Multilevel Incremental View Maintenance

Submission #74

ABSTRACT

This paper introduces and studies multilevel incremental view maintenance, an incremental evaluation framework that allows a query optimizer to extend the use of materialized views from answering user queries to also *answering the delta queries* used by a DBMS to refresh views on updates. The aggressive recursive use of this idea can eliminate all joins from certain queries, resulting in query processing with fine-grained in-place updates that refreshes multiple views without classical query operators. This calls for compilation of queries to highly efficient low-level imperative code.

Multilevel incremental view maintenance bears the potential to be faster than classical incremental view maintenance, admitting very high view refresh rates, and to have greater expressive power than query languages supported in stream engines. Extensive experimentation with our compiler as well as existing DBMS and stream engines confirms this. Multilevel incremental view maintenance usually dominates previous continuous querying approaches, frequently by multiple orders of magnitude, for queries involving many joins or nested aggregation. However, managing domains of parameters in auxiliary views can be costly, which turns out to be a problem for certain queries with inequalities.

1. INTRODUCTION

It is immediately plausible that one can do better than re-evaluate a query from scratch whenever the database changes a little. Incremental view maintenance (IVM) capitalizes on this insight [10, 33, 8, 30, 12, 18, 19, 17, 35, 13, 15, 26]. It is a solid, settled technique that has been implemented in many commercial DBMS, but has seen little new research activity in recent years and has gathered a little dust.

Now there is an exciting and potentially game-changing new development [4, 25, 24], an extreme form of IVM where all query evaluation work reduces to adding data (updates or materialized query results) to other materialized query results. No join processing or anything semantically equivalent happens at any stage of processing. This works for a fragment of SQL with equijoins and aggregation, but without inequality joins or nesting aggregates.

Let us digest this, because the last claim goes counter to query processing intuitions to the point of absurdity. The main idea is the following: Classical IVM revolves around the idea that a materialized view can be maintained under updates by evaluating a so-called delta query and adding its result to the materialized view. The delta query captures how the query result changes in response to a change in the database. The new observation is that the delta query can be materialized and incrementally maintained using the same idea, making use of a delta query to the delta query, which again can be materialized and incrementally maintained, and so on, recursively. This works for classes of queries whose deltas are in some way structurally simpler than the base queries (e.g. having fewer joins), allowing this recursive query transformation to terminate. (It does so with a final trivial k-th delta query that does not refer to the database at all.) Termination is ensured for select-project-join queries with certain forms of aggregation, but some other features of SQL (specifically aggregations nested in where-conditions) have to be excluded.

So where do the joins go? They *really* go away, as a benefit of incremental computation. If we want to compute $(x+1)*y$ and know $x * y$ and y, we only need to add y to $x * y$, and the multiplication goes away. This is what happens when incrementally maintaining a join, where the join takes the place of multiplication. The pattern just sketched in basic algebra is not just an intuition but exactly what happens, and [25] develops the algebraic framework to formalize this. We observe that the symbol 1 above represents the update workload. In the incremental query processing framework, it must be a *constant* number of tuples that are changed in each incrementation step.

On paper, this approach 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 for incremental maintenance of the base query also applies to the delta query. Argued from the viewpoint that joins are expensive and this approach eliminates them, one should expect a potential for excellent query performance.

But does this expectation translate into real performance gains? A priori, the cost of the bulk addition of materialized views or the costs associated with storing and managing additional auxiliary materialized views (for delta queries) might be more considerable than expected.

This paper presents the lessons learned in an effort to realize recursive IVM, spanning nearly three years of intense work, to generalize it to be applicable to most of SQL, and to understand its strengths and drawbacks.

The contributions are as follows.

- Multilevel IVM bears the promise of providing materialized views of complex SQL queries, without window semantics or other restrictions, at very high refresh rates. We start by arguing that there is a need for such functionality, creating a benchmark consisting of automated trading and ETL workloads. We show that state of the art systems cannot deliver materialized views refreshed at the rates that some application domains (algorithmic trading, real-time analytics) require. This is the challenge we set ourselves in this paper.

