive use of OLAP, OLTP, or stream processors. Furthermore, query requirements on updates cannot be serviced by the functionality and semantics of any one single system, from Complex Event Processing (CEP) engines to active databases, and database views.

Consider the example of algorithmic trading (see [20] for an application overview). Here, strategy designers want to use analytics—expressible in a declarative language like SQL—on order book data in their algorithms. Order books consist of the orders waiting to be executed at a stock exchange. These order books change very frequently. However, some orders may stay in the order book relatively long before they are executed or revoked, precluding the use of stream engines with window semantics. Applications such as scientific simulations and intelligence analysis also involve tracking entities of interest over widely ranging time periods, resulting in large stateful and dynamic computation.

The DBToaster project [3,21,22] builds and studies data management systems designed for large datasets that evolve

rapidly through high-rate update streams. We aim to combine the advantages of DBMSes (rich queries over recent and historical data, without restrictive window semantics) and CEP engines (low latency and high view refresh rates).

The technical focus of this article is on an extreme form of incremental view maintenance (IVM) that we call *higher-order*[1] IVM. We make use of discrete forward differences (delta queries) recursively, on multiple levels of derivation. That is, we use delta queries ("first-order deltas") to incrementally maintain the view of the input query, then materialize the delta queries as views too. We maintain these views using delta queries to the delta queries ("second-order deltas"), use third-order delta queries to incrementally maintain the second-order views, and so on. Our use of higher-order deltas is quite different from earlier work on choosing which query *subexpressions* to materialize and incrementally maintain for best performance [36]. Our technique for constructing higher-order deltas is closer in spirit to discrete wavelet and numerical differentiation methods, and we use a superficial analogy to the Haar wavelet transform as the motivation for calling the base technique a *viewlet transform*.

Example 1 Consider a query Q that counts the number of tuples in the product of relations R and S. For now, we only want to maintain the view of Q under insertions. Denote by Δ_R (resp. Δ_S) the change to a view as one tuple is inserted into R (resp. S). Suppose we materialize the four views:

- Q (0-th order),
- $\Delta_R Q = \text{count}(S)$ and $\Delta_S Q = \text{count}(R)$ (first order), and
- $\Delta_R \Delta_S Q = \Delta_S \Delta_R Q = 1$ (second order, a "delta of a delta query").

We can simultaneously maintain all these views based on each other using exclusively summation and avoiding the computation of any products. The fourth view is constant and independent of the database. Each of the other views is refreshed when a tuple is inserted by adding the appropriate delta view. For instance, as a tuple is inserted into R, we add $\Delta_R Q$ to Q and $\Delta_R \Delta_S Q$ to $\Delta_S Q$. (No change is required to $\Delta_R Q$, since $\Delta_R \Delta_R Q = 0$.) Suppose R contains 2 tuples and S contains 3 tuples. The table below presents the sequence of states of the materialized views when performing several insertions into R and S. When we add a tuple to S, we increment Q by 2 ($\Delta_S Q$) to obtain 8 and $\Delta_R Q$ by 1 ($\Delta_S \Delta_R Q$) to get 4. If we subsequently insert a tuple into R, we increment Q by 4 ($\Delta_R Q$) to 12 and $\Delta_S Q$ by 1, to 3. A similar process applies for the next two insertions of S tuples.

Again, the main benefit of using the auxiliary views is that we can avoid computing the product $R \times S$ (or in general, joins) by simply summing up views. In this example, the view values of the $(k+1)$-th row can be computed by just three pairwise additions of the values from the k-th row.

[1] In the sense of higher-order derivative, not higher-order function.

time point	insert into	$\|\|R\|\|$	$\|\|S\|\|$	Q	$\Delta_R Q$	$\Delta_S Q$	$\Delta_R \Delta_S Q$, $\Delta_S \Delta_R Q$
0	–	2	3	6	3	2	1
1	S	2	4	8	4	2	1
2	R	3	4	12	4	3	1
3	S	3	5	15	5	3	1
4	S	3	6	18	6	3	1

The above example shows the simplest query for which the viewlet transform includes a second-order delta query, omitting any complex query features (e.g., predicates, self-joins, nested queries). Viewlet transforms can handle general update workloads including deletions and updates, as well as queries with multi-row results.

For a large fragment of SQL, *higher-order IVM avoids join processing*, reducing all the view refreshment work to summation. Joins are only needed for view definitions that include inequality joins or nested aggregates. The viewlet transform repeatedly (recursively) performs delta rewrites. In the absence of nested aggregates, each k-th order delta is structurally simpler than the $(k-1)$-th order delta query. The viewlet transform terminates, as for some n, the n-th order delta is always constant, depending only on the update but not on the database. In the above example, the second-order delta is constant, independent of any database relation.

Our higher-order IVM framework, *DBToaster*, realizes as-incremental-as-possible query evaluation over SQL with a query language extending the bag relational algebra, query compilation, and a variety of novel materialization and optimization strategies. DBToaster bears the promise of providing materialized views of long-running SQL queries, without window semantics or other restrictions, at very high view refresh rates. The data may change rapidly, and still part of it may be long-lived. DBToaster can use this functionality as the basis for richer query constructs than those supported by stream engines. DBToaster takes SQL view queries as input and automatically incrementalizes them into C++ or Scala trigger code where all work reduces to fine-grained, low-cost updates of materialized views.

We have developed and released DBToaster as a query compiler [14]. The compiler produces dedicated binaries (or source code) that implement long-running query engines for SQL views. The resulting source code or binary can be embedded in client applications, or can operate as a stand-alone system that consumes data from files or sockets. We present a system and architecture overview in [21].

Example 2 Consider the query

```
Q = SELECT SUM(LI.PRICE * O.XCH)
FROM Orders O, Lineitem LI
WHERE O.ORDK = LI.ORDK;
```

on a TPC-H like schema of Orders and Lineitem where line items have prices and orders have currency exchange rates. The query asks for total sales across all orders weighted by exchange rates. We materialize the views for query Q as well as the first-order views Q_{LI}, representing $\Delta_{LI} Q$, and Q_O, representing $\Delta_O Q$. The second-order deltas are constant with respect to the database and are inlined in the following insert trigger programs for query Q.

on insert into O **values** $(ordk, custk, xch)$:

$Q \mathrel{+}= xch * Q_O[ordk]$

$Q_{LI}[ordk] \mathrel{+}= xch$

on insert into LI **values** $(ordk, ptk, price)$:

$Q \mathrel{+}= price * Q_{LI}[ordk]$

$Q_O[ordk] \mathrel{+}= price$

The query result is again scalar, but the auxiliary views are not. Our language generalizes these views from SQL's multi-sets to maps that associate multiplicities with tuples. This is again a very simple example (more involved ones are presented throughout the article), but it illustrates something notable: while classical IVM has to evaluate the first-order deltas, which takes linear time in this example (e.g., $\Delta_O Q$[ordk] is `SELECT SUM(LI.PRICE) FROM Lineitem LI WHERE LI.ORDK`=ordk), we sidestep this by performing IVM on the deltas. This way our triggers can be evaluated in *constant* time for single-tuple inserts in this example. The delete triggers for Q are the same as the insert triggers with += replaced by −= everywhere.

This example presents single-tuple update triggers. The viewlet transform is not limited to this and supports bulk updates. However, delta queries for single-tuple updates have further optimization potential, which the DBToaster compiler leverages to create very efficient code that refreshes views whenever a new update arrives. By not queuing updates for bulk processing, DBToaster maximizes view availability and minimizes view refresh latency, enabling ultra-low latency monitoring applications.

On paper, higher-order IVM clearly dominates classical IVM. If classical IVM is a good idea, then doing it recursively is an even better idea. The same efficiency improvement argument in favor of IVM of the base query also holds for IVM of the delta query. Considering that joins are expensive and this approach eliminates them, higher-order IVM has the potential for excellent query performance.

In practice, how well do our expectations of higher-order IVM translate into real performance gains? A priori, the costs associated with storing and managing auxiliary materialized views for higher-order delta queries might be more considerable than expected. This article presents the lessons learned in an effort to realize higher-order IVM and to understand its strengths and drawbacks. Our contributions are:

1. We present the concept of higher-order IVM and describe the viewlet transform. This part of the article generalizes and consolidates our earlier work [3,22].
2. There are cases (inequality joins and certain nesting patterns) when a naive viewlet transform is too aggressive, and certain parts of queries are better re-evaluated than incrementally maintained. We develop heuristic rules for trading off between materialization and lazy evaluation for the best performance.
3. We have built the DBToaster system, which implements higher-order IVM. It combines an optimizing compiler that creates efficient update triggers, based on the techniques discussed above, with a runtime system to keep views continuously fresh as updates stream in at high rates. (The runtime system is currently single-core and main-memory based, but this is not an intrinsic limitation of our method. In fact, our trigger programs are particularly parallel-friendly [22]. See [21] for a more detailed system description.) We have also made our system implementation publicly available [14].
4. We present the first set of extensive experimental results on higher-order IVM obtained using DBToaster. Our experiments indicate that our compilation approach dominates the state of the art, often by multiple orders of magnitude. This is particularly the case for queries consisting of many joins or nested aggregations. On a workload of automated trading, scientific, and ETL queries, we show that current systems cannot sustain fresh views at the rates required in algorithmic trading and real-time analytics, while higher-order IVM takes a big step toward making these applications viable.

Most of our benchmark queries contain features like nested subqueries that no commercial IVM implementation supports, while our approach handles them all.

2 Related work

2.1 A brief survey of IVM techniques

Database view management is a well-studied area with over three decades of supporting literature. A recent survey of the topic can be found in [10]. We focus on the aspects of view materialization most pertinent to the DBToaster project. Our work innovates on the high-order aspect to IVM, which is orthogonal to all previous work.

Incremental view maintenance algorithms and formal semantics Maintaining query answers has been considered under both the set [6,7] and bag [9,16] relational algebra. Generally, given a query on N relations $Q(R_1, \ldots, R_N)$, classical IVM uses a first-order delta query $\Delta_{R_1} Q = Q(R_1 \cup \Delta R_1, R_2, \ldots R_N) - Q(R_1, \ldots, R_N)$ for each input relation

R_i in turn. The creation of delta queries has been studied for query languages with aggregation [34] and bag semantics [16], but we know of no work to formally examine delta queries of nested and correlated subqueries. Kawaguchi et al. [19] has considered view maintenance in the nested relational algebra (NRA), however this has not been widely adopted in any commercial DBMS. Finally, Yang et al. [44] considered temporal views, and [28] outer joins and nulls, all for flat SPJAG queries without generalizing to subqueries, the full compositionality of SQL, or the range of standard aggregates.

Materialization and query optimization strategies Selecting queries to materialize and reuse during processing has spanned fine-grained approaches from subqueries [36] and partial materialization [25,37], to coarse-grained methods as part of multiquery optimization and common subexpressions [18,46]. Picking views from a workload of queries typically uses the AND-OR graph representation from multiquery optimization [18,36], or adopts signature and subsumption methods for common subexpressions [46]. Ross et al. [36] selects sets of subqueries of view definitions for materialization. The outcome is related to higher-order IVM, as the delta of a simple query is frequently a subquery. Naturally, for such queries, both approaches select the same (optimal) materialization strategy. However, our delta operation also captures nonlinearities in the deltas of more complex queries (e.g., self-joins) and produces different materialization strategies. We also use query rewriting strategies that deal with correlated subqueries, inspired by work on query decorrelation such as [39]. Our experiments include results for a DBMS that uses a similar nested subquery materialization strategy. Additionally, our work builds on higher-order IVM, extending it into a complete query compilation system.

Physical DB designers [2,47] often use the query optimizer as a subcomponent to manage the search space of equivalent views, reusing its rewriting and pruning mechanisms. For partial materialization methods, ViewCache [37] and DynaMat [25] use materialized view fragments, the former materializing join results by storing pointers back to input tuples, the latter subject to a caching policy based on refresh time and cache space overhead constraints.

Evaluation strategies For efficient maintenance with first-order delta queries, Colby et al. [11] and Zhou et al. [45] study eager and lazy evaluation to balance query and update workloads, Salem et al. [38] argues for asynchronous view maintenance, and [12] investigates the interaction of different view freshness models. In addition, evaluating maintenance queries has been studied extensively in Datalog with semi-naive evaluation (which also uses first-order deltas) and DRed (delete-rederive) [17]. Finally, Ghanem et al. [15] argues for view maintenance in stream processing, which reinforces our position of using IVM as a general-purpose change-propagation mechanism for collections, on top of which window and pattern constructs can be defined.

2.2 Update processing mechanisms

Triggers and active databases Triggers, active databases and event-condition-action (ECA) mechanisms [4] provide general-purpose reactive behavior in a DBMS. The literature considers recursive and cascading trigger firings and restrictions to ensure restricted propagation. Trigger-based approaches require developers to manually convert queries to delta form, a painful and error-prone process especially in the higher-order setting. Without manual incrementalization, triggers suffer from poor performance and cannot be optimized by a DBMS when written in C or Java.

Data stream processing Data stream processing [1,31] and streaming algorithms combine two aspects of handling updates: (1) shared, incremental processing (e.g., paired vs paned windows, sliding windows) and (2) sublinear algorithms (with polylogarithmic space bounds). The latter are approximate processing techniques that are difficult to compose and have had limited adoption in commercial DBMS. Advanced processing techniques in the streaming community also focus mostly on approximate techniques when processing cannot keep up with stream rates (e.g., load shedding, prioritization [41]), on shared processing (e.g., on-the-fly aggregation [27]), or specialized algorithms and data structures [13]. Our approach to streaming is about generalizing incremental processing to (non-windowed) SQL semantics (including nested subqueries and aggregates). Of course, windows can be expressed in this semantics if desired. Similar principles are discussed in [15].

Automatic differentiation and incrementalization, and applications Beyond the database literature, the programming language literature has studied automatic incrementalization [29] and automatic differentiation. Automatic incrementalization is by no means a solved challenge, especially when considering general recursion and unbounded iteration. Automatic differentiation considers deltas of functions applied over scalars rather than sets or collections, and lately in higher-order fashion [35]. Bridging these two areas of research would be fruitful for supporting UDFs and general computation on scalars and collections in DBToaster.

3 Queries and deltas

In this section, we present the internal data model, *generalized multiset relations* (GMRs), and the query language, *AGgregate CAlculus* (AGCA), of DBToaster and show how to compute delta queries. The design of the data model and query language avoids complex case distinctions when processing different forms of updates (specifically, deletions)

during IVM. The language is algebraic in flavor, is expressive (it captures most of SQL), and has few syntactic cases, which facilitates the construction of powerful optimizers and compilers.

Generalized multiset relations (GMRs) generalize multiset relations (as in SQL) to collections of tuples, each annotated with tuples of rational multiplicities (i.e., from $\mathbb{Q}$). This allows us to treat databases and updates uniformly; for instance, a deletion is a relation with negative multiplicities, and applying an update to a database means unioning/adding it to the database. It also allows us to use multiplicities to represent aggregate query results (which do not need to be integers). As a consequence, when performing delta processing on aggregate queries, growing an aggregate means to add to the aggregate value rather than to delete the tuple with the old aggregate value and insert a tuple with the new aggregate value. Maintaining aggregates in the multiplicities allows for simpler and cleaner bookkeeping, and having multiple "multiplicities" for a tuple allows for multiple aggregates and bookkeeping attributes to be maintained together in a single GMR.

AGgregate CAlculus is a very simple language, essentially constructed from GMRs and infinite interpreted relations (which capture conditions) using just four operations—addition, its inverse, multiplication, and sum-aggregation. This syntactic simplicity facilitates rich optimizations in the DBToaster compiler. For the purpose of understanding the delta processing framework (and proving it correct), one can view the query language as a polynomial ring over GMRs with an addition operation that at once generalizes multiset union (as known from SQL) and updating, and a multiplication operation that generalizes the natural join operation. This ring-theoretic framework was initially developed in [22] and has been refined in [23].

The multiplication operation also implements sideways binding passing and enforces range restriction as known in the context of relational calculus. This allows the language to be algebraic without a need for an explicit selection operation. AGCA encodes selection as a multiplication of a query with a condition just like relational calculus does. Multiplication is defined in such a way that query results are guaranteed to be always finite.

3.1 Data model

It is convenient to model tuples as having their own schema; this way we can use the same definition for varying environments. Formally, we define a relation tuple $\vec{t}$ as a partial function from a vocabulary of column names to data values. We write $\vec{t}$ as $\langle A : v \mid A \in \mathrm{dom}(\vec{t})\rangle$, where v is a value from the codomain which we may also identify as $\vec{t}(A)$; $\langle\rangle$ signifies the empty tuple. The set of all tuples is denoted by $\mathbb{T}$.