- We develop the vision of multilevel IVM further into a workable system. While the techniques of [4, 25] as well as existing implementations of IVM in commercial DBMS are very restricted and exclude nested queries and other features of SQL, we create the machinery to perform IVM and even recursive IVM on most of SQL (with the exception of support for null values).[1] To do this, we generalize the techniques of [4, 25] to not always materialize full delta queries but instead subexpressions that allow us to perform IVM and maximize the performance obtained. This leads us to a query optimizer in which the materialization of subqueries is a degree of freedom in optimization, and can be applied anywhere in the input query or the delta queries obtained by applying this optimization.

  To put ourselves in the position of using such an optimizer, we have to create suitable intermediate representations (IR) of queries that support binding patterns for sideways information passing. We study when and how to efficiently initialize views, and present query decomposition and factorization techniques that lead to efficient formulations of update triggers that refresh our views.

- Multilevel IVM eliminates most query operators and yields low-level update trigger programs for view refreshing that have simple and regular structure. It seems natural to exploit this, and to follow an approach of *compiling* the view refreshing code down to machine code, rather than to interpret in a query execution engine. We present our approach to achieving this, which makes use of sophisticated deforestation and fusion techniques from the compilers literature. These optimizations in general have no expression in the high-level relational algebra-like IR we use for multilevel IVM rewriting, so we introduce lower-level functional IR for this purpose.

- We have implemented our compiler and performed extensive experimentation with it. Our experiments indicate that frequently, particularly for queries that consist of many joins of nested aggregation subqueries that are not correlated through subqueries, our compilation approach dominates the state of the art, often by multiple orders of magnitude. There are also queries in our benchmark on which our techniques do not fare well; these usually involve the creation and maintenance of huge auxiliary views whose data is rarely used by other views. These scenarios could be much improved upon by suitable garbage collection strategies on auxiliary views. This is future work, and we consider it likely that once such a technique has been integrated into our compiler, it will outperform the state-of-the-art on an even wider range of queries.

[1]Most of our benchmark queries contain features such as nested subqueries that none of the commercial implementations of IVM support, while our approach supports them all. Our delta processing techniques are relevant to classical IVM in their own right, not just to multilevel IVM.

The structure of the paper follows the order of contribution just laid out.

2. MOTIVATING APPLICATIONS AND STATE-OF-THE-ART SYSTEMS

Multilevel IVM facilitates high-performance, lightweight monitoring for a wide range of applications. Consider the following uses:

1. An automated stock market trading system monitors buy and sell order books of a particular stock to identify the best time and price for its own orders. The key analyses for making such decisions can frequently be performed via SQL views, which however have to be fresh at very high rates (about 500 times a second for high-volume stocks) to be effective.
2. A corporate data warehouse monitors production facilities, warehoused inventory and active demand for products in order to identify supply chain problems. If data warehouse loading can be performed in real time, the usefulness of the warehouse is leveraged.

These uses require frontend analysis systems to drive rapid decision-making in contrast to offline data warehouses for deep exploratory querying. While the need for high-performance monitoring and view refresh is abundantly clear in algorithmic trading, advertising, security and disaster prediction [32], timely response is emerging as an essential ingredient of competitive advantage and operational efficiencies in business applications, and benefits other domains such as regulatory compliance [7] and scientific simulations [20]. To the best of our knowledge, no data management systems have focused on highly-efficient continuous SQL-based analysis and aggregation of large datasets that change in small high-impact increments.

Query Workloads for Monitoring. The above scenarios of algorithmic trading on order books and online business decision support often involve computing a variety of statistics, and comparing and contrasting them prior to taking action. Figure 1 lists the processing properties of a query workload inspired by these domains that we use in this paper, with SQL code in Appendix A.

The algorithmic trading queries operate on the bid and ask order book streams of a day's worth of orders for the MSFT symbol on NASDAQ (approx. 2.5 million orders). The online decision support queries are taken from the TPC-H and SSB benchmarks, and processed over a randomized, replayed update stream. Our workload has a range of joins, predicates, and correlated subqueries, up to depth 2. While a depth of 2 may seem low, we have not seen many queries in popular DBMS benchmarks have more levels.

The pertinent DBMS methods are triggers, and stream systems. We benchmarked these queries on a mix of open-source and commercial stream systems and DBMS (Esper, Postgres, CSPE, CDB)[2], Figure 2 shows their view refresh performance. We do not include any native IVM results from commercial DBMS because despite its wide availability, there are still many limitations on its usability. In fact, none of the major DBMS supported incremental refresh for the majority of our workload given the features in Figure 1. For many DBMS, the documented query restrictions [22, 1, 2] indicate that view materialization and maintenance cannot generally be applied side-by-side with query optimization.

[2]We anonymize the commercial systems due to licensing restrictions on publishing benchmarks.

	Query	# Tables, join type. =: equi x: cross	Where-clause	Group-bys	# Subqueries and depth
Finance	AXF	2, =	$\vee, <$	yes	0 / 0
	BSP	2, =	$\wedge, <$	yes	0 / 0
	BSV	2, =	None	no	0 / 0
	MST	2, x	$\wedge, <$	yes	2 / 1
	PSP	2, x	$\wedge, <$	no	2 / 1
	VWAP	1	$<$	no	2 / 1
ETL	TPCH3	3, =	$\wedge, <$	yes	0 / 0
	TPCH11	2, =	None	yes	0 / 0
	TPCH17	2, =	$<$	no	1 / 1
	TPCH18	3, =	$<$	yes	1 / 2
	TPCH22	1	$=, <$	yes	2 / 1
	SSB4	7, =	$<$	yes	0 / 0

Figure 1: Features of the algorithmic trading, online decision support, and cluster monitoring query workload used for experiments.

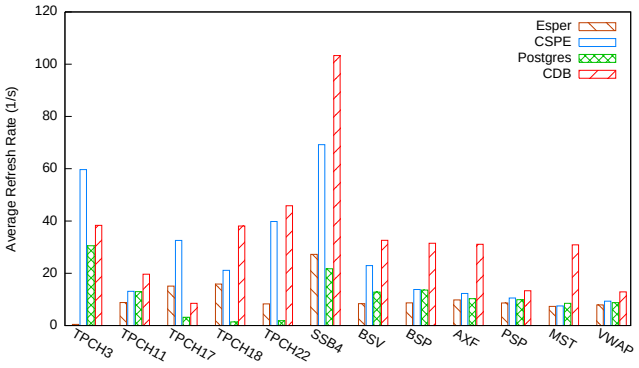


Figure 2: A comparison of four stream systems and DBMS on full refresh view maintenance.

Triggers and Active Databases. Trigger mechanisms facilitate the manipulation of database state in a reactive manner, as DML statements execute. The literature on triggers has focused on issues such as cascading and recursion, and to a lesser extent single-level IVM [6]. Triggers can be implemented in a variety of languages from plpgsql, to Python or C. For procedural SQL languages, triggers are compiled in the same way as user-defined queries for interpretation by an executor. This executor is designed for set-at-a-time computation rather than fine-grained, pipelined computation.

We implemented a *repetitive* full refresh approach to maintain views for our query workload, where after each update a trigger re-evaluates the query from scratch. Figure 2 shows that for the small-state finance workload, trigger performance does not vary across the types of queries. Here small-state refers to the state accumulated while processing updates. In the finance workload there are nearly as many deletes as inserts (unsuccessful orders are often removed from the order books), and relations do not grow too large. For large-state workloads, on Postgres, queries with nesting perform signficantly worse than on CDB. Overall, CDB performs the best across all engines, primarily with the mature optimizer in CDB choosing better plans and index usage.

Stream and Complex Event Processing. These systems (SPEs) provide low-latency push-based processing and at first glance may appear to be ideal for frontend analytics. However, monitoring applications must often work with long-lived, large state, and this requirement is not well-addressed by current SPEs. They either rely on in-memory tables with suboptimal engines or external DBMS access, and do not support incremental computation over large state.