A *generalized multiset relation (GMR)* $R : \mathbb{T} \to \mathbb{Q}^k$ is a function from relation tuples to tuples of rational numbers such that $R(\vec{t}) \neq \langle 0\rangle^k$ for at most a finite number of tuples $\vec{t}$. A GMR succinctly encodes k relations, indicating the *multiplicity* with which each tuple of $\mathbb{T}$ occurs in each relation. The set of all such functions is denoted by $\mathbb{Q}^k_{\mathrm{Rel}}$. We write $\mathrm{sch}(R)$ to denote a common schema of GMR R, which subsumes the tuple schemas. Below, we also use classical singleton relations (without multiplicities) and the natural join operator $\bowtie$. We write $\{\vec{t}\}$ to construct a singleton relation from tuple $\vec{t}$ with the schema $\mathrm{sch}(\{\vec{t}\}) = \mathrm{dom}(\vec{t})$. For tuples $\vec{s}, \vec{t}$ that are consistent ($\{\vec{s}\} \bowtie \{\vec{t}\} \neq \emptyset$), we can write $\vec{s}\vec{t}$ for the consistent concatenation ($\{\vec{s}\vec{t}\} = \{\vec{s}\} \bowtie \{\vec{t}\}$).

3.2 Query language

We now formally define AGCA over $\mathbb{Q}^2_{\mathrm{Rel}}$. In the following semantics, the first field of the multiplicity tuple is used for bookkeeping purposes to track the bag multiplicity of the tuple, while the second field stores the aggregate value being computed. AGCA may be generalized from $\mathbb{Q}^2_{\mathrm{Rel}}$ to $\mathbb{Q}^k_{\mathrm{Rel}}$ for any $k > 2$ by cloning its behavior with respect to the "value" field. This generalization is omitted to avoid notation clutter.

Syntax AGgregate CAlculus (AGCA) expressions are built from constants, variables, relational atoms, conditions, and variable assignments ($:=$), using operations bag union $+$, natural join $*$, and aggregate sum $\mathrm{Sum}_{\vec{A}}$. The abstract syntax is:

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

Here x denotes variables (which we also call columns), $\vec{t}$ tuples of variables, $\vec{A}$ tuples of group-by variables, R relation names, c constants from $\mathbb{Q}$, and θ denotes comparison operators ($=$, $\neq$, $>$, $\geq$, $<$, and $\leq$). We also use $x\,\theta\,y$ as syntactic sugar for $(x - y)\,\theta\,0$.

Note that $x := q$ is a special condition essentially equivalent to $x = q$, with one catch. 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). To make this information flow explicit, we create a syntactic distinction between the case where only one of the variables is safe from the left ($:=$) and the case where both are safe ($=$).

Informal semantics Intuitively, the following steps lead to the definition of AGCA:

1. Take the fragment of relational algebra on multiset relations with just the operations of selection σ, natural join $\bowtie$, union $+$, and (multiplicity-preserving) projection π. Allow queries of this algebra to be nested into selection conditions; projections to nullary relations yield a

numerical multiplicity—a tuple count—that can be compared with numerical database fields (e.g., we can write $\sigma_{A<\pi_{()}(R)}(S)$, when S has a numerical column A). Allow deletions to be expressed using a negative multiplicity (necessitating an additive inverse operation).

2. Promote selection conditions to interpreted relations and enforce a range restriction policy to ensure that all queries define finite relations, both as in relational calculus. That is, we may write $R \bowtie (A < B)$ for $\sigma_{A<B}(R)$ and have no further need for an explicit selection operation. However, $(A < B)$ by itself is not a valid query because an unbounded number of tuples satisfy the condition.
   This simplifies the algebra: of the four remaining operations, we will see that we can treat union, the additive inverse, and projection alike in delta processing, reducing the number of operations to two: union and join.
3. Generalize this language to GMRs while preserving distributivity of $+$ and $\bowtie$. This turns GMRs with these two operations and the explicit additive inverse into a ring.
   There is essentially only one way to do this, as shown in [23], and this solution is the semantics of AGCA. This ring structure makes delta processing extremely simple.
4. Implement an operation $x := Q$ to lift multiplicities to tuple values. This is a powerful and subtle operation which requires GMRs of type $\mathbb{Q}^2_{\text{Rel}}$ to maintain, separately, aggregates and true multiplicities.

With the above intuition in mind, the reader should be able to validate that the formal semantics presented in Fig. 1 is a solution to the specification just given. Note that in these semantics, the generalized union, join, and projection operations are denoted by $+$, $*$, and $\text{Sum}_{\vec{A}}$.

Semantics The formal semantics of AGCA is given by an evaluation function $[\![\cdot]\!]$ that, for a query Q, a database $\mathbf{D}$, and a *context*—a tuple $\vec{b}$ of "bound variables"—evaluates to an element $[\![Q]\!](\mathbf{D}, \vec{b})$ of $\mathbb{Q}^2_{\text{Rel}}$ constructed as follows. Note that in several the recursive cases, arithmetic on the multiplicity/value tuples is performed vector-wise. AGgregate CAlculus admits sideways information passing. That is, query expressions are evaluated relative to a given *context* $\vec{b}$—an association of variables and their values—which is provided from the outside. The language, specifically the multiplication operation, dictates how such bindings are to be passed to the right during query evaluation.

The definition of $[\![R(\vec{x})]\!]$ allows column renaming. The evaluation of variables x (e.g., $[\![x]\!]$) *fails* if they are unbound at evaluation time. We consider a query in which this may happen illegal and exclude such queries from AGCA. Observe that $R - S = R + (-S)$ does not refer to the difference operation of relational algebra, but to the additive inverse for GMRs: for instance, $\emptyset - R = -R$ in AGCA ($\emptyset$ can be written in AGCA as the constant 0), 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*.

Base cases

Constant Value

$$[\![c]\!](\cdot,\cdot) := \vec{t} \mapsto \begin{cases} \langle 1, c\rangle & .. \ \vec{t} = \langle\rangle \\ \langle 0, 0\rangle & .. \text{ otherwise} \end{cases}$$

Variable Value

$$[\![x]\!](\cdot, \vec{b}) := \vec{t} \mapsto \begin{cases} \textbf{fail} & .. \ x \notin \text{dom}(\vec{b}) \\ \langle 1, \vec{b}(x)\rangle & .. \text{ otherwise, if } \vec{t} = \langle\rangle \\ \langle 0, 0\rangle & .. \text{ otherwise} \end{cases}$$

Relation

$$[\![R(\vec{x})]\!](\mathbf{D}, \vec{b}) := \vec{t} \mapsto \begin{cases} \langle m, m\rangle & .. \ m = R^{\mathbf{D}}(\langle A_i : \vec{t}(x_i) \mid A_i \in \text{sch}(R)\rangle), \\ & \quad \{\vec{b}\} \bowtie \{\vec{t}\} \neq \emptyset, |\text{dom}(\vec{t})| = |\text{sch}(R)| \\ \langle 0, 0\rangle & .. \text{ otherwise} \end{cases}$$

Comparison

$$[\![x\,\theta\,0]\!](\mathbf{D}, \vec{b}) := \vec{t} \mapsto \begin{cases} \textbf{fail} & .. \ x \notin \text{dom}(\vec{b}) \\ \langle 1, 1\rangle & .. \ \vec{b}(x)\,\theta\,0, \ \vec{t} = \langle\rangle \\ \langle 0, 0\rangle & .. \text{ otherwise} \end{cases}$$

Recursive cases

Bag Union

$$[\![Q_1 + Q_2]\!](\mathbf{D}, \vec{b}) := \vec{t} \mapsto [\![Q_1]\!](\mathbf{D}, \vec{b})(\vec{t}) +^{\mathbb{Q}^2} [\![Q_2]\!](\mathbf{D}, \vec{b})(\vec{t})$$

Additive Inverse

$$[\![-Q]\!](\mathbf{D}, \vec{b}) := \vec{t} \mapsto [\![Q]\!](\mathbf{D}, \vec{b})(\vec{t}) *^{\mathbb{Q}^2} \langle -1, -1\rangle$$

Natural Join

$$[\![Q_1 * Q_2]\!](\mathbf{D}, \vec{b}) := \vec{t} \mapsto \sum^{\mathbb{Q}^2}_{\substack{\{\vec{t}\}=\{\vec{r}\}\bowtie\{\vec{s}\} \\ \{\vec{b}\}\bowtie\{\vec{r}\}\neq\emptyset}} [\![Q_1]\!](\mathbf{D}, \vec{b})(\vec{r}) *^{\mathbb{Q}^2} [\![Q_2]\!](\mathbf{D}, \vec{b}\vec{r})(\vec{s})$$

Summation with Group-by

$$[\![\text{Sum}_{\vec{A}}\, Q]\!](\mathbf{D}, \vec{b}) := \vec{t} \mapsto \begin{cases} \sum^{\mathbb{Q}^2}_{\{\vec{t}\}\bowtie\{\vec{s}\}=\{\vec{s}\}} [\![Q]\!](\mathbf{D}, \vec{b})(\vec{s}) & .. \ \text{dom}(\vec{t}) = \vec{A}, \\ & \quad \{\vec{b}\} \bowtie \{\vec{t}\} \neq \emptyset \\ \langle 0, 0\rangle & .. \text{ otherwise} \end{cases}$$

Variable Assignment

$$[\![x := Q]\!](\mathbf{D}, \vec{b}) := \vec{t} \mapsto \begin{cases} \langle 1, 1\rangle & .. \ \vec{t}_2 = \langle x_i : t(x_i) \mid x_i \neq x\rangle, \exists m.\ m \neq 0, \\ & \quad \langle m, \vec{t}(x)\rangle = [\![Q]\!](\mathbf{D}, \vec{b})(\vec{t}_2) \\ \langle 0, 0\rangle & .. \text{ otherwise} \end{cases}$$

Fig. 1 The formal evaluation semantics of AGCA ($[\![\cdot]\!]$). The operators $+^{\mathbb{Q}^2}$, $*^{\mathbb{Q}^2}$ and $\sum^{\mathbb{Q}^2}$ are vector-wise instances of $+$, $*$ and $\sum$ respectively

Example 3 Let R be a GMR of $\mathbb{Q}^2_{\text{Rel}}$

$R^{\mathbf{D}}$	A	B	
	1	2	$\mapsto \langle m_1, q_1\rangle$
	3	5	$\mapsto \langle m_2, q_2\rangle$
	4	2	$\mapsto \langle m_3, q_3\rangle$

where m_i, q_i denote rational multiplicities. Then

$[\![R(x,y)]\!](\mathbf{D}, \langle x:3\rangle)$	x	y	
	3	5	$\mapsto \langle m_2, q_2\rangle$

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

The AGCA version of the query $\sigma_{A<B}(R)$ evaluates to

$[\![R(x,y)*(x<y)]\!](\mathbf{D}, \langle\rangle)$	x	y	
	1	2	$\mapsto \langle m_1, q_1\rangle$
	3	5	$\mapsto \langle m_2, q_2\rangle$

For instance,

$$
\begin{aligned}
&[\![R(x,y)*(x<y)]\!](\mathbf{D},\langle\rangle)(\langle x:1, y:2\rangle)\\
&= \sum_{\{\langle x:1,y:2\rangle\}=\{\vec{r}\}\bowtie\{\vec{s}\}}^{\mathbb{Q}^2} [\![R(x,y)]\!](\mathbf{D},\langle\rangle)(\vec{r}) * [\![x<y]\!](\mathbf{D},\vec{r})(\vec{s})\\
&= [\![R(x,y)]\!](\mathbf{D},\langle\rangle)(\langle x:1, y:2\rangle)\\
&\quad *^{\mathbb{Q}^2} [\![x<y]\!](\mathbf{D},\langle x:1, y:2\rangle)(\langle\rangle) = \langle m_1, q_1\rangle
\end{aligned}
$$

Sum aggregates serve as multiplicity-preserving projections: The result of $\mathrm{Sum}_{\vec{A}} Q$ is the tuples of the projection of Q on $\vec{A}$, and each tuple's multiplicity is the sum of the multiplicities of the tuples that were projected down to it. An aggregation $\mathrm{Sum}_{\vec{A}} R$ almost works like the SQL query `SELECT` $\vec{A}$, `SUM(1) FROM R GROUP BY` $\vec{A}$. The only difference is that SQL puts the aggregate values into a new column, while $\mathrm{Sum}_{\vec{A}} R$ puts them into the multiplicity of the group-by tuples. We can express more general aggregate summations using clever arithmetics on multiplicities.

Example 4 The sum-aggregate query $\mathrm{Sum}_{[y]}(R(x,y) * 2 * x)$ generalizes the SQL query `SELECT B, SUM(2 * A) FROM R GROUP BY B` to GMRs. Applied to the GMR of Example 3, it yields

$[\![\mathrm{Sum}_{[y]}(R(x,y)*2*x)]\!](\mathbf{D},\langle\rangle)$	y	
	2	$\mapsto \langle m_1+m_3, 2q_1+8q_3\rangle$
	5	$\mapsto \langle m_2, 6q_2\rangle$

For instance,

$$
\begin{aligned}
&[\![\mathrm{Sum}_{[y]}(R(x,y)*2*x)]\!](\mathbf{D},\langle\rangle)(\langle y:2\rangle)\\
&= \sum_{\vec{r},\vec{s},\vec{t}}^{\mathbb{Q}^2} [\![R(x,y)]\!](\mathbf{D},\langle\rangle)(\vec{r}) \;*^{\mathbb{Q}^2}\; [\![2]\!](\mathbf{D},\vec{r})(\vec{s})\\
&*^{\mathbb{Q}^2} [\![x]\!](\mathbf{D},\vec{r}\vec{s})(\vec{t})\\
&= \langle m_1 * 1, q_1 * 2\rangle *^{\mathbb{Q}^2} [\![x]\!](\mathbf{D},\langle x:1, y:2\rangle)(\langle\rangle)\\
&+^{\mathbb{Q}^2} \langle m_3 * 1, q_3 * 2\rangle *^{\mathbb{Q}^2} [\![x]\!](\mathbf{D},\langle x:4, y:2\rangle)(\langle\rangle)\\
&= \langle m_1 + m_3, 2 * q_1 + 8 * q_3\rangle
\end{aligned}
$$

Using the assignment operator, variables can also take on values of non-grouping aggregates, or those that evaluate to a single value for a given set of bindings. That way we can express queries with nested aggregates. Nested aggregates may be correlated with the outside as usual in SQL.

Example 5 Assume relation R has columns (A, B) and relation S has columns (C, D). The SQL query

```
SELECT * FROM R  WHERE B
< (SELECT SUM(D) FROM S WHERE A > C)
```

is equivalent to $\mathrm{Sum}_{[A,B]}\big(R(A,B) * (z := Q_n) * (B < z)\big)$ with $Q_n = \mathrm{Sum}_{[\,]}(S(C,D) * (A > C) * D)$.

AGgregate CAlculus has no explicit syntax for universal quantification or aggregates other than Sum, but these features can be expressed using (nested) sum-aggregate queries. Special handling of these features in delta processing and query optimization could yield performance better than what we report in our experiments. However, granting these definable features specialized treatment is beyond the scope of this article. As a consequence, our implementation provides native support for only the fragment presented above, and the experiments use only techniques described in the article. This language specification covers all of the core features of SQL with the exception of null values and outer joins.

3.3 Computing the delta of a query

Next, we show how to construct delta queries. AGCA has the nice property of being *closed under taking deltas*: For each query expression Q, there is an expression ΔQ *of the same language* that captures how the result of Q changes as the database $\mathbf{D}$ is changed by update workload $\Delta\mathbf{D}$,

$$\Delta Q(\mathbf{D}, \Delta\mathbf{D}) := Q(\mathbf{D} + \Delta\mathbf{D}) - Q(\mathbf{D}).$$

Due to the strong compositionality of the language, we can turn any AGCA expression into its delta by repeatedly applying the following rules syntactically to expressions until we obtain an AGCA expression over GMRs and delta GMRs (updates). We write u to denote an update, and $\Delta_u Q$ for the delta of expression Q with respect to that update. Thus for a GMR R, $\Delta_u R$ is the change to R made in update u.

$$
\begin{aligned}
\Delta_u(Q_1 + Q_2) &:= (\Delta_u Q_1) + (\Delta_u Q_2)\\
\Delta_u(Q_1 * Q_2) &:= ((\Delta_u Q_1) * Q_2) + (Q_1 * (\Delta_u Q_2))\\
&\quad + ((\Delta_u Q_1) * (\Delta_u Q_2))\\
\Delta_u - Q &:= -\Delta_u Q\\
\Delta_u c &:= 0\\
\Delta_u x &:= 0\\
\Delta_u(x\,\theta\,0) &:= 0\\
\Delta_u(x := Q) &:= (x := (Q + \Delta_u Q)) - (x := Q)\\
\Delta_u(\mathrm{Sum}_{\vec{A}} Q) &:= \mathrm{Sum}_{\vec{A}}(\Delta_u Q)
\end{aligned}
$$

The correctness of the rules follows from the fact that the GMRs with $+$ and $*$ form a ring (for example, the delta rule for $*$ is a direct consequence of distributivity) and that $\mathrm{Sum}_{\vec{A}}$ can be thought of as the repeated application of $+$ [23].