Our implementation varied significantly across Esper and CSPE, due to a clear lack of standardized semantics and portability across SPEs [9, 23]. In both Esper and CSPE, we maintained indexed in-memory tables to represent base relations, due to the inapplicability of windows to handle arbitrary updates and deletes. For Esper, we were unable to use subqueries in stream-to-table join operations, which despite being supported in the language, resulted in differing, inconsistent snapshots of tables that were used multiple times in a query (e.g. self-joins), similar in effect to the state bug from views [13]. CSPE on the other hand did not support subqueries in stream-to-table operations.

On both systems, our approach performed stream-to-table joins to trigger the replay of tables as streams via scans, as updates arrive. The replayed tables were then processed by stream-only operators to implement fully pipelined plans. Additionally, we explicitly restricted both Esper and CSPE to single-threaded execution and relied on their deterministic tuple processing order to avoid complex locking and barriering to implement aggregate subqueries. We found such serialization added substantial programming complexity and performance overheads to our implementation.

Figure 2 shows that for the finance workload, there is little variation between Esper and CSPE, although CSPE does consistently outperform Esper, but cannot match triggers in CDB. For the large-state queries, CSPE outperforms Esper by significantly, and provides a viable alternative to triggers in CDB with an overall lower variance in performance across both nested (TPCH 17, 18, 22) and flat queries. Esper performs poorly on TPCH 3 by missing index usage.

The takeaway from these experiments is that with complex analysis workloads involving nested queries and aggregate comparisons, SPEs often cannot take advantage of the semantics they were designed for (e.g. windows, patterns), and do not offer advantages over trigger processing. Furthermore neither system achieves a high absolute refresh rate, of more than roughly 100 refreshes a second suggesting limited responsiveness and poor scaling when considering many simultaneous queries.

3. THE COMPILER

In this section we present our compiler. We start by sketching its architecture. The compiler accepts an SQL query and schema definition (essentially create table statements for the relations the query accesses) as input. The parser turns the input query into a first intermediate representation (IR) that is an adapted form of relational algebra. The first compilation stage generates trigger programs for performing multilevel incremental view maintenance (IVM). The statements of these trigger programs increment the values of view data structures (called maps) by expressions of this IR. The IR and basics of the transformation are described in Section 3.1. Section 3.2 presents optimizations for simplifying IRs in the first compilation stage.

In Section 3.3 we discuss the creation of triggers for multilevel IVM by extracting and materializing strict subexpressions rather than the full query. This is necessary to be able to deal with nested aggregation which otherwise, if we always materialize the largest query we can, leads to recursion in compilation that does not terminate. In general, the choice of subquery to materialize and incrementally maintain is a degree of freedom in query optimization. In this section we give heuristic rules for making this choice.

The next stage of the compiler, described in Section 3.4, turns the statements of the trigger program created in the

first stage into an IR for purely functional programming, essentially lambda calculus without general recursion but with special higher-order functions for transforming collections (primitives such as map and reduce). This stage allows us to perform deforestation and fusion optimizations too low-level to have expression in the first, relational algebra-like IR. Section 3.4 also describes our infrastructure for code generation and our runtime system.

3.1 Queries with binding patterns, deltas, and recursive IVM

Next we develop an IR for SQL queries that is suitable for compilation of database queries and to perform the transformations necessary to enable efficient multilevel IVM. This language is a refinement of the query language defined in [25], but here we aim at better readability and avoid unnecessary formality. The language is based on positive relational algebra with aggregation but makes the following modifications:

- The data model is that of relations where tuples have *integer* multiplicities. This generalizes SQL bag semantics to allow for negative multiplicities. This way, databases and updates can be treated uniformly. A relation in which all multiplicities are nonnegative can be either thought of as an insertion or as a database (=an insertion into the empty database) and a relation in which all tuple multiplicities are negative is a deletion.
- We replace relational (bag) union by an operation $+$ that adds two relations R and S by assigning to each tuple in the result the integer sum of the multiplicities of that tuple in R and S.
- The join operation is the natural join, where the multiplicity of a tuple in the result is the integer product of the multiplicities of the two tuples from which it is formed.
- Conditions are queries in their own right; there is no explicit selection operation. Thus, we write a relational algebra selection $\sigma_{A<B}(R)$ as $R \bowtie (A < B)$.
- Sum-aggregates are of the form $\text{Sum}_{\vec{A}}Q$ where Q is a query and $\vec{A}$ is a tuple of group-by columns (which we will also call variables). The result are 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. Sum-aggregates can be viewed as multiplicity-respecting projection.
- Terms have a scalar rather than relation value. A sum-aggregate without grouping column evaluates to a term. Terms can be used as queries; in that case, they evaluate to the nullary tuple with multiplicity the value of the term. Thus an SQL query "select sum(A) from R" can be written in our IR as $\text{Sum}(R \bowtie A)$; we first multiply the value of A into the multiplicity of each tuple and then sum up all these multiplicities.
- There are no explicit operations for projection and relational difference. A multiplicity-aware projection is implemented by sum-aggregation and universal queries can be expressed by counting aggregation (a popular homework exercise in database courses). All SQL aggregate functions can be expressed using sums (counts), ratios of sums (avg), and nested aggregates (min and max)[3].

[3]Here, the materialization of the nested subquery nicely corresponds to the materialization any IVM technique has to carry out to find second-best values when a min or max value is deleted.

Let us recap: The language departs from relational bag algebra in essentially two ways; by having integer tuple multiplicities to deal uniformly with insertions and deletions, and by *taking aggregate values out of the tuple* and putting them down into the multiplicity. This has a very desirable consequence: Incremental computation is all about modifying multiplicities. As we will see later, by keeping aggregate values in the multiplicities, delta processing for aggregates is vastly simplified. Aggregate queries dominate analytical workloads and can greatly profit from incremental view maintenance. Thus it makes sense to optimize our IR for aggregate processing.

In summary, the abstract syntax of the IR is two-sorted (consisting of queries q and terms t):

$$\begin{aligned} q &:= R \mid t\ \theta\ t \mid t \mid q + q \mid q \bowtie q \mid \text{Sum}_{\vec{A}}(q) \\ t &:= c \mid A \mid t + t \mid t * t \mid \text{Sum}(q) \end{aligned}$$

Thus queries can be formed from relation names, atomic conditions, addition, natural join, and sum-aggregation. Terms are formed from constants, variables/columns, addition, multiplication, and sum-aggregation without group-by. Terms are queries returning the nullary relation in which tuple $\langle\rangle$ has as multiplicity the value of the term. We use $Q_1 - Q_2$ as syntactic sugar for $Q_1 + (-1) \bowtie Q_2$. Apart from this, the semantics was given above.

We will often use SQL syntax for conciseness, which will correspond to our IR in the natural way analogous to how we translate between classical bag relational algebra and SQL.

Expressions of the IR have binding patterns: There are input variables or parameters without which we cannot evaluate these expressions, and there are output variables, the columns of the schema of the query result. Each expression Q has input variables or parameters $\vec{x_{in}}$ and a set of output variables $\vec{x_{out}}$, which form the schema of the query result. We denote such an expression as $Q[\vec{x_{in}}][\vec{x_{out}}]$. The input variables are those that are not *range-restricted* in a calculus formulation, or equivalently have to be understood as *parameters* in a SQL query because their values cannot be computed from the database: They have to be provided so that the query can be evaluated.

Binding patterns represent information flow. In general, this flow is not exclusively bottom-up. Some of an expression's variables are input variables or parameters which cannot be computed from the query but have to be given to the query so that it can be evaluated. The most interesting case of this is a correlated nested aggregate, viewed in isolation (which must be possible for small-step compositionality). In such an aggregate, the correlation variable from the outside is such an input variable. The aggregate query can only be computed if a value for the input variable is given.

We illustrate this by a few examples in Table 1. All of the expressions there are valid queries of our IR, typed by indicating the input and output variables.