The special case of single-tuple updates is interesting since it allows us to simplify delta queries further and to generate particularly efficient view refresh code. We write $\pm R(\vec{t})$ to denote the insertion/deletion of a tuple $\vec{t}$ into/from relation R of the database.

$$\Delta_{\pm R(\vec{t})}\big(R(x_1,\ldots,x_{|\mathrm{sch}(R)|})\big) := \pm \prod_{i=1}^{|\mathrm{sch}(R)|} (x_i := t_i)$$

$$\Delta_{\pm R(\vec{t})}\big(S(x_1,\ldots,x_{|\mathrm{sch}(S)|})\big) := 0 \qquad (R \neq S)$$

Example 6 Consider the AGCA query

$$Q = \mathrm{Sum}_{[\,]}\big(R(A,B) * S(C,D) * (B = C) * A * D\big)$$

which is equivalent to the query of Example 2. We abbreviate the Sum subexpression as Q_1. Let us study the insertion/deletion of a single tuple $\langle A : x, B : y\rangle$ to/from R. Since

$$\Delta_{\pm R(x,y)} R(A,B) = \pm(A := x) * (B := y)$$

by the delta rule for $*$,

$$\Delta_{\pm R(x,y)} Q_1 = \pm(A := x) * (B := y) * S(C,D) * (B = C) * A * D = \pm S(C,D) * (y = C) * x * D$$

For the main query, we get $\Delta_{\pm R(x,y)} Q = \mathrm{Sum}_{[\,]}(\Delta_{\pm R(x,y)} Q_1)$.

Example 7 Consider the overall query with a nested aggregate from Example 5. The delta for insertion/deletion of a tuple $\langle C : x, D : y\rangle$ to/from relation S is:

$$\mathrm{Sum}_{[A,B]}\big(R(A,B) * \Delta_{\pm S(x,y)}(z := Qn) * (B < z)\big)$$

Following the delta rule for $:=$,

$$\Delta_{\pm S(x,y)}(z := Qn) = \big(z := (Q_n \pm \Delta_{\pm S(x,y)} Q_n)\big) - (z := Qn)$$

where

$$\Delta_{\pm S(x,y)} Q_n = \mathrm{Sum}_{[\,]}\big((C := x) * (D := y) * (A > C) * D\big) = (A > x) * y$$

which is a way of writing "if $(A > x)$ then y else 0".

3.4 Binding patterns

AGgregate CAlculus queries have binding patterns which represent information flow. In general, this flow is not exclusively bottom-up. *Input variables* are parameters whose values cannot be computed from the database and without which we cannot evaluate the query. *Output variables* represent columns of the query result schema.

The most interesting case of input variables occurs in a correlated nested subquery, viewed in isolation. In such a subquery, a correlation variable from the outside is such an input variable. The subquery can only be computed if a value for the input variable is given.

Example 8 In Example 5, all columns of R's schema are output variables. In the subexpression Q_n, A is an input variable and there are no output variables since the aggregate is non-grouping.

Taking a delta *adds input variables*, parameterizing the query with the update. In Example 6, the delta query uses input variables x and y to pass the update.

In $\Delta_{\pm S(x,y)} Q_n = (A > x) * y$ of Example 7, A, x, and y are input variables.

4 The viewlet transform

We are now ready for the viewlet transform. In this section, we exclude variable assignments $x := Q$ from the query language where Q contains a sum aggregate. This restriction is eliminated in the next section. This query language fragment has the following nice property: ΔQ is structurally strictly simpler than Q when query complexity is measured as follows. For union-free queries, the *degree* $\deg(Q)$ of query Q is the number of relations joined together. Distributivity allows pushing unions above joins and thus gives a degree to queries with unions: the maximum degree of the union-free subqueries. Queries are strongly analogous to polynomials, and the degree of queries is defined precisely as it is defined for polynomials (where the relation atoms of the query correspond to the variables of the polynomial).

Theorem 1 ([23]) *If* $\deg(Q) > 0$ *then* $\deg(\Delta Q) = \deg(Q) - 1$.

The viewlet transform uses the simple fact that a delta query is a query too. Thus it can be incrementally maintained using the delta query of the delta query, which again can be materialized and incrementally maintained, and so on, recursively. By the above theorem, this recursive query transformation terminates at the $\deg(Q)$-th recursion level, when the obtained delta query is a "constant" (degree 0) independent of the database and dependent only on updates.

In the following, we write $\Delta^l Q[u_1,\ldots,u_l]$ $(l \geq 0)$ to denote a view representing the query $\Delta_{u_l} \cdots \Delta_{u_1} Q$ parameterized by updates $u_1,\ldots,u_l$. In general, this is a higher-order delta query, but the case $l = 0$ is simply the query Q. Just like in Sect. 3.3, an update is a database of GMRs, potentially holding inserts and deletes (represented by positive and negative tuple multiplicities) for any relation of the database being updated.

Definition 1 Given a query Q, the viewlet transform turns Q into the following update trigger, which maintains the view of Q plus a set of auxiliary views.

on update u **do** {
  for $k = 0$ **to** $\deg(Q) - 1$ **do**
    foreach $u_1, \ldots, u_k$ **do** $\Delta^k Q[u_1, \ldots, u_k]$ +=
      $\begin{cases} \Delta_u \Delta_{u_k} \cdots \Delta_{u_1} Q & \ldots\ k = \deg(Q) - 1 \\ \Delta^{k+1} Q[u_1, \ldots, u_k, u] & \ldots \text{ otherwise} \end{cases}$
}

The viewlet transform owes its name to a superficial analogy with the Haar wavelet transform, which also materializes a hierarchy of differences.

At runtime, each trigger statement loops over the domains of the variables $u_1, \ldots, u_k$ (in each case, the domain of databases). This is of course not practical.

One way to turn the viewlet transform into a practical scheme is to restrict the updates to be constant-size batches. Without true loss of generality, we will look at single-tuple updates. These offer particular optimization potential, and we focus on these in the remainder of this article. This requires, however, to create multiple triggers to handle the different update events (inserts and deletes for multiple relations).

$+R(\vec{t})$ denotes the insertion of a single tuple $\vec{t}$ into relation R, while $-R(\vec{t})$ denotes the corresponding deletion. We create insert and delete triggers in which the argument is the tuple, rather than a GMR, and thus avoid looping over relation-typed variables. Using only field variables is one key to efficient triggers; the other is the potential to rewrite and simplify queries. We give an example and study this in detail in the next section.

Example 9 Consider query Q from Example 6, with single-tuple updates. This query has degree two. Let $\text{sgn}_R, \text{sgn}_S \in \{+, -\}$. Then one of the second-order deltas is

$$(\Delta_{\text{sgn}_R R(x,y)} \Delta_{\text{sgn}_S S(z,u)} Q)[x, y, z, u] = \text{sgn}_R \text{sgn}_S (y = z) * x * u$$

A trigger for events $\pm R(x, y)$ can be obtained as follows. Variables x and y are arguments of the trigger and are bound at runtime, but variables z and u need to be looped over. On the other hand, the right-hand side of the trigger is only non-zero in case that $y = z$. So we can substitute z by y everywhere and eliminate z. Using this simplification, the viewlet transform produces the following trigger for $+R(x, y)$:

Q += $(\Delta_{+R(x,y)} Q)[x, y]$;
foreach u **do** $\Delta_{+S(y,u)} Q[y, u]$ += $x * u$;
foreach u **do** $\Delta_{-S(y,u)} Q[y, u]$ -= $x * u$;

The construction of the remaining triggers happens analogously. The trigger contains an update rule for the (in this case, scalar) view Q for the overall query result. The rule uses the auxiliary view $\Delta_{\pm R(x,y)} Q$, which is maintained in the update triggers for S. The trigger also contains update rules for the auxiliary views $\Delta_{\pm S(y,u)} Q$ that are used to update Q in the update triggers for S.

The reason why we omitted deltas $\Delta_{\pm R(\ldots)} \Delta_{\pm R(\ldots)} Q$ and $\Delta_{\pm S(\ldots)} \Delta_{\pm S(\ldots)} Q$ is that these are guaranteed to be 0, as the query does not have a self-join.

An additional optimization presented in the next section eliminates the loops on u using distributivity and associativity, leading to the triggers of Example 2.

We observe that the structure of the work that needs to be done is extremely regular and (conceptually) simple. Moreover, there are no classical coarse-grained query operators left, so it makes no sense to give this workload to a classical query optimizer. There are for-loops over many variables, which have the potential to be very expensive. But the work is also perfectly data-parallel, and there are no data dependencies comparable to those present in joins. All this provides justification for making heavy use of compilation.

We refer to the viewlet transform as presented in this section as the naive viewlet transform. The following section presents improvements and optimizations.

5 Higher-order IVM

The DBToaster compilation algorithm is a practical implementation of the viewlet transform, which we call *higher-order IVM*. Like the viewlet transform, higher-order IVM transforms a query Q into a *trigger program*—a set of triggers that maintain the materialized view (as a *map* or *dictionary*) M_Q on query Q, and a set of supplemental materialized views. As before, each trigger consists of *update statements*, each of the form **foreach** $\vec{x}$ **do** $M_Q[\vec{x}]$ += $Q'[\vec{x}]$;

The key observation behind higher-order IVM is that full delta queries materialized by the naive viewlet transform may be very expensive, or simply impossible to maintain.

An example of the former situation is a delta query including a Cartesian product (i.e., a product of two subqueries $Q_1 * Q_2$ with no output variables in common). As we will soon see, such queries arise quite frequently in the viewlet transform and are expensive to maintain.

Full delta queries that are impossible to maintain include (1) Delta queries that contain input variables and therefore lack finite support, and (2) Deltas of queries with nested subqueries, to which Theorem 1 does not apply.

The key insight behind DBToaster, which makes higher-order IVM possible, is that it is unnecessary to materialize the full delta query. When generating update statements for a materialized view Q, DBToaster materializes the delta terms $\Delta_u Q$ as one or more subqueries of each delta query. When the corresponding update statement is executed, $\Delta_u Q$ is computed from the materialized subqueries.

Materializing subqueries of the delta query instead of the full delta query increases the cost of evaluating trig-

ger statements. By carefully selecting an appropriate set of subqueries, however, the increased *execution cost* is offset by a substantial reduction in view *maintenance costs*.

We now formally define higher-order IVM (HO-IVM). Recall that the viewlet transform produces a sequence of statements, each of the form: $Q[\vec{x}]$ += $\Delta_u Q[\vec{x}]$. Unlike the viewlet transform, HO-IVM does not materialize $\Delta_u Q$ as a single materialized view. Instead, HO-IVM identifies a set of subqueries $\vec{M}_{\Delta_u Q}$ to materialize and rewrites the statement into an equivalent statement $Q[\vec{x}]$ += $\Delta_u Q'[\vec{x}]$, evaluated over these materialized views. Then, instead of recurring on $\Delta_u Q$ as in the viewlet transform, HO-IVM recurs individually on each $M_i \in \vec{M}_{\Delta_u Q}$. We refer to the rewritten query and the set of materialized subqueries as a *materialization decision* for $\Delta_u Q$, denoted $\langle \Delta_u Q', \vec{M}_{\Delta_u Q} \rangle$.

Example 10 Consider the following query

$$Q[\,] = \mathrm{Sum}_{[\,]}(R(A, B) * S(B, C) * T(C, D))$$

The insertion trigger for $+S(b, c)$ includes the statement

$$Q[\,] \mathrel{+}= \mathrm{Sum}_{[\,]}(R(A, b) * T(c, D))\,;$$

Under the viewlet transform, we materialize and maintain the expression $\mathrm{Sum}_{[b,c]}(R(A, b) * T(c, D))$. DBToaster materializes that expression in terms of two sub-expressions:

$$M_1[b] := \mathrm{Sum}_{[b]}(R(A, b)) \qquad M_2[c] := \mathrm{Sum}_{[c]}(T(c, D))$$

The insertion trigger then includes the statement:

$$Q[\,] \mathrel{+}= \mathrm{Sum}_{[\,]}(M_1[b] * M_2[c]);$$

Algorithm 1 summarizes HO-IVM and Sects. 5.1–5.3 discuss heuristics for obtaining a materialization decision (which define the `materialize()` procedure).

Algorithm 1 HO-IVM(Q, M_Q)

Require: A query Q to be maintained as M_Q.
Ensure: A list of update statements T_u for each update event u.
  for all Relation Name R used in Q **do**
    for all $u \in \{+R, -R\}$ **do**
      let $\vec{x}$ = the input/output variables of $\Delta_u Q$
      let $\langle Q', \{M_i := Q_i\} \rangle$ = `materialize`($\Delta_u Q$)
      update $T_u = T_u$: :(**foreach** $\vec{x}$ **do** $M_Q[\vec{x}]$ += $Q'[\vec{x}]$)
      for all i **do** `HO-IVM`(Q_i, M_i)

Note that the use of materialization decisions need not be limited to *delta* queries. At the user's request, DBToaster can materialize the user-provided (i.e., top-level) query piecewise as opposed to maintaining a single view with the full result that the user is interested in. Although doing so creates a computational overhead when the view contents are accessed, it can substantially reduce maintenance costs. Computing averages is a common example where piecewise materialization is beneficial, as it involves maintaining two separate, simpler aggregates: the count and sum of the input. The average value can be easily reconstituted from the sum and count when it is necessary. This generalized form of higher-order IVM is made explicit in Algorithm 2.

Algorithm 2 Generalized HO-IVM(Q)

Require: A query Q to be maintained.
Ensure: A query Q', equivalent to Q.
Ensure: A list of update statements T_u for each update event u.
  let $\langle Q', \{M_i := Q_i\} \rangle$ = `materialize`(Q)
  for all i **do** `HO-IVM`(Q_i, M_i)

5.1 Heuristic optimization

We present our approach to selecting a materialization decision as a set of independent heuristic rewrite rules that are repeatedly applied to the naive materialization decision $\langle (M_{Q,1}), \{M_{Q,1} := Q\} \rangle$, up to a fixed point.

For clarity, we present these rules in terms of a materialization operator $\mathscr{M}$. For example, one possible materialization decision for the query $Q := Q_1 * Q_2$ is

$$\mathscr{M}(Q_1) * \mathscr{M}(Q_2) \equiv \langle (M_{Q,1} * M_{Q,2}), \{M_{Q,i} := Q_i\} \rangle.$$

All but the trivial rewrite rules are presented in Fig. 2. The full array of heuristic optimizations is discussed in depth below. Figure 3 shows how these rules apply to the experimental workload discussed in Sect. 9.

Duplicate view elimination As the simplest optimization, we observe that the viewlet transform produces many duplicate views. This is primarily because the delta operation typically commutes with itself; $\Delta_R \Delta_S Q = \Delta_S \Delta_R Q$ for any Q without nested aggregates over R or S. Structural equivalence on the materialized view queries is typically sufficient to identify this type of view duplication. View deduplication substantially reduces the number of views created.

Query decomposition In most relational optimizers, the generalized distributive law [5] plays an important role in widening the search space for optimal materialization decisions. It is also significant in DBToaster's heuristic optimization strategy. Queries with disconnected join graphs are particularly expensive to materialize. The query decomposition rewrite rule presented in Fig. 2.1 exploits the generalized distributive law to break up such queries into smaller components for materialization.

If the join graph of Q includes multiple disconnected components Q_1, Q_2, ...(i.e., Q is the Cartesian product $Q_1 \times Q_2 \times \ldots$), it is better to materialize each component independently as $\mathscr{M}(Q_1) * \mathscr{M}(Q_2) * \ldots$.

The cost of selecting from (iterating over) $\mathscr{M}(Q)$ is similar to the cost of selecting from $\mathscr{M}(Q_1) * \mathscr{M}(Q_2) * \ldots$, as both require an iteration over $|Q_1| \times |Q_2| \times \ldots$ elements. Furthermore, maintaining each individual Q_i is less compu-

Query Decomposition

$$\mathcal{M}(\mathrm{Sum}_{\vec{A}\vec{B}}(Q_1 * Q_2)) \Rightarrow \mathcal{M}(\mathrm{Sum}_{\vec{A}}(Q_1)) * \mathcal{M}(\mathrm{Sum}_{\vec{B}}(Q_2)) \quad (1)$$

$\vec{A}$ and $\vec{B}$ are any disjoint sets of variables.

Factorization and Polynomial Expansion

$$\mathcal{M}(\mathrm{Sum}_{\vec{A}}(Q_L * (Q_1 + Q_2 + \ldots) * Q_R)) \Leftrightarrow$$
$$\mathcal{M}(\mathrm{Sum}_{\vec{A}}(Q_L * Q_1 * Q_R)) + \mathcal{M}(\mathrm{Sum}_{\vec{A}}(Q_L * Q_2 * Q_R)) + \ldots \quad (2)$$

Input Variables

$$\mathcal{M}(\mathrm{Sum}_{\vec{A}}(Q * f(\vec{B}\vec{C}))) \Rightarrow \mathrm{Sum}_{\vec{A}}(\mathcal{M}(\mathrm{Sum}_{\vec{A}\vec{B}}\, Q) * f(\vec{B}\vec{C})) \quad (3)$$

Q is the maximal subquery that contains no input variables.
f is a subquery that contains no relation terms
$\vec{A}$ is any set of variables
$\vec{B}$ is the set of output variables of Q referenced by f
$\vec{C}$ is the set of input variables referenced by f

Nested Aggregates and Decorrelation

$$\mathcal{M}(\mathrm{Sum}_{\vec{A}}(Q_O * (x := Q_N) * f(x, \vec{B}))) \Rightarrow$$
$$\mathrm{Sum}_{\vec{A}}(\mathcal{M}(\mathrm{Sum}_{\vec{A}\vec{B}}(Q_O)) * (x := \mathcal{M}(Q_N)) * f(x, \vec{B})) \quad (4)$$

Q_O is the maximal subquery for which x is not an *input* variable.
Q_N is a subquery containing at least one relation term.
f is a subquery containing no relation terms.
$\vec{A}$ is any set of variables
$\vec{B}$ is the set of output variables of Q_O referenced by f or Q_N

Fig. 2 Rewrite rules for partial materialization. *Bidirectional arrows* indicate rules that are applied heuristically from *left to right* while materializing an expression, but applied in reverse to some expressions. Note that for any query Q with output variables $\vec{A}$, the property $Q = \mathrm{Sum}_{\vec{A}}(Q)$ holds

	Query	Features				Rules			
		T, J	Wc	Gb	Nst.	D	P	I S/C	N R/I
TPC-H	Q1	1	$<$	✓	-	-	✓	S	I
	Q2	5,=	$\wedge, =$	-	1	✓	✓	S	I
	Q3	3,=	$\wedge, <$	✓	-	✓	✓	-	-
	Q4	1	$\wedge, <$	✓	1	-	✓	S	I
	Q5	6,=	$\wedge, <$	✓	-	✓	✓	-	-
	Q6	1	$\wedge, <$	-	-	-	-	-	-
	Q7	6,=	$\wedge, \vee, <$	✓	1	✓	✓	-	-
	Q8	7,=	$\wedge, =, <$	✓	1	✓	✓	S	R
	Q9	6,=	$\wedge, =$	✓	1	✓	✓	-	-
	Q10	4,=	$\wedge, =, <$	✓	-	✓	✓	-	-
	Q11	2,=	-	✓	-	✓	✓	S	I
	Q11a	2,=	-	✓	-	-	-	-	-
	Q12	2,=	$\wedge, =, <$	✓	-	-	✓	-	-
	Q13	2,=	$\neq$	✓	1	-	✓	S	I
	Q14	2,=	$\wedge, <$	-	-	-	✓	S	R
	Q15	2,=	$\wedge, <$	✓	2	-	✓	S	I
	Q16	2,=	$\vee, =, \neq$	✓	1	✓	✓	S	R
	Q17	2,=	$<$	-	1	✓	✓	S	I
	Q17a	2,=	$<$	-	1	✓	✓	S	I
	Q18	3,=	$<$	✓	2	✓	-	S	R,I
	Q18a	3,=	$<$	✓	2	✓	✓	S	I
	Q19	2,=	$\vee, =, <$	-	-	-	✓	-	-
	Q20	2,=	$\wedge, =, <$	-	2	-	✓	S	I
	Q21	4,=	$\wedge, =, <$	✓	1	✓	✓	S	I
	Q22	1	$=, <$	✓	1	-	✓	S	R,I
	Q22a	1	$=, <$	✓	1	-	✓	S	R,I
	SSB4	7,=	$<$	✓	-	✓	-	-	-
Finance	AXF	2,=	$\vee, <$	✓	-	-	✓	S	-
	BSP	2,=	$\wedge, <$	✓	-	-	✓	-	-
	BSV	2,=	-	-	-	-	✓	-	-
	MST	2,x	$\wedge, <$	✓	1	-	✓	S	R,I
	PSP	2,x	$\wedge, <$	-	1	-	✓	S	R,I
	VWAP	1	$<$	-	1	-	✓	C	R
Sci.	MDDB1	4,=	$\wedge, =$	✓	-	✓	✓	-	-
	MDDB2	10,=	$\wedge, \vee, <$	-	-	✓	✓	-	-

Fig. 3 Workload features and rewrite rules applied to each query. Features notation: **T**, **J** = Number of tables, Join type (=: equi, x: cross); **Wc** = Where-clause ($\wedge$: conjunction, $\vee$: disjunction, $=$: equality, $\neq$: inequality, $<$: range inequality); **Gb** = GroupBy-clause; **Nst.** = Nesting depth. Rules notation: **D** = query decomposition; **P** = factorization and polynomial expansion; **I** = input variables, with a subquery, **S**, or a view cache, **C**; **N** = nested aggregates and decorrelation, with complete re-evaluation of the nested query, **R**, or incremental evaluation, **I**

tationally expensive: the decomposed materialization stores (and maintains) only $|Q_1| + |Q_2| + \ldots$ values, while the combined materialization handles $|Q_1| * |Q_2| * \ldots$ values.

Taking a delta of a query with respect to a single-tuple update replaces a relation in the query by a singleton constant tuple, effectively eliminating one hyperedge from the join graph and creating new disconnected components that can be further decomposed. Consequently, this optimization plays a major role in the efficiency of DBToaster and for ensuring that the number of maps created for any acyclic query is polynomial. Example 10 shows this optimization.

Polynomial expansion and factorization As described, the query decomposition optimization operates exclusively over conjunctive queries (i.e., AGCA expressions without addition). To support decomposition across unions, we observe that addition and aggregate summations commute:

$$\mathrm{Sum}_{\vec{A}}(Q_1 + Q_2) = \mathrm{Sum}_{\vec{A}}(Q_1) + \mathrm{Sum}_{\vec{A}}(Q_2)$$

and that the generalized distributive law [5] applies:

$$Q_1 * (Q_2 + Q_3) = (Q_1 * Q_2) + (Q_1 * Q_3)$$

Consequently, any query can be expanded into a sum of multiplicative clauses, where each clause is a conjunctive query (analogous to a query in disjunctive normal form).

As part of the heuristic-based materialization strategy, queries are fully expanded so that each multiplicative clause may be materialized independently. This in turn makes it possible to effectively perform query decomposition. The rewrite rule for this process, which we refer to as polynomial expansion, is presented in Fig. 2.2.

Note that this rule can also be applied in reverse. If a common term (Q_L and Q_R in the rewrite rule) appears in several multiplicative clauses, the term can be *factored* out of the sum of these multiplicative clauses for an equivalent, smaller and cheaper query expression. It is often possible (and beneficial) to factorize the rewritten query Q' after obtaining a final materialization decision $\langle Q', \{\ldots\}\rangle$, and the expression is no longer required to be in factored form.

Input variables The delta operation introduces input variables, which in turn makes it possible to create delta queries without finite support. For example, consider the query

$$Q[A, B, C] = R(A, B) * S(C) * (B < C) * A$$

The delta $\Delta_{+R(x,y)} Q[x, y, C] = S(C) * (y < C) * x$ has two input variables (x, y), making it impossible to fully materialize it.

A trivial solution to this problem is to simply avoid materializing terms that contain input variables. The rewrite rule shown in Fig. 2.3 uses the generalized distributive law [5] to do precisely this, by pulling terminals that contain input variables out of the materialization operator. Note that as with the query decomposition operator, this rewrite rule relies on polynomial expansion to simplify the expression into a sum of multiplicative clauses.

In addition to extracting input variables from the materialized query, this rewrite rule also pushes summation into the materialized expression.[2] This is analogous to a common optimization in view maintenance and query processing, where aggregation and projection operators are pushed down as far as possible into the evaluation pipeline.

In addition to the trivial solution, several other strategies for dealing with input variables are possible. For queries where the input variables appear in comparison predicates (e.g., $S(C) * (y < C)$), data structure-based solutions like range indices are possible, but beyond the scope of this work. A third strategy based on caching is discussed below.

Deltas of nested aggregates AGCA encodes nested subqueries using the assignment operator $(:=)$. Recall that the delta rule for this operator is

$$\Delta_u (x := Q) := (x := Q + \Delta_u Q) - (x := Q)$$

The delta query references the original query (twice) and is clearly not simpler than the original query (as per Theorem 1). On such expressions, the (naive) viewlet transform fails to terminate.

Of course, queries with assignment are not always catastrophic. If $\Delta_u Q = 0$, then

$$(x := Q + \Delta_u Q) - (x := Q) = (x := Q) - (x := Q) = 0$$

For assignments where the query Q being assigned to x corresponds to a simple arithmetic expression, the delta is always empty. However, if Q contains a relation term $R(\vec{A})$ (i.e., Q represents a nested subquery), then the deltas $\Delta_{\pm R}$ must be handled as a special case.

The rewrite rule presented in Fig. 2.4 identifies nested subqueries and uses the generalized distributive law [5] to extract them for independent materialization. Thus, only expressions without nested subqueries are materialized, and higher-order IVM terminates.

As with rules 1 and 3, this rule relies on polynomial expansion to simplify the expression into a sum of multiplicative clauses. Furthermore, just like the input-variable rewrite rule, this rule aggressively pushes aggregates down into the newly created expressions.

Although this rule is necessary to guarantee compiler termination, it can introduce unnecessary overheads into the evaluation of delta queries. When naively used, this rule might separately materialize a nested subquery that does not reference the delta relation. A refinement of this optimization analyzes a given delta query before applying the rewrite rule: Considering the expression from Fig. 2.4 and update u to relation R, it is only necessary to apply the rewrite rule to Q_N when Q_N includes a reference to R. If it does not, then $\Delta_u Q_N = 0$, and the rewrite is unnecessary to ensure termination of higher-order IVM.

Example 11 Recall the delta query $\Delta_{\pm S(x,y)} Q$ of Example 7.

$$\begin{aligned}\mathrm{Sum}_{[A,B]} (&R(A, B) * (z := (Q_n \pm (A > x) * y)) * (B < z) \\ &- R(A, B) * (z := Q_n) * (B < z))\end{aligned}$$

where $Q_n = \mathrm{Sum}_{[\,]} (S(C, D) * (A > C) * D)$. The materialization decision for this delta query materializes two subqueries: $M_{Q,1} := R(A, B)$ and $M_{Q,2} := \mathrm{Sum}_{[C]}(S(C, D) * D)$. On every update of relation S, the delta evaluation effectively evaluates the outer query twice: once using the new value $Q_n \pm \Delta Q_n$, and once using the old value of Q_n. Conversely, the delta for updates to R, $\Delta_{\pm R} Q$ always has a lower degree than Q and is materialized as a single map (the nested-aggregates rewrite rule is ignored).

In this example, we see one additional possibility for optimization of nested aggregates. The delta for insertions into S is actually *more* expensive than re-evaluating the entire update (The outer query is evaluated twice in the delta, but just once in the original).

Thus, in some situations DBToaster produces an update statement that replaces the map being maintained, instead of updating it. As a general rule, the incremental approach pays off when the inner query is correlated on an equality, and the delta's arguments bind at least one of these variables; then the delta query only aggregates over a subset of the tuples in the outer query. For instance, if the nested query from Example 7 were to have $(A = C)$ instead of $(A > C)$, then only a subset of the aggregated tuples would have been affected by the delta, leading to the incremental approach as a better choice. Based on this analysis, the heuristic optimizer decides whether to re-evaluate or incrementally maintain any given delta query.

Note that although Q is being recomputed, we can still accelerate the computation by materializing Q piecewise. Although the expression being materialized is not a delta, we still compute a materialization decision (as in generalized higher-order IVM).

[2] Note that the operation need not actually have a sum aggregate. An expression Q with output variables $\vec{A}$ is equivalent to the expression $\mathrm{Sum}_{\vec{A}}(Q)$.

Because we are already materializing the expression Q, care must be taken to avoid creating a self-referential loop in this materialization decision. The default materialization decision $\mathscr{M}(Q)$ is meaningless, as Q defines the view being maintained. We avoid this by first applying the nested-query rewrite heuristic as aggressively as possible to eliminate all nested subqueries in the expressions being materialized. Because recomputation is only appropriate for queries with nested subqueries (otherwise it is better to perform IVM), the resulting expression is guaranteed to be simpler than Q.

5.2 Specialized data structures

Thus far we have considered only straightforward view materialization, where views are stored in map-like data structures. But for some queries, advanced data storage primitives can provide opportunities to materialize more complex expressions—particularly those involving input variables. As many of these opportunities are data-dependent, DBToaster's heuristic optimizations rely on user-input to direct selection of an appropriate data structure. In practice, a cost-based tuning advisor could also be used to automate the selection process with minimal user involvement.

As one example of a specialized data structure, we discuss a data structure capable of materializing expressions with arbitrary input variables: *view caches*. This data structure is analogous to partially materialized views [25,37].

A view cache materializes AGCA expressions with input variables. The cache stores multiple full copies of the materialized view, each for a different valuation of the input variable(s) that appear in the cache's defining expression.

When a lookup is performed on the cache, these input variables must be bound to specific values. If the cache contains a materialized view for that particular valuation, the materialized view is returned as a normal map. Otherwise, the cache's defining query is evaluated as normal, and the result is stored in the cache.

Unlike a traditional cache, the contents of a view cache are not invalidated when the underlying data changes. Instead, whenever the data is updated, each materialized view stored in the view cache is updated as normal.

View caches are only beneficial when a small working set for the domain of an input variable can be expected. Otherwise the heuristic optimizer refrains from creating them.

5.3 Simplifying delta expressions

Although the delta operation reduces the expression degree, it tends to make the expression itself longer and more complicated. It introduces input variables into the expression. For products and some conditions, it creates additional additive terms.

Example 12 Consider the following expression:

$$Q[A, B] = R(A) * R(A) * S(B)$$

Applying the delta rules leaves us with the expression

$$\begin{aligned}\Delta_{+R(x)} Q[A, B] = &\big((A := x) * R(A)\\ &+ R(A) * (A := x) + (A := x) * (A := x)\big) * S(B)\\ &+ R(A) * R(A) * 0 + \big((A := x) * R(A)\\ &+ R(A) * (A := x) + (A := x) * (A := x)\big) * 0\end{aligned}$$

This expression is quite complex, but can be simplified to $\Delta_{+R(x)} Q[x, B] = (2 * R(x) + 1) * S(B)$

This added complexity increases both compilation and evaluation costs. Therefore, as part of higher-order IVM, we regularly apply several simplifying transformations to AGCA expressions; some of these correspond to common relational algebra transformations. Like with the heuristic rules, these simplifications are applied repeatedly, up to a fixed point.

Unification We propagate range restrictions through AGCA expressions by a two-step process. First, we transform equality predicates into equivalent assignment expressions. Second, we propagate assignments through the expression, eliminating them if appropriate.

In the first stage, we identify equality comparison terms that can be rewritten into an assignment-compatible form, where a single variable appears on the left-hand side of the expression. Each such equality comparison is commuted left through product terms and out of Sum operators until either (1) the left-hand variable falls out of scope, (2) commuting it further would cause a variable appearing on the right-hand side to fall out of scope. If condition 1 is satisfied, we convert the equality into an assignment.

An equality comparison may have multiple assignment-compatible rewritings. At most one of these rewritings can possibly satisfy condition 1, as a second variable falling out of scope would violate condition 2. When multiple rewritings are available, we continue commuting until condition 1 is satisfied for precisely 1 rewriting, or until condition 2 is violated for all rewritings.

Once all equality comparisons have been converted into assignments, we propagate assignments throughout the expression. This is analogous to beta reduction in lambda calculus, although there are semantic restrictions on AGCA expressions that can prevent us from fully reducing the assignment. There are three such limitations: (1) AGCA forbids range restrictions to be incorporated directly into relation terms, (2) AGCA disallows computationally intensive (i.e., nested aggregate) expressions to be incorporated directly into comparison operations, and (3) If the assignment creates a range restriction on the domain of the query output, the assignment must remain in the expression.

The assignment is propagated as aggressively as possible. If none of the above limitations are violated and the variable is not in the schema of the query, then the variable is no longer used and the assignment can be safely removed.

Partial evaluation and algebraic identities Delta derivation frequently produces expressions containing sums of terms that differ only by a constant multiplier. The polynomial factorization heuristic presented in Sect. 5.1 is applied (ignoring the materialization operator) to group and sum up the constant multipliers.

During this optimization stage, AGCA expressions are partially evaluated by merging constant values that appear in a sum or product together and by applying the standard algebraic identities $Q+0=Q$, $Q*1=Q$, and $Q*0=0$. This last identity is especially useful during delta computation, as the delta operation produces many zeroes, as well as expressions of the form $Q-Q$.

Extracting range restrictions Assignments that create a range restriction on the output of a query can sometimes be pulled out of the query. The primary application of this technique is for update trigger statements, where a range restriction on the statement's loop variables can be applied directly to the map being updated.

The procedure is as follows. After the query has been fully simplified, we identify all assignments where the right-hand value is a single trigger variable that can be commuted up to the left-most position of a product term at the root of the query. These assignments are extracted from the query and used to create a mapping from loop variables to trigger variables, which is applied to both the query and the variables of the map being updated.