This language specification covers all of the core features of SQL with the exception of null values and outer joins.

3.1.1 Computing the delta of a query.

This language has the nice property of being *closed under taking deltas*. For each query expression Q, there is an expression ΔQ of our IR that expresses how the result of Q changes as the database D is changed by update workload ΔD. This works for updates containing an unbounded number of insertions and deletions, but we will subsequently only study single-tuple inserts and deletes. This is because such

SQL	IR	Bdg.pat.
select * from R	R	$[][A,B]$
—	$C < D$	$[C,D][]$
—	$C = D$	$[C][D]$ or $[D][C]$
select A from R where B < C	$\mathrm{Sum}_A(R \bowtie (B < C))$	$[C][A]$
select A, sum(1) from R where B < C group by A	$\mathrm{Sum}_A(R \bowtie (B < C))$	$[C][A]$
B < (select sum(D) from S where A > C)	$B < \mathrm{Sum}(S \bowtie D \bowtie (A > C))$	$[A,B][]$
select * from R where B < (select sum(D) from S where A > C)	$\mathrm{Sum}_{A,B}(R \bowtie B < \mathrm{Sum}(S \bowtie D \bowtie (A > C)))$	$[][A,B]$

Table 1: Example queries with their binding patterns (written as [input vars][output vars]). Relation R and S have schema A, B and C, D, respectively.

updates allow for particularly efficient view refresh code that can be run to respond online to each tuple change.

Thanks to the strong compositionality of the language, we only have to give delta rules for the individual operators. These rules are given and studied in detail in [25]. In short,

$$\begin{aligned}\Delta(Q_1 + Q_2) &:= (\Delta Q_1) + (\Delta Q_2),\\ \Delta(\mathrm{Sum}\ Q) &:= \mathrm{Sum}\ (\Delta Q),\\ \Delta(Q_1 \bowtie Q_2) &:= ((\Delta Q_1) \bowtie Q_2) + (Q_1 \bowtie (\Delta Q_2))\\ &\quad + ((\Delta Q_1) \bowtie (\Delta Q_2)).\end{aligned}$$

ΔR is the update to R. In the case that the update does not change R (but other relation(s)), ΔR is empty.

The deltas of conditions are 0 if they do not contain nested queries. The delta for a condition with a nested query is more complicated. For example, our rule for $\Delta(x > Q)$ is $(x > (Q + \Delta Q)) \bowtie (x \le Q) - (x \le (Q + \Delta Q)) \bowtie (x > Q)$. We refer to [25] for the general case of conditions.

Let us be precise though about computing delta queries. In general, each expression has input and output variables, and taking a delta in general *adds variables* parameterizing the query with the update. From now on we only consider single-tuple insertions $+R(\vec{x})$ into or deletions $-R(\vec{x})$ from a relation R.

Example 3.1. Given schema $R(AB), S(CD)$, and query

```
select sum(A * D) from R, S where B < C
```

or, in our IR, $\mathrm{Sum}(R \bowtie S \bowtie B < C \bowtie A * D)[][]$.

The delta for this query and the insertion of a tuple $\langle x, y\rangle$ into R is as follows. Let e be $S \bowtie B < C \bowtie A * D$. Then

$$\Delta_{+R(x,y)}\mathrm{Sum}(R \bowtie e) = \mathrm{Sum}\Delta_{+R(x,y)}(R \bowtie e)$$

and by the delta rule for $\bowtie$,

$$\begin{aligned}\Delta_{+R(x,y)}(R \bowtie e) &= (\Delta_{+R(x,y)}R) \bowtie e\\ &+ R \bowtie \Delta_{+R(x,y)}e\\ &+ (\Delta_{+R(x,y)}R) \bowtie \Delta_{+R(x,y)}e\end{aligned}$$