For instance, consider the delta query from Example 12. Its simplified version contains terms of the form: $(A:=x)*R(A)*S(B)$. Here, we can extract the assignment and use it to eliminate the loop over variable A in the update statement; the final statement is **foreach** B **do** $Q[x,B]$ += $\Delta_R Q[x,B]$. Note that one of the variables appearing on the left-hand side of the update statement has been bound to a corresponding trigger variable (x).

A similar technique is crucial for efficiently maintaining nested aggregate deltas, as seen in the following example.

Example 13 Consider the query $Q[A,B]=(B:=R(A))$. If R contains 2 tuples as follows, then $Q[A,B]$ is:

$[\![R(A)]\!](\mathbf{D},\langle\rangle)$	A		
	1	$\mapsto$	$\langle 1,1\rangle$
	2	$\mapsto$	$\langle 3,3\rangle$

$[\![Q]\!](\mathbf{D},\langle\rangle)$	A	B		
	1	1	$\mapsto$	$\langle 1,1\rangle$
	2	3	$\mapsto$	$\langle 1,1\rangle$

The delta of Q with respect to the update $+R(a)$ is

$$\Delta_{+R(a)}Q[A,B]=(B:=R(A)+(A:=a))-(B:=R(A))$$

The GMR for the delta with respect to the insertion of $\langle A:1\rangle$ into R includes two tuples with nonzero multiplicities:

$[[\Delta_{+R(1)}Q]](\mathbf{D},\langle\rangle)$	A	B		
	1	1	$\mapsto$	$\langle -1,-1\rangle$
	1	2	$\mapsto$	$\langle 1,1\rangle$

However, while evaluating the delta, two intermediate GMRs are instantiated with $|R|$ tuples each:

$$\left(\begin{array}{c|cc} & A & B \\ \hline 1 & 1 & 2 \mapsto \langle 1,1\rangle \\ 2 & 2 & 3 \mapsto \langle 1,1\rangle \end{array}\right) - \left(\begin{array}{c|cc} & A & B \\ \hline 1 & 1 & 1 \mapsto \langle 1,1\rangle \\ 2 & 2 & 3 \mapsto \langle 1,1\rangle \end{array}\right)$$

Tuples in these GMRs corresponding to tuples with zero multiplicities in $\Delta_{+R(1)}R(A)$ (i.e., $\langle A:2,B:3\rangle$) cancel out. A simpler, equivalent query would be:

$$\Delta_{+R(x)}Q[A,B]=(A:=x)*((B:=R(A)+1)-(B:=R(A)))$$

Recall the delta rule for nested queries

$$\Delta_u(x:=Q)=(x:=Q+\Delta_u Q)-(x:=Q).$$

After computing the nested delta $\Delta_u Q=(\Delta_u Q)_{rr}*(\Delta_u Q)e$, we extract all range restrictions $(\Delta_u Q)_{rr}$ and prepend them to the delta of the full expression. The revised delta rule is:

$$\Delta_u(x:=Q)=(\Delta_u Q)_{rr}*((x:=Q+(\Delta_u Q)_e)-(x:=Q)).$$

6 Examples

In this section, we provide several examples of higher-order IVM. Our goal is to illustrate how the heuristic optimizations interact to produce an efficient view maintenance program and to highlight interesting behaviors of DBToaster.

The examples are rather involved. To promote clarity, we explicitly give output variables with maps. We write $Q[\vec{x_{\text{out}}}]$ to denote a map Q with output variables $\vec{x_{\text{out}}}$, which form the schema of the query result.

6.1 Simplified TPC-H query 18

We explain the DBToaster compilation process on a query that contains an equality-correlated nested aggregate:

```
SELECT C.CK, SUM(LI.QTY) FROM C, O, LI
WHERE C.CK = O.CK AND O.OK = LI.OK AND
  100 < (SELECT SUM(LI1.OK) FROM LI AS
          LI1 WHERE LI.OK = LI1.OK)
GROUP BY C.CK
```

The query is a simplified version of Q18a from our test workload (see the appendix in our technical report [24]). For simplicity, we use the condensed schema $C(CK)$, $O(CK,OK)$,

on insert into C **values** `(ck)`:
01 $Q[ck]$ += $Q_C[ck]$
02 **foreach** `OK` **do** $Q_{LI}[ck, OK]$ += $Q_{LI,C}[ck, OK]$
03 $Q_{O1}[ck]$ += 1

on insert into O **values** `(ck,ok)`:
04 $Q[ck]$ += $Q_{O1}[ck] * Q_{O2}[ok] * (x := Q_{O2}[ok]) * (100 < x)$
05 $Q_{LI}[ck, ok]$ += $Q_{O1}[ck]$
06 $Q_{LI,C}[ck, ok]$ += 1
07 $Q_C[ck]$ += $Q_{O2}[ok] * (x := Q_{O2}[ok]) * (100 < x)$

on insert into LI **values** `(ok,qty)`:
08 **foreach** `CK` **do**
$Q[CK]$ += $Q_{LI}[CK, ok]$
$* \big(((Q_{O2}[ok] + qty) * (x := Q_{O2}[ok] + qty))$
$- (Q_{O2}[ok] * (x := Q_{O2}[ok]))\big) * (100 < x)$
09 **foreach** `CK` **do**
$Q_C[CK]$ += $Q_{LI,C}[CK, ok]$
$* \big(((Q_{O2}[ok] + qty) * (x := Q_{O2}[ok] + qty))$
$- (Q_{O2}[ok] * (x := Q_{O2}[ok]))\big) * (100 < x)$
10 $Q_{O2}[ok]$ += qty

Fig. 4 DBToaster insert trigger program for Q18a

and $LI(OK, QTY)$. Figure 4 shows the generated trigger program. The AGCA expression Q for the query is:

$$\mathrm{Sum}_{[CK]}\big(C(CK) * O(CK, OK) * LI(OK, QTY) * QTY * (x := Q_n) * (100 < x)\big)$$

where $Q_n = \mathrm{Sum}_{[]}\big(LI(OK_1, QTY_1) * (OK = OK_1) * QTY_1\big)$. First, we simplify the subquery Q_n. Unification eliminates the equality predicate to yield an expression with no input variables: $Q'_n = \mathrm{Sum}_{[OK]}\big(LI(OK, QTY_1) * QTY_1\big)$.

Due to space limitations we only show the derivation of insertions into Orders O and Lineitem LI. Insertions into Customer C are a simple extension, while deletions for all relations are duals of insertions and are omitted entirely.

Insertions into orders The first-order delta of Q for the insertion of a single tuple $\langle CK : ck, OK : ok\rangle$ is

$$\Delta_{+O(ck,ok)} Q := \mathrm{Sum}_{[CK]}\big(C(CK) * LI(OK, QTY) * (OK := ok) * (CK := ck) * QTY * (x := Q'_n) * (100 < x)\big)$$

The delta expression gets simplified after propagating the assignments: every occurrence of OK and CK is replaced with ok and ck, respectively; these assignments are also safe to remove. The delta expression is the following:

$$\mathrm{Sum}_{[ck]}\big(C(ck) * LI(ok, QTY) * QTY * (x := Q'_n) * (100 < x)\big)$$

with OK replaced by ok inside Q'_n.

Query decomposition splits the delta into three parts: $C(ck)$ has no common columns with the rest of the expression and is materialized as a separate map. The remaining expression can also be divided into two subexpressions that share only the trigger variable ok. Then, since the selection predicate is being applied to a singleton, we can safely materialize only the aggregate in the assignment. Applying these optimizations yields the following materialization decision:

$$\mathscr{M}\big(\mathrm{Sum}_{[ck]}(C(ck))\big) * \mathscr{M}\big(\mathrm{Sum}_{[ok]}(LI(ok, QTY) * QTY)\big) * \mathrm{Sum}_{[]}\big((x := \mathscr{M}(Q'_n)) * (100 < x)\big)$$

The trigger statement uses the following set of views (Fig. 4, line `04`, note that Q_{O2} is used twice):

$$Q_{O1}[CK] := \mathrm{Sum}_{[CK]}\big(C(CK)\big)$$
$$Q_{O2}[OK] := \mathrm{Sum}_{[OK]}\big(LI(OK, QTY) * QTY\big)$$

The delta query for $Q_{O1}[CK]$ and insertions into C is:

$$\Delta_{+C(ck)} Q_{O1} := \{\langle CK : ck\rangle \mapsto \langle 1, 1\rangle\}$$

which corresponds to trigger statement `03`. $Q_{O2}[OK]$ is maintained similarly with trigger statement `10`.

Insertions into lineitem With the nested subquery correlated on an equality, DBToaster chooses to compute the first-order delta of Q for the insertion of a single tuple $\langle OK : ok, QTY : qty\rangle$. The revised rule for nested subqueries yields:

$$\Delta_{+LI(ok,qty)}(x := Q'_n) := (OK := ok) * \big((x := Q'_n + qty) - (x := Q'_n)\big)$$

Following the delta rule for products

$$\Delta_{+LI(ok,qty)} Q := \mathrm{Sum}_{[CK]}\big(C(CK) * O(CK, OK) * \Delta Q_{LI} * QTY * (100 < x)\big)$$

where

$$\Delta Q_{LI} := (OK := ok) * \big((QTY := qty) * (x := Q'_n) + LI(OK, QTY) * \big((x := Q'_n + qty) - (x := Q'_n)\big) + (QTY := qty) * \big((x := Q'_n + qty) - (x := Q'_n)\big)\big)$$

Polynomial expansion, partial evaluation, and unification result in:

$$\Delta_{+LI(ok,qty)} Q := \mathrm{Sum}_{[CK]}\big(C(CK) * O(CK, ok) * (LI(ok, QTY) * QTY * (x := Q'_n + qty) - LI(ok, QTY) * QTY * (x := Q'_n) + qty * (x := Q'_n + qty)) * (100 < x)\big)$$

Decomposition and polynomial expansion let us extract the terms $\text{Sum}_{[CK,ok]}(C * O)$ and $\text{Sum}_{[ok]}(LI(ok, QTY) * QTY)$ as separate maps. The rewrite rules for nested aggregates and input variables materialize Q'_n. The final materialization is:

$$\begin{aligned}&\mathscr{M}\big(\text{Sum}_{[CK,ok]}(C(CK) * O(CK, ok))\big)\\&\quad *\big(\mathscr{M}(Q_2) * (x := \mathscr{M}(Q'_n) + qty)\\&\quad -\mathscr{M}(Q_2) * (x := \mathscr{M}(Q'_n))\\&\quad +qty * (x := \mathscr{M}(Q'_n) + qty)\big) * (100 < x)\end{aligned}$$

where $Q_2 = \text{Sum}_{[ok]}\big(LI(ok, QTY) * QTY\big)$.

Apart from the outermost materialization (of $C \bowtie O$), the remaining five materialized maps are identical to Q_{O2}, which is already being maintained. Thus, only one additional view, $Q_{LI}[CK, OK] := \text{Sum}_{[CK,OK]}(C(CK) * O(CK, OK))$, has to be maintained. Rewriting the materialization decision produces trigger statement `08`.

Note that statement `08`, which updates Q, does include a loop. However, even though DBToaster is not explicitly aware of TPC-H's foreign key dependencies, in this example, only one customer (CK) will be updated.

Q_{LI} is maintained in a manner analogous to that of Example 6, resulting in trigger statements `02`, `03`, `05`, and `06`.

6.2 The Pricespread query (PSP)

Next, we look at the query PSP from our test workload, which has two nested aggregates:

```
SELECT SUM(A.P - B.P) FROM A, B WHERE B.
V > (SELECT SUM(B'.P * 0.0001) FROM B
 AS B') AND A.V>(SELECT SUM(A'.P * 0.
0001) FROM A AS A')
```

Again, for simplicity, we use the condensed schema $B(P, V)$ and $A(P, V)$. The AGCA expression Q for the query is:

$$\begin{aligned}&\text{Sum}_{[\,]}\big(B(BP, BV) * A(AP, AV) * (AP - BP)\\&\quad *(v1 := \text{Sum}_{[\,]}(B(BP', BV') * BV' * 0.0001))\\&\quad *(BV > v1) * (v2 := \text{Sum}_{[\,]}(A(AP', AV') * AV'\\&\quad * 0.0001)) * (AV > v2)\big)\end{aligned}$$

Figure 5 shows the trigger program. Since the aggregates have no correlated variables, they can be decorrelated. Subsequently, there is no benefit to using deltas to update the final query result and our compilation heuristics decide on full recomputation for updates to both A and B. Hence, rather than describing the full compilation process for this example, we focus on the process of materializing the full query.

The join graph of this expression is intriguing. It consists of two mostly disconnected, symmetric components, one for

on insert into `B` **values** (bv, bp) :

```
01  Q_B1[bv] += bp
02  Q_B2[] += bv
03  Q_B3[bv] += 1
04  Q[] :=
      (Sum_[]((v1 := Q_B2[] * 0.0001) * Q_B3[BV'] * (BV' > v1))
      * Sum_[]((v2 := Q_A2[] * 0.0001) * Q_A1[AV'] * (AV' > v2)))
    -(Sum_[]((v1 := Q_B2[] * 0.0001) * Q_B1[BV'] * (BV' > v1))
      * Sum_[]((v2 := Q_A2[] * 0.0001) * Q_A3[AV'] * (AV' > v2)))
```

Fig. 5 The DBToaster trigger program for PSP insertions into B. The deletion trigger for B and the triggers for A are symmetric

$B(BP, BV)$ and one for $A(AP, AV)$. In fact, the only edge between these two is the term $(AP - BP)$. Our materialization strategy exploits both this, and the fact that integer addition and bag union are identical in AGCA.

Starting with the default materialization strategy $\mathscr{M}(Q)$, we perform polynomial expansion (Rule 2). Because AGCA does not separate integer addition from bag union, this distributes the rest of the expression over the term $(AP - BP)$.

$$\begin{aligned}&\mathscr{M}\big(\text{Sum}_{[\,]}\big(B(BP, BV) * A(AP, AV) * AP\\&\quad *(v1 := \text{Sum}_{[\,]}(B(BP', BV') * BV' * 0.0001))\\&\quad *(v2 := \text{Sum}_{[\,]}(A(AP', AV') * AV' * 0.0001))\\&\quad *(BV > v1) * (AV > v2)\big)\big)\\&\quad -\mathscr{M}\big(\text{Sum}_{[\,]}\big(B(BP, BV) * A(AP, AV) * BP\\&\quad *(v1 := \text{Sum}_{[\,]}(B(AP', BV') * BV' * 0.0001))\\&\quad *(v2 := \text{Sum}_{[\,]}(A(AP', AV') * AV' * 0.0001))\\&\quad *(BV > v1) * (AV > v2)\big)\big)\end{aligned}$$

We can now decorrelate the nested aggregates (Rule 4). This expression contains two identical aggregates, each computing the total volume of B or A. We call these Q_{B2} and Q_{A2}. As only one relation appears in each aggregate, maintenance requires only a single statement each, shown in the trigger program for B as statement `02`.

$$\begin{aligned}&\text{Sum}_{[\,]}\big(\mathscr{M}\big(\text{Sum}_{[BV,AV]}(B(BP, BV) * A(AP, AV) * AP)\big)\\&\quad *(v1 := Q_{B2}[\,] * 0.0001) * (BV > v1)\\&\quad *(v2 := Q_{A2}[\,] * 0.0001) * (AV > v2)\big)\\&\quad -\text{Sum}_{[\,]}\big(\mathscr{M}\big(\text{Sum}_{[BV,AV]}(B(BP, BV) * A(AP, AV)\\&\quad *BP)\big) * (v1 := Q_{B2}[\,] * 0.0001) * (BV > v1)\\&\quad (v2 := Q_{A2}[\,] * 0.0001) * (AV > v2)\big)\end{aligned}$$

After polynomial expansion the expression computes two joins instead of one. The hypergraphs of the simpler joins, however, contain disconnected components. We can apply

decomposition (Rule 1) to each.

$$
\begin{aligned}
&\mathrm{Sum}_{[\,]}\big(\mathscr{M}\big(\mathrm{Sum}_{[BV]}(B(BP,BV))\big)\\
&\quad *\mathscr{M}\big(\mathrm{Sum}_{[AV]}(A(AP,AV)*AP)\big)\\
&\quad *(v1 := Q_{B2}[\,]*0.0001)*(BV > v1)\\
&\quad *(v2 := Q_{A2}[\,]*0.0001)*(AV > v2)\big)\\
&-\mathrm{Sum}_{[\,]}\big(\mathscr{M}\big(\mathrm{Sum}_{[BV]}(B(BP,BV)*BP)\big)\\
&\quad *\mathscr{M}\big(\mathrm{Sum}_{[AV]}(A(AP,AV))\big)\\
&\quad *(v1 := Q_{B2}[\,]*0.0001)*(BV > v1)\\
&\quad *(v2 := Q_{A2}[\,]*0.0001)*(AV > v2)\big)
\end{aligned}
$$

Now, no further rules are applicable. We materialize four additional maps: for each volume we maintain both the count and sum of prices of both relations. In the trigger program, maps Q_{B3} and Q_{A3} maintain the counts using statement `03` and its dual in A; maps Q_{B1} and Q_{A1} maintain the price sums using statements `01` and its dual in A.

Because the total volume of each relation changes with every insertion, we must recompute the price and count totals for the relation that changes. Specialized data structures such as range trees could further reduce the cost of doing so by allowing us to efficiently maintain expressions of the form $\mathscr{M}\big(\mathrm{Sum}_{[BV]}\big(B(BP,BV)*(BV > v1)\big)\big)$.

Nevertheless, by exploiting the connection between addition and bag union, DBToaster evaluates this expression using exclusively scans (in contrast to first computing a Cartesian product as a traditional database system would).

7 Query engine compilation and system overview

DBToaster is a compiler implemented in OCaml. It consists of several compilation stages, their intermediate representations that transform AGCA into an efficient imperative implementation and a runtime library to provide basic data loading and instrumentation of our generated query engines. This article has primarily focused on frontend compilation, which implements AGCA and its GMRs from Sect. 3. In this section, we briefly outline the intermediate representations used in our compiler's backend, including a simple trigger language based on AGCA, as well as a functional language and an imperative language. Compilation applies optimizations in all of these stages prior to code generation.

7.1 Backend compilation

We have implemented our AGCA queries and the GMR data model as a trigger program language that combines pure AGCA expressions with view maintenance as side effects. We parse AGCA expressions directly from SQL and produce trigger programs from higher-order IVM.

While we can directly interpret trigger programs, for efficient execution, we translate them to a lower-level program. The next step is a functional language, K3 [40], which is inspired by the Collection Programming Language in the Kleisli functional query system [8,43]. Its main features are its use of a nested collections data model, the incorporation of group-by aggregation to the query language, and a rich set of optimizing program rewrites on collection transformers. Examples of our collection transformers include `map`, `fold`, `groupby`, and `flatten`, as frequently found in functional programming languages such as Scala, Haskell and OCaml.

Our program transformations rely on the definitions of collection transformers through structural recursion, and the subsequent axiomatization of the monad construct realized by structural recursion. By bringing programming language inspired (compile-time) optimizations to query processing, we are able to perform holistic query optimization, which is frequently limited by operator abstractions in query plans. Other recent works have observed a variety of benefits from holistic optimization [26,32]. In the rest of this section, we highlight K3's compile-time optimizations, relating them to common methods used in database query optimizers. A full coverage of this topic is outside the scope of this work.

Our transformations start by normalizing program expressions into a lambda-conditional form, where `if`-statements are lifted to the minimal lambda expressions binding any variables present in the conditional. With conditionals often determining the use of delta queries or initial value queries, this ensures maximal applicability of optimizations inside the two different forms of queries. Next, DBToaster applies constant simplification as well as a conservative form of beta reduction to inline repeated occurrences of scalar values and collection arguments that are used at most once during function application.

Subsequently, we aggressively apply code inlining and simplification through a series of function composition and fusion operations. Along with our collection transformer optimizations, this helps to eliminate large intermediate collections in delta processing and initial value computations, in a similar way to deforestation algorithms [30]. Other notable optimizations include common subexpression elimination to factor repeated expressions into function application, as well as delayed tuple construction to minimize the flow of packed values throughout expressions. These optimizations are summarized in Fig. 6. DBToaster's Scala code generator emits source code directly from our functional representation, making extensive use of Scala's standard libraries.

Our intermediate imperative language defines a minimal set of imperative control constructs (conditionals, loops, and sequential evaluation) and is designed to facilitate code generation in imperative target languages such as C++. Currently, we perform a limited set of optimizations on our imperative

Optimization	Description
Condition normalization	Yields a normalized program where `if`-expressions appear only as the first expression in a function body.
Beta reduction	Inlines function arguments, with a compile-time cost analysis to avoid expensive argument re-evaluation.
Fusion and deforestation	Removes intermediate collection construction by fusing multiple collection transformations.
Partial aggregation	Reduces intermediate collection sizes, especially for nested collections undergoing flattening.
Common subexpressions	Extracts repeated expressions as function application, with consideration of inlining applied by beta reductions.
Effect normalization	Lifts effects to their earliest feasible application to reduce the memory footprint of large values.
Index construction	Rewrites collection transformers applying equality predicates to build and probe index data structures.

Fig. 6 A summary of program transformations and optimizations applied in the K3 language

language. As ongoing work, we are working on optimizations that are not captured in our functional layer. This includes low-level loop and peephole optimizations, for example loop unrolling and tiling, and the improved data locality and vector operations exposed therein. Many structural loop optimizations, such as loop fusion and ordering, are already captured by our collection transformer optimizations.

From an imperative representation, the DBToaster code generator supplements source code synthesis with the ability to select and adapt specialized data structures to the query being maintained. We implement our GMRs as a map data structure (e.g., an STL `map`) that associates tuples to their multiplicities. View caches are implemented as two-level nested maps, with the outer tier corresponding to input variables in our binding patterns, and the inner tier mapping output variables to multiplicities as above. This two-level mapping forms the primary index of our data structure and is accompanied by a range of secondary indexes based on access patterns in delta queries.

Specifically, we maintain secondary indexes for all binding patterns present in triggers produced by higher-order IVM. While our index selection could be refined by physical database design techniques, we found the number of extra indexes to be relatively small throughout our workloads.

DBToaster currently uses the Boost Multi-Index (BMI) library for its C++ data structures. BMI provide in-memory data structures that implement a generic container over composite types (e.g., tuples or C structs), with an ability to compose multiple types of indexes into a single data structure. Thus we can specify secondary indexes over different attributes, for in-memory implementations of hash, tree and sequential data structures. We have developed an equivalent library collection for Scala and have included it as part of our query runtime library.

7.2 System overview and application usage

The DBToaster compiler produces query processors that are aggressively specialized to a specific query workload, rather than ad hoc queries. End-users interact with a DBToaster-generated query processor in one of three ways:

1. *Standalone binaries*, where users may run the binary on a file, or specify a listening socket through which data can be sent to the engine. The engine can output a stream of view query results to a file or a network connection.
2. *Shared libraries*, where application developers may link against our library and directly access the data structures representing our views while they are concurrently maintained. We are exploring asynchronous notification methods to support push-based application logic, including callback functions and futures registered with our views.
3. *Source code*, where application developers may adapt and extend our query processing engine as desired, for example to use custom data structures to implement views.

DBToaster produces extensible query engines capable of custom stream pre-processing and workload generation, as well as on-demand querying of views. Our object-oriented design enables users to inherit our engine in their applications, where they may override a pre-processing method invoked on each arriving event. This allows users to perform basic data extraction, transformation, cleaning and logging functionality prior to delta processing.

Users may also direct our compiler to use the generalized form of higher-order IVM. The result is a query engine that mixes pull- and push-based processing, providing users with a rich API to retrieve query results. Our API methods pull and compute query results from a set of materialized views that are maintained with higher-order delta queries. This mode of operation produces results at a lower frequency than the application's update rates. The set of materialized views are those considered by higher-order IVM when starting delta rewrites one level down in the query.

Our compiled binaries implement a single-core, single-threaded query executor. Our implementation strategy has focused on novel view maintenance rather than the full range of state-of-the-art query execution mechanisms. Thus, our results represent a lower limit on performance and scalability, both of which could be substantially improved with a parallel engine. As ongoing work [21], we are developing a distributed main-memory runtime that exploits aggregate memory and network bandwidth available in large clusters and datacenters. Furthermore, we plan to use K3 for mul-

tithreaded and vectorized engines that utilize flexible view data structures as inspired by database cracking.

8 Experiment setup and methodology

For the experiments in this paper, we used the DBToaster Public Beta, rev. 2827, released on February 11th, 2013 [14]. We evaluated the experimental performance of DBToaster on Redhat Enterprise Linux on an Intel Xeon E5620 2.4 GHz processor with 16 GB of RAM (on a single core). The C++ code generated was compiled using g++ 4.4.6 and linked against the Boost library v1.50; generated Scala code was compiled with version 2.10.0 of the Scala compiler and run on the Java HotSpot VM (build 23.6-b04). Gperftools v2.0 was used to estimate the memory consumption of our query binaries.

We compare our compilation algorithm with a commercial DBMS with IVM capabilities (DBX) and a stream processing system (SPY). Since these systems are not optimized for our workload, we also provide a shared-infrastructure comparison by emulating their functionalities—query re-evaluation and IVM—within DBToaster-generated binaries.

Data and query workload Our workload covers algorithmic order book trading (financial), online business decision support scenarios (TPC-H) and scientific queries (MDDB, [33]), which involve computing a variety of statistics. Figure 3 lists the query and evaluation properties of our workload.[3] Due to limitations of DBToaster, we made several changes to the TPC-H queries: (1) We ignored ORDER BY clauses and, to make the results comparable, also dropped the LIMIT clause; (2) We rewrote MIN, MAX aggregates using equivalent nested subqueries; (3) We replaced Q13's LEFT OUTER join with a natural join; (4) Finally, for convenience we rewrote HAVING clauses into subqueries and inlined INTERVAL expressions into constants.

The financial queries VWAP, MST, AXF, BSP, PSP, and BSV were run on a 2.63 million tuple trace of an order book update stream, representing one day of stock market activity for MSFT. These are updates to a Bids and Asks table with the schema (`timestamp, order_id, broker_id, price, volume`). The TPC-H benchmark queries Q1-Q22, and SSB4 were run on a stream of updates adapted from a database generated by DBGEN [42]. We simulate a system that monitors a set of "active" orders by randomly interleaving insertions on all relations and injecting random deletions of Orders and Lineitem rows to keep the Orders and Lineitem tables at around 30,000 tuples and 120,000 tuples, respectively. All updates preserve the foreign key constraints that exist between the TPC-H tables. Most experiments use a stream synthesized from a scaling factor 0.1 database (100 MB), while our scaling experiments extend these results up to a scaling factor of 10 (10 GB). Finally, the scientific workload was run on a 3.6 million tuple trace (128 MB) of insertions into a table of atom positions during a molecular dynamics simulation.

DBToaster setup The DBToaster compiler produces IVM code for both C++ and Scala. The compilers for these languages produce binaries with distinct (and surprising) performance characteristics. Our evaluation includes the results for both languages.

DBToaster emulates the behavior of a traditional view maintenance system by terminating recursive delta materialization early. The remaining compiler stages (functional optimization and target-language generation) operate as usual. Our evaluation includes: (1) The HO-IVM algorithm as presented in this paper (DBToaster), (2) A full re-evaluation of the query on every change (REP), (3) The HO-IVM algorithm used without recursion (first-order deltas are materialized) to emulate traditional IVM (IVM), and (4) A naive application of the viewlet transform that aggressively materializes as much of each query as possible, creating view caches and employing partial materialization to decorrelate nested subqueries, but ignoring the rules for join graph decomposition and delta expression simplification.

For each compilation method, we measured the memory consumption of the C++ programs. To this end, we produced instrumented binaries for each experiment and processed the same fraction of the stream as without profiling.

DBMS setup We compare DBToaster against a commercial DBMS. Due to licensing restrictions, we refer to it using the anonymized name DBX. In order to measure the rate at which DBX is able to refresh the query results as consistently as possible with other systems, we preload all updates to be performed on all base tables into a single table called Agenda. The Agenda table's schema is the union of all of the input table schemas and includes columns identifying the type of update (insert or delete), the table being updated, and the update's sequence number. Each trial iterates over the updates in Agenda in order, inserting or deleting one tuple and then refreshing the query results, either by re-evaluating the query (DBX-REP), or by using the system's built-in capability to incrementally maintain materialized views (DBX-IVM). In order to minimize the overheads of the system, we disable log collection as much as possible.

For re-evaluation, we completely re-evaluate the query after each update and store its results in a separate table that gets truncated before each re-evaluation. Because generating materialized views that can be incrementally maintained is non-trivial, has many restrictions, and requires extra update

[3] The detailed queries can be found in our technical report [24] or on our Web site http://www.dbtoaster.org.

logs, for IVM we use the provided tuning advisor in order to derive the proper view setup for each of the queries.

In many cases, the tuning advisor suggested views that were not precisely identical to the input queries. We encountered situations in which the advisor added group-by columns or relaxed WHERE clauses by dropping conditions or replacing disjunctions with single expressions, covering a superset of the original condition. We can only speculate that these transformations were meant to allow the generated view to support answering a larger class of queries. For complex queries that could not be maintained as a single view, the advisor generated nested subviews to be incrementally maintained and a top-level view to be re-evaluated on every commit. Out of 36 queries that we experimented with, 20 required up to 5 nested subviews.

SPY setup As a second comparison point, we use a commercial stream processor. We refer to the stream processor using the anonymized name SPY, again due to licensing restrictions. One major semantic difference between traditional stream processing engines and DBToaster is that stream processing engines are optimized to operate on windows of input streams, while DBToaster is designed to handle the whole history of a stream. We benchmark SPY by reading the same Agenda table used for DBX directly into a stream to minimize event dispatch overheads.

We implemented the queries using the dialect of SQL supported by SPY. Since the queries in our benchmark cannot be efficiently expressed using window semantics, we used auxiliary in-memory tables for all relations. Our implementation of the queries assigns a monotonically increasing number to each event and dispatches it to a stream corresponding to the affected relation. This stream updates the in-memory relation by inserting or removing the affected tuple. Then, the query result is re-evaluated and recorded together with the event number and a timestamp. Full recomputation is necessary as SPY does not support IVM.

Although we attempted to maintain the original query semantics, the SQL dialect employed by SPY imposes some limitations. A severe limitation is that in-memory tables may not be joined together; each in-memory table may only be joined with a stream, requiring manual selection of a join order. Our heuristic for this order was to minimize the size of intermediate streams.

9 Experimental results

Our results show view refresh rates on stream traces, replayed with a timeout of two hours. Details of the traces are provided in Sect. 8. These results show that:

- DBToaster consistently outperforms the two commercial systems we tested against, often by multiple orders of magnitude (Fig. 7, 8).
- The performance gap between higher-order IVM and Traditional IVM is even greater within the DBToaster run-

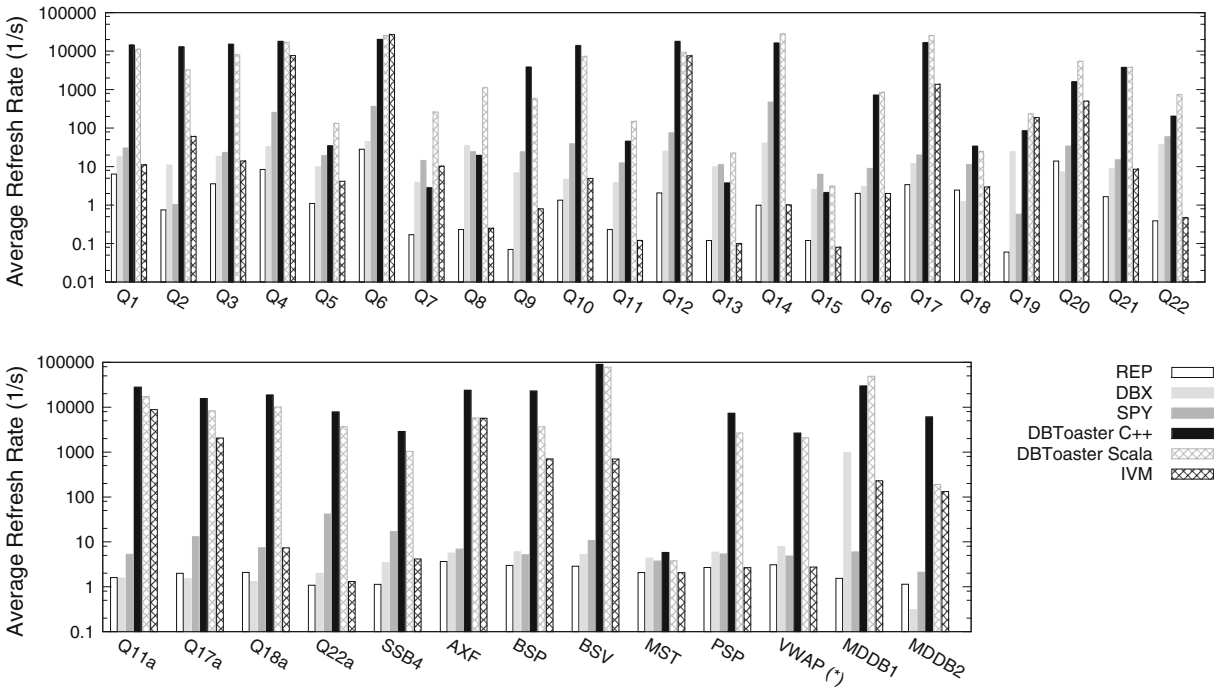


Fig. 7 DBToaster performance overview. Note the log scale on the *y*-axis. (*asterisk*) For VWAP, where DBToaster uses view caching, we compare against a strategy that avoids input variables

Query	REP C++	REP Scala	DBX Rep	DBX IVM	SPY	DBToaster C++	DBToaster Scala	Naive C++	Naive Scala	IVM C++	IVM Scala
Q1	5.39	6.41	17.74	1.58	29.46	14,109.08	11,214.98	5,378.30	1,500.17	9.83	11.16
Q2	1.76	0.75	10.74	1.24	1.01	12,742.67	3,239.07	0.03	0.03	659.94	60.33
Q3	8.19	3.58	18.22	0.7	22.55	15,045.62	8,021.88	3.54	2.02	121.10	14.03
Q4	24.99	8.42	32.3	2.06	252.04	17,604.34	16,911.45	63.26	51.65	11,614.26	7,678.18
Q5	5.46	1.10	9.67	0.32	19.05	34.18	131.68	0.15	0.19	25.22	4.14
Q6	27.76	28.29	44.24	0.43	361.97	20,021.32	25,509.86	19,025.53	26,485.42	21,030.09	26,910.07
Q7	1.70	0.17	3.84	1.35	14.1	2.76	261.01	0.03	0.03	10.49	10.17
Q8	0.39	0.23	34.83	2.45	23.88	19.42	1,122.93	0.09	0.07	0.34	0.25
Q9	0.29	0.07	6.75	1.76	23.74	3,778.52	578.81	0.03	0.03	1.15	0.80
Q10	5.57	1.34	4.6	0.47	38.23	13,697.35	7,317.77	0.12	0.19	79.45	4.92
Q11	0.63	0.23	3.74	2.84	12.32	45.12	149.39	0.26	0.16	0.41	0.12
Q11a	6.77	1.60	1.57	0.86	5.28	28,060.43	17,315.17	15,164.32	3,506.81	29,152.02	8,808.10
Q12	2.77	2.06	24.71	1.62	74.47	17,440.02	9,353.63	20.68	23.17	14,900.63	7,576.58
Q13	0.19	0.12	9.64	2.94	10.9	3.69	22.37	1.26	0.95	0.16	0.10
Q14	3.47	0.99	39.6	1.56	464.88	15,953.58	28,047.48	199.45	152.52	3.60	1.01
Q15	0.13	0.12	2.49	1.93	6.14	2.07	3.10	–	–	0.12	0.08
Q16	3.25	2.01	2.94	1.87	8.82	713.07	843.56	0.20	0.37	3.28	2.00
Q17	8.07	3.40	11.77	2.1	19.64	16,456.54	25,408.05	0.76	0.66	14,918.33	1,373.52
Q17a	6.22	1.99	1.51	1.34	13.06	15,617.53	8,285.00	1.18	0.99	6,190.91	2,060.98
Q18	1.44	2.45	0.08	1.2	11.16	33.51	24.86	0.34	0.27	1.75	2.95
Q18a	3.91	2.09	0.6	1.31	7.42	18,725.53	10,085.95	0.11	0.09	107.89	7.38
Q19	0.20	0.06	23.84	1.42	0.57	83.98	236.54	28.44	81.93	69.82	187.78
Q20	48.28	13.91	7.16	1.19	33.63	1,586.76	5,427.83	0.85	0.91	3,553.37	502.33
Q21	5.17	1.65	8.72	1.33	14.72	3,703.25	3,782.58	0.29	0.55	189.88	8.50
Q22	0.27	0.39	36.05	1.6	58.22	201.72	742.50	6.76	0.47	0.33	0.47
Q22a	0.94	1.08	1.4	1.98	41.68	7,868.03	3,687.80	176.88	68.31	1.19	1.31
SSB4	2.42	1.13	3.43	0.51	16.92	2,877.63	1,039.36	0.03	0.02	64.10	4.15
AXF	3.63	3.66	5.62	1.32	6.91	23,817.13	5,764.69	2,168.17	779.99	15,808.05	5,677.57
BSP	3.31	2.99	6	1.61	5.18	23,040.81	3,673.54	192.16	191.56	1,261.01	703.86
BSV	3.28	2.87	5.23	1.55	10.63	90,116.98	77,797.66	54,810.63	6,921.83	1,284.29	702.49
MST	3.59	2.07	4.37	1.26	3.73	5.81	3.81	5.90	5.88	1.52	2.06
PSP	2.90	2.68	5.93	1.96	5.38	7,319.93	2,658.12	362.82	365.24	2.87	2.67
VWAP	4.42	3.08	7.93	2.12	4.81	2,649.42	2,087.87	2,436.26	2,692.96	4.36	2.75
MDDB1	5.62	1.54	972.22	1.02	5.96	29,842.28	48,784.54	–	–	9,163.44	230.31
MDDB2	3.42	1.14	0.31	0.26	2.11	6,093.05	189.56	0.02	0.03	3,656.16	131.68

Fig. 8 Comparison between DBToaster and two commercial query engines (in view refreshes per second). Both the DBMS (DBX) and stream system (SPY) columns show the cost of full refresh on each update. Higher numbers are better

time. Through aggressive optimization, we believe that DBToaster's performance can be improved by at least another order of magnitude.
- DBToaster exhibits consistent performance and memory usage over time (Figs. 9, 10, and 11).
- These results scale to longer streams (Fig. 12).

In all figures, we use the following notation:

- **DBToaster** is the full HO-IVM algorithm.
- **REP** and **IVM** are DBToaster repeatedly re-evaluating queries, and emulating non-recursive IVM, respectively.
- **Naive** is a simplified form of the viewlet transform that aggressively materializes entire delta queries.
- **DBX-REP** and **DBX-IVM** are a commercial database system performing view maintenance by re-evaluation and non-recursive IVM, respectively.
- **SPY** is a commercial stream processing engine.

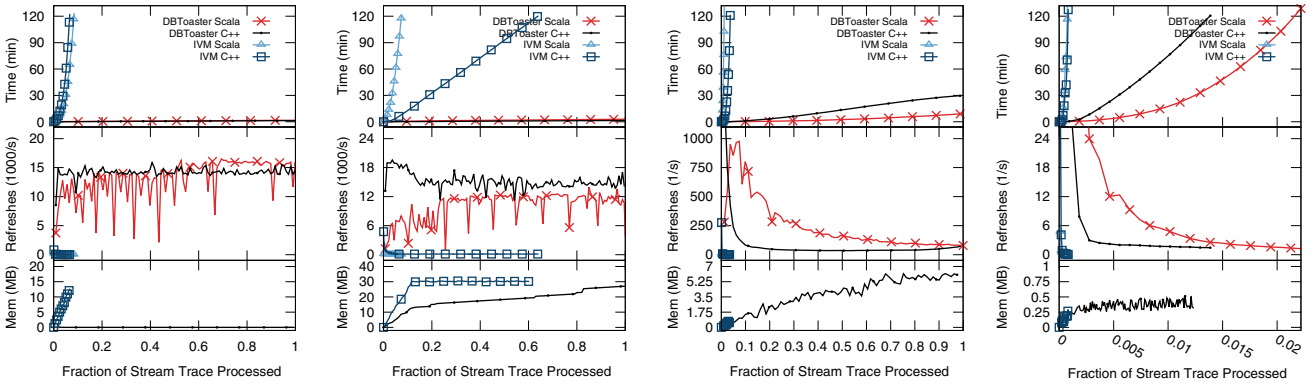
DBToaster results are presented with both C++ and Scala as target languages.

9.1 Higher-order IVM performance

We now compare the performance of DBToaster with a commercial DBMS (DBX) and a stream processor (SPY).

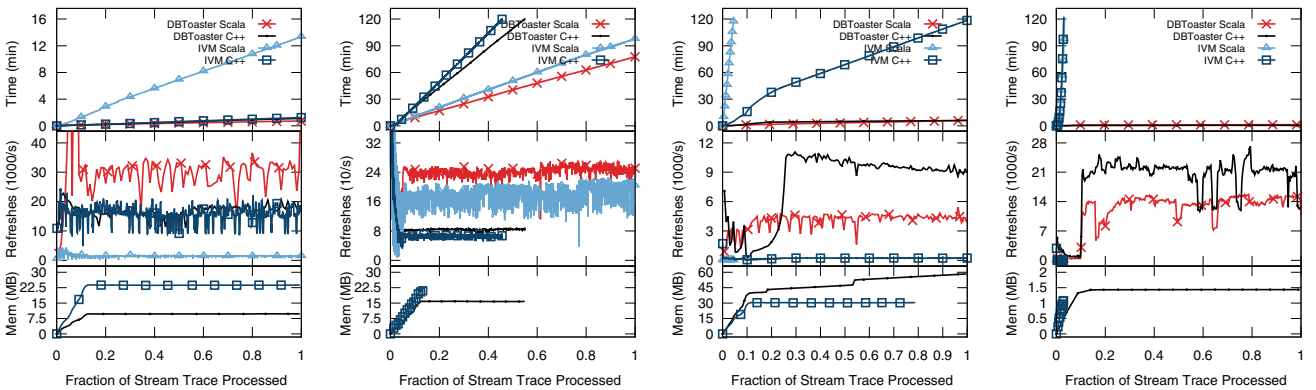
Comparison with commercial systems Figure 8 shows the performance of DBToaster's higher-order IVM alongside all comparison systems. We summarize our findings, because an in-depth itemized breakdown of overheads is outside the scope of this article.

When recomputing the query results after each update (DBX-REP), DBX experienced view refresh rates between 0.08 and 972.22 refreshes per second, with average and median values of 37.03 and 6. When using DBX's support for IVM, however, view refresh rates dropped to between 0.14 and 2.94. This drop in performance when using IVM is



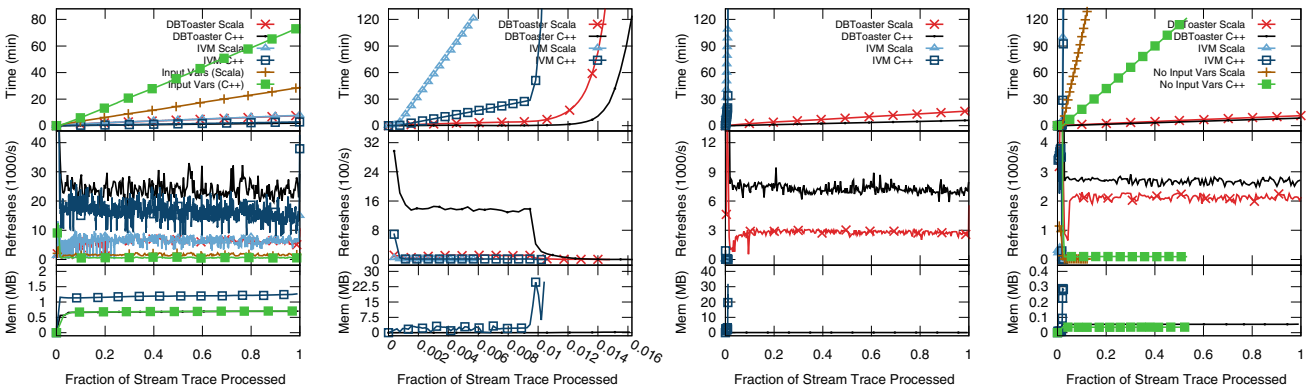
(a) Q1 **(b)** Q3 **(c)** Q11 **(d)** Q15

Fig. 9 **a** A join-free query. **b** A 3-way linear join. **c** A 2-way join with an aggregate subquery in the FROM clause and an uncorrelated nested aggregate. **d** A 2-way join with an aggregate subquery in the FROM clause and an inequality-correlated nested aggregate in the EXIST clause. DBToaster completes only a small fraction of the trace since the update cost grows quadratically with the number of distinct suppliers



(a) Q17 **(b)** Q19 **(c)** Q21 **(d)** Q22a

Fig. 10 **a** A 2-way join with an equality-correlated nested aggregate. **b** A 2-way join with three disjunctive clauses. **c** A 4-way join with an equality- and an inequality-correlated subqueries. **d** A single relation with an equality- and an inequality-correlated nested aggregates. Insertions into the Customer relation complete within the first 10 % of the stream



(a) AXF **(b)** MST **(c)** PSP **(d)** VWAP

Fig. 11 **a** A 2-way inequality join. DBToaster outperforms view caching due to the large domain of the input variables. **b** A 2-way join with two uncorrelated, and two inequality-correlated nested aggregates. None of the tested engines completed the trace within the 2-h limit. **c** A 2-way join with two uncorrelated nested aggregates. **d** A single relation with an inequality-correlated and an uncorrelated nested aggregate. DBToaster chooses the view cache method, so we compare against an approach that aggressively avoids input variables

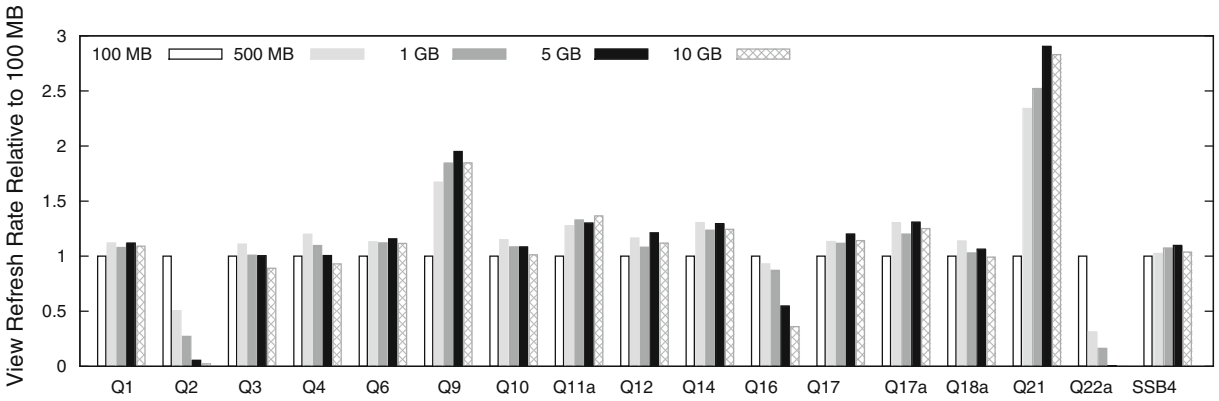


Fig. 12 Performance scaling on a subset of TPC-H queries

counter-intuitive and prompted us to trace the execution of our program. DBX's tracing utility revealed that most of the execution time was spent parsing several parametrized system queries used in the bookkeeping. As the amount of useful work to be performed after a single update is quite small, the time spent parsing those system queries ends up dominating the overall running time. Maintaining catalog information across many tables for high-rate updates also substantially impacts latencies and throughput.

The performance gap between SPY and DBToaster is a result of the lack of support for IVM in SPY, and synchronization used to prevent the asynchronous system from producing inconsistent results. Due to the nature of the test queries, we are unable to make use of SPY's window semantics and are forced to use in-memory tables instead. Even though we use indexes on the in-memory tables wherever it makes sense, SPY seems to be unable to take full advantage of them in queries with complex predicates, contributing to poor performance, as exemplified in Q19.

Join-free queries The simplest queries in our workload, Q1 (Fig. 9a) and Q6, aggregate TPC-H's Lineitem relation. As these queries involve only one relation, the first-order delta depends solely on the values inserted or deleted.

The materialized view of Q6 stores a single aggregate value and has a constant update cost. Thus, the view refresh rates of the DBToaster, Naive, and IVM methods are almost identical. In all these cases, the generated Scala programs outperform the C++ programs. The REP compilation exhibits low refresh rates as it performs a complete scan over Lineitem upon every update. Unlike the other methods for which the memory overhead is negligible, REP requires a bounded amount of memory to store the set of active tuples.

Q1 evaluates multiple group-by aggregates over Lineitem. DBToaster treats these aggregates as separate AGCA expressions and maintains each individually. Since many of these share common subexpressions, duplicate view elimination and polynomial expansion are essential for achieving high view refresh rates. Consequently, although Q1 has substantially more aggregates than Q6 (8 vs 1), the view refresh rate of the C++ code is only 30 % lower. Because the result set contains a fixed number of tuples (based on the limited domain of the group-by columns), DBToaster uses only a fixed amount of memory to store the additional maps.

DBToaster inlines the computation of algebraic aggregates. For instance, DBToaster computes averages from separate sum and count aggregates: Because the current incarnation of AGCA supports only one "multiplicity" per tuple, average is expressed as the product of the sum and inverse count. HO-IVM requires two recursive steps to separate out the (linear) count from the (nonlinear) inverse count. This accounts for IVM's poor performance on Q1, as it must fully recompute the inverse count on every change. As future work, we plan to extend AGCA to generalize GMRs to have multiple "multiplicities". This will allow DBToaster to store multiple aggregate values per tuple and improve the efficiency of this class of queries.

Equijoins Q11a, Q12, Q14, and Q19 (Fig. 10c) contain two-way joins without nested aggregates. The first level deltas correspond nearly to the base relations. For Q11a and Q12, the DBToaster and IVM methods produce virtually identical results. Because IVM materializes entire base relations, it has a slightly lower refresh rate for Q19 than DBToaster, which materializes only relevant columns. As in Q1, Q14 has to maintain an inverse count, resulting in poor performance for IVM. In Naive, range restrictions are not extracted from deltas of nested aggregate expressions (Sect. 5.3), necessitating a full scan of each materialized nested aggregate whenever it changes. The effect of this optimization is most evident in these four queries.

Query decomposition also plays an important role in efficiency of DBToaster for queries containing linear joins of 3 or more relations. Decomposition avoids materialization of cross products, improving performance and reducing memory consumption. For instance, the delta of Q10 (a 4-way equijoin) with respect to the Orders relation creates a cross

product between Customer and Lineitem (which are only connected through Orders in the original query). In Naive, the entire cross product is materialized, resulting in performance five orders of magnitude worse.

Due to foreign key constraints in the TPC-H schema (of which DBToaster is not made aware) most loops in Q3's trigger program have only one iteration, and the cost of updating either the Orders or Lineitem relation is constant. For queries with multi-way joins and selection predicates (Q3, Q5, Q10, SSB4, MDDB1, and MDDB2), DBToaster further outperforms IVM by pushing predicates into the materialized views and projecting away unused columns.

DBToaster considers the contents of Nation, Region, and all the scientific relations except AtomPositions as static. It loads static relations into memory before processing the streams. It avoids materialization of deltas needed to support updates to these relations, effectively reducing the join width of certain queries (Q5, Q10, SSB4, MDDB1, and MDDB2) and eliminating several potentially high maintenance maps.

Nested aggregates Q17 (Fig. 10a), Q17a, and Q18a[4] are multi-way join queries with nested aggregates that are correlated on an equality with the outer query. In these cases, DBToaster's strong performance comes from decorrelating the nested subquery and range-restricting the domain of the generated delta expressions for updates to the Lineitem relation (on which all nested subqueries are based).

Q22a (Fig. 10d) includes two nested aggregates, an uncorrelated aggregate on Customer and an equality-correlated aggregate on Orders. The first subquery causes DBToaster to re-evaluate the top-level query (as per Sect. 5.1) since the delta with respect to updates to Customer is not simpler than the whole query. This re-evaluation strategy iterates over the whole Customer relation (materialized with only necessary columns) to compute the top-level aggregate for the customers with no orders. In contrast, DBToaster uses an incremental strategy for updates into Orders, since the equality-based correlation between the second subquery and the outer query restricts the domain of the corresponding delta expression, making the update cost constant. Therefore, Q22a has two different maintenance costs for updates into the base relations. This is seen in the performance graph as the query's slow startup ends once the last customer has been inserted.

VWAP (Fig. 11d) has a nested aggregate correlated on an inequality. The small domain of the correlation variable (`price`) makes this an ideal candidate for view caching. The performance graph shows the benefit of view caching over avoiding the materialization of maps with input variables.

PSP (Fig. 11c) includes two uncorrelated nested aggregates. It benefits from top-level query re-evaluation on each update. As in Sect. 6.2, polynomial expansion and graph decomposition are essential to avoid computation of a cross product between the base relations. DBToaster evaluates the query using six auxiliary materialized views with constant time updates: Two views maintain single aggregate values, while the others are linear in the number of distinct values of the column being compared to the nested aggregate (`volume`). The finite domain of these values results in a nearly constant view refresh rate and memory consumption.

MST (Fig. 11b) is fundamentally similar to PSP, but rather than comparing its uncorrelated aggregates against columns from the base relations, they are each compared against another nested aggregate correlated on an inequality. This is a worst case scenario for DBToaster, as it cannot incrementally process this query in better than $O(n^2)$ time without specialized indexes (e.g., aggregate range trees).

Inequijoins AXF (Fig. 11a) and BSP are 2-way joins with inequality join-predicates. The performance graph of AXF shows the inefficiency of view caching in this case. The view caching approach treats both the join variable (`price`) and one of the aggregate variables (`volume`) as input variables; together, these input variables have an extremely large domain. In BSP, the join variable (timestamp) also has an unbounded domain. In both cases, DBToaster outperforms view caching by precluding materialized views with input variables. DBToaster also achieves a small speed boost compared to IVM by not materializing the entire base relation.

Queries with EXIST or IN clauses Q2, Q4, Q16, and Q21 (Fig. 10c) contain clauses that check for the existence of the nested subquery results. DBToaster transforms each subquery into a count aggregate, assigns this value to a fresh variable, and adds an additional constraint over that variable according to the semantics of the clause (e.g., $x = 0$ for the NOT EXIST clause). As all the subqueries of the above queries are correlated on an equality, DBToaster decides to incrementally maintain the top-level views for updates to the subquery relations. For queries that are also correlated on an inequality (Q2 and Q21), DBToaster avoids materializing maps with input variables due to the large domain of the correlation variables (`supplycost` and `suppkey`, respectively). Q21 has constant time updates to Lineitem and Orders, and a linear time update in the number of orders for one supplier. The higher update cost results in a lower view refresh rate within the first 20 % of the stream, until insertions into the Supplier relation complete.

Subqueries in FROM clauses DBToaster maintains separate materialized views for subqueries that appear in the FROM clause (Q7, Q8, and Q9). For Q7 and Q8, we observe that the C++ backend fails to transform DBToaster's functional representation into efficient imperative code, causing huge memory overheads and poor performance. In contrast, DBToaster derives Scala code directly from its internal representation.

[4] Q17, Q17a produce incorrect results due to floating point errors.

The Scala compiler further optimizes the code, resulting in performance better by two orders of magnitude.

Complex queries The remaining TPC-H queries Q11, Q13, Q15, Q18, Q20, and Q22 combine the above characteristics. Our experiments show that the update costs for these queries coincide with their structural complexity.

Q11 (Fig. 9c) has a group-by aggregate in its FROM clause and an uncorrelated nested aggregate that appears in an inequality at the top level. DBToaster exploits the fact that both subqueries share the same structure to reduce the number of generated maps. Since the update stream contains only insertions to the base relations, the amount of memory used to store additional views grows continuously. The costs of updating Supplier and Partsupp are linear in the number of distinct `partkey` values. Thus, the view refresh rate levels off as the number of tuples in the materialized views reaches the maximum number of distinct group-by values.

Q15 (Fig. 9d) is a variation of the original TPC-H query where a nested subquery and an EXIST clause replace the max aggregate. Since both subqueries are identical, duplicate view elimination reduces the number of auxiliary views. However, the update cost for this query grows quadratically with the number of distinct `suppkey` values in Lineitem, as shown on the graph. To improve performance of MIN, MAX, and theta-joins in general, we plan to extend DBToaster with specialized tree-based data structures.

9.2 Stream scalability

This section analyzes the scaling behavior of DBToaster for a subset of the TPC-H queries over a larger stream of updates. Our focus is on measuring view refresh rates in terms of the stream length and query complexity, rather than the working set size.

The workload for this experiment was synthesized from databases created by DBGEN at scaling factors 0.5, 1, 5, and 10 (500 MB, 1, 5, and 10 GB, respectively). An update stream was built by randomly interleaving tuples from the base relations, while preserving the reference integrity. As before, after inserting 30,000 Orders tuples and 120,000 Lineitem tuples, we randomly inject deletions into these two relations in order to keep their sizes roughly constant. Tuples in other TPC-H relations are never deleted.

The length of the update stream increases with larger scaling factors. However, the size of the working set depends on the query structure. Materialized views that reference Customer, Part, Supplier, or Partsupp might grow with larger scaling factors, while views defined solely over Orders or Lineitem have a bounded working set size.

Figure 12 presents the results of our scaling experiments. For most queries performance stays roughly constant as the stream length grows. Q2 and Q16 select over insert-only relations (Part, Supplier, and Partsupp); thus, the memory overhead of DBToaster grows with the scaling factors. The view refresh rates drop as the maintenance cost for these queries is linear in the size of in-memory data structures. In contrast, Q11a also queries insert-only relations, but exhibits good scaling behavior due to the cardinality constraints between its base relations and use of index data structures. The running time of Q22a is dominated by the first 10 % of the stream in all cases, before the Customer relation has been fully inserted. The cost of inserting a new customer is linear in the size of the Customer relation. After all customer tuples have been processed, the refresh rate increases to a constant 8000 tuples per second, regardless of scale.

Q9 and Q21 demonstrate an increase of the view refresh rates for larger stream lengths. The reason for this behavior is as follows. In our workload, the working set sizes of Orders and Lineitem are constant, regardless of the scaling factor. With larger scaling factors the base relations get larger; thus, we have to place more deletions to maintain the size invariant (As an extreme case, imagine that the working set size of Orders is 1; then we have to double the number of Orders tuples in the stream as every insertion is followed by a deletion). Placing more deletions increases the fraction of Orders and Lineitem tuples in the stream. This in turn affects the view refresh rates of these queries, as both have constant costs with respect to updates to the Lineitem relation.

10 Conclusion

We presented DBToaster, a compiler and optimizer framework for higher-order IVM that uses aggressive simplification of recursive delta queries and a plethora of materialization strategies to make recursive IVM viable. Our compilation method is effective on a wide range of select-project-join-aggregate queries, including those with nested subqueries, which are unsupported by current IVM mechanisms. Our methods provide view refresh rates that often improve on today's tools by several orders of magnitude.

Acknowledgments This work was supported by ERC Grant 279804.

References

1. Abadi, D., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., et al.: The design of the Borealis stream processing engine. In: CIDR, pp. 277–289 (2005)
2. Agrawal, S., Chaudhuri, S., Narasayya, V.R.: Automated selection of materialized views and indexes in SQL databases. In: VLDB, pp. 496–505 (2000)

3. Ahmad, Y., Koch, C.: DBToaster: A SQL compiler for high-performance delta processing in main-memory databases. PVLDB **2**(2), 1566–1569 (2009)
4. Aiken, A., Hellerstein, J.M., Widom, J.: Static analysis techniques for predicting the behavior of active database rules. ACM TODS **20**(1), 3–41 (1995)
5. Aji, S.M., McEliece, R.J.: The generalized distributive law. IEEE Trans. Inf. Theory **46**(2), 325–343 (2000)
6. Blakeley, J.A., Larson, P.Å., Tompa, F.W.: Efficiently updating materialized views. In: SIGMOD, pp. 61–71 (1986)
7. Buneman, P., Clemons, E.K.: Efficiently monitoring relational databases. ACM TODS **4**(3), 368–382 (1979)
8. Buneman, P., Naqvi, S.A., Tannen, V., Wong, L.: Principles of programming with complex objects and collection types. Theor. Comput. Sci. **149**(1), 3–48 (1995)
9. Chaudhuri, S., Krishnamurthy, R., Potamianos, S., Shim, K.: Optimizing queries with materialized views. In: ICDE, pp. 190–200 (1995)
10. Chirkova, R., Yang, J.: Materialized views. Found. Trends Databases **4**(4), 295–405 (2012)
11. Colby, L.S., Griffin, T., Libkin, L., Mumick, I.S., Trickey, H.: Algorithms for deferred view maintenance. In: SIGMOD, pp. 469–480 (1996)
12. Colby, L.S., Kawaguchi, A., Lieuwen, D.F., Mumick, I.S., Ross, K.A.: Supporting multiple view maintenance policies. In: SIGMOD, pp. 405–416 (1997)
13. Cormode, G., Muthukrishnan, S.: What's hot and what's not: tracking most frequent items dynamically. ACM TODS **30**(1), 249–278 (2005)
14. DBToaster Public Beta revision 2827, Feb. 11, 2013. http://www.dbtoaster.org/index.php?page=download
15. Ghanem, T.M., Elmagarmid, A.K., Larson, P.Å., Aref, W.G.: Supporting views in data stream management systems. ACM TODS **35**(1), 1–47 (2010)
16. Griffin, T., Libkin, L.: Incremental maintenance of views with duplicates. In: SIGMOD, pp. 328–339 (1995)
17. Gupta, A., Mumick, I.S., Subrahmanian, V.S.: Maintaining views incrementally. In: SIGMOD, pp. 157–166 (1993)
18. Gupta, H., Mumick, I.S.: Selection of views to materialize in a data warehouse. IEEE TKDE **17**(1), 24–43 (2005)
19. Kawaguchi, A., Lieuwen, D.F., Mumick, I.S., Ross, K.A.: Implementing incremental view maintenance in nested data models. In: DBPL, pp. 202–221 (1997)
20. Kearns, M., Ortiz, L.: The Penn-Lehman automated trading project. IEEE Intell. Syst. **18**(6), 22–31 (2003)
21. Kennedy, O., Ahmad, Y., Koch, C.: DBToaster: Agile views for a dynamic data management system. In: CIDR, pp. 284–295 (2011)
22. Koch, C.: Incremental query evaluation in a ring of databases. In: PODS, pp. 87–98 (2010)
23. Koch, C.: Incremental query evaluation in a ring of databases. Technical Report EPFL-REPORT-183766, https://infoscience.epfl.ch/record/183766 (2013)
24. Koch, C., Ahmad, Y., Kennedy, O., Nikolic, M., Nötzli, A., Lupei, D., Shaikhha, A.: Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views (2013). Technical report EPFL-REPORT-183767, extends this article by an appendix that lists the full query workload as well as experimental parameters and trace figures that did not find space in this article; http://infoscience.epfl.ch/record/183767
25. Kotidis, Y., Roussopoulos, N.: A case for dynamic view management. ACM TODS **26**(4), 388–423 (2001)
26. Krikellas, K., Viglas, S., Cintra, M.: Generating code for holistic query evaluation. In: ICDE (2010)
27. Krishnamurthy, S., Wu, C., Franklin, M.J.: On-the-fly sharing for streamed aggregation. In: SIGMOD, pp. 623–634 (2006)
28. Larson, P.Å., Zhou, J.: Efficient maintenance of materialized outer-join views. In: ICDE, pp. 56–65 (2007)
29. Liu, Y.A., Stoller, S.D., Teitelbaum, T.: Static caching for incremental computation. ACM TOPLAS **20**(3), 546–585 (1998)
30. Marlow, S., Wadler, P.: Deforestation for higher-order functions. In: Functional Programming, pp. 154–165 (1992)
31. Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G.S., Olston, C., Rosenstein, J., Varma, R.: Query processing, approximation, and resource management in a data stream management system. In: CIDR (2003)
32. Neumann, T.: Efficiently compiling efficient query plans for modern hardware. PVLDB **4**(9), 539–550 (2011)
33. Nutanong, S., Carey, N., Ahmad, Y., Szalay, A.S., Woolf, T.B.: Adaptive exploration for large-scale protein analysis in the molecular dynamics database. In: SSDBM, p. 45 (2013)
34. Palpanas, T., Sidle, R., Cochrane, R., Pirahesh, H.: Incremental maintenance for non-distributive aggregate functions. In: VLDB, pp. 802–813 (2002)
35. Pearlmutter, B.A., Siskind, J.M.: Lazy multivariate higher-order forward-mode AD. In: POPL, pp. 155–160 (2007)
36. Ross, K.A., Srivastava, D., Sudarshan, S.: Materialized view maintenance and integrity constraint checking: trading space for time. In: SIGMOD, pp. 447–458 (1996)
37. Roussopoulos, N.: An incremental access method for ViewCache: concept, algorithms, and cost analysis. ACM TODS **16**(3), 535–563 (1991)
38. Salem, K., Beyer, K.S., Cochrane, R., Lindsay, B.G.: How to roll a join: Asynchronous incremental view maintenance. In: SIGMOD, pp. 129–140 (2000)
39. Seshadri, P., Pirahesh, H., Leung, T.C.: Complex query decorrelation. In: ICDE, pp. 450–458. IEEE (1996)
40. Shyamshankar, P., Palmer, Z., Ahmad, Y.: K3: Language design for building multi-platform, domain-specific runtimes. In: International Workshop on Cross-model Language Design and Implementation (XLDI) (2012)
41. Tatbul, N., Çetintemel, U., Zdonik, S.B., Cherniack, M., Stonebraker, M.: Load shedding in a data stream manager. In: VLDB, pp. 309–320 (2003)
42. Transaction Processing Performance Council: TPC-H benchmark specification. http://www.tpc.org/hspec.html (2011)
43. Wong, L.: Kleisli, a functional query system. J. Funct. Program. **10**(1), 19–56 (2000)
44. Yang, J., Widom, J.: Incremental computation and maintenance of temporal aggregates. VLDB J. **12**(3), 262–283 (2003)
45. Zhou, J., Larson, P.Å., Elmongui, H.G.: Lazy maintenance of materialized views. In: VLDB, pp. 231–242 (2007)
46. Zhou, J., Larson, P.Å., Freytag, J.C., Lehner, W.: Efficient exploitation of similar subexpressions for query processing. In: SIGMOD, pp. 533–544 (2007)
47. Zilio, D.C., Zuzarte, C., Lightstone, S., Ma, W., Lohman, G.M., Cochrane, R., Pirahesh, H., Colby, L.S., Gryz, J., Alton, E., Liang, D., Valentin, G.: Recommending materialized views and indexes with IBM DB2 design advisor. In: ICAC, pp. 180–188 (2004)