and $\Delta_{+R(x,y)}(e) = 0$ since $\Delta_{+R(x,y)}S = 0$, $\Delta_{+R(x,y)}B\theta C = 0$, and $\Delta_{+R(x,y)}(A * D) = 0$. (This follows again from the delta rule for $\bowtie$, applied twice.) Now $\Delta_{+R(x,y)}R$ is the singleton relation $\{\langle A : x, B : y\rangle\}$: the actual insertion. The expression $\{\langle A : x, B : y\rangle\} \bowtie e$ can be simplified to $\{\langle A : x\rangle\} \bowtie S \bowtie y\theta C \bowtie x * D$. Thus the delta of the input query is $\mathrm{Sum}(\{\langle A : x\rangle\} \bowtie S \bowtie y < C \bowtie x * D)[x, y][]$ which can be further simplified to $\mathrm{Sum}(S \bowtie y < C \bowtie x * D)[x, y][]$. In SQL, this is: `select sum(x * D) from S where y < C`.

3.1.2 Recursive incremental view maintenance.

Before we come to multilevel IVM, let us recap the special case of recursive IVM of [25, 24], where we try to be as greedily incremental as possible. If we restrict the query language to exclude aggregates nested into conditions (for which the delta query was complicated), the 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. We can use distributivity to push unions above joins and so give a degree to queries with unions, as 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.

Theorem 3.1 ([25]). *If $deg(Q) > 0$, then*

$$deg(\Delta Q) = deg(Q) - 1.$$

Recursive IVM makes use of the simple fact that a delta query is a query too. Thus it can be incrementally maintained, making use of a delta query to the delta query, which again can be materialized and incrementally maintained, and so on, recursively. By the above theorem, this recursive query transformation terminates in the $\deg(Q)$-th recursion level, when the rewritten query becomes a "constant" (independent of the database, and dependent only on updates).

The goal of compilation is to create on-insert and on-delete triggers for every relation occurring in the query. Conceptually, a query $Q[\vec{x}_{in}][\vec{x}_{out}]$ over relations $R_1, \ldots, R_k$ is compiled as follows: We write M_Q for the materialized view of query Q. Then the trigger program for the event $\pm R_{i_{j+1}}(\vec{y}_{j+1})$ consists of the statements

foreach $\vec{x}_{out}, \vec{y}_1, \ldots, \vec{y}_j$ do
$$M_{\Delta_{s_{i_j}R_{i_j}}\cdots\Delta_{s_{i_1}R_{i_1}}Q}[\vec{x}_{in}\vec{y}_1\ldots\vec{y}_j][\vec{x}_{out}] \pm =$$
$$M_{\Delta_{s_{i_{j+1}}R_{i_{j+1}}}\cdots\Delta_{s_{i_1}R_{i_1}}Q}[\vec{x}_{in}\vec{y}_1\ldots\vec{y}_{j+1}][\vec{x}_{out}].$$

for each $j \in 0, 1, \ldots, \deg(Q)$, $\langle i_1 \ldots i_{j+1}\rangle \in \{1, \ldots, k\}^{j+1}$, and $s_{i_l} \in \{+, -\}$. For correctness, unless we maintain old and new versions of the maps (which would be costly), these statements have to be ordered by increasing j.

Example 3.2. We return to the query Q of Example 3.1, and $(\Delta_{\pm R(x,y)}Q)[x, y][]$ which was already computed there. We compute:

$$(\Delta_{\pm S(z,u)}Q)[z, u][] = \mathrm{Sum}(R \bowtie B < z \bowtie A * u)$$

and

$$\begin{aligned}(\Delta_{\pm S(z,u)}\Delta_{\pm R(x,y)}Q)[x, y, z, u][] &=\\ (\Delta_{\pm R(x,y)}\Delta_{\pm S(z,u)}Q)[x, y, z, u][] &= \mathrm{Sum}(y < z \bowtie x * u)\\ &= (y < z)\ ?\ (x * u) : 0\end{aligned}$$

where $(y < z)\ ?\ (x * u) : 0$ is the functional if-then-else of C (if $y < z$ then $x * u$ else 0). The on-insert into R trigger $+R(x, y)$ following the construction described above is the program

$M_Q[][]$ += $M_{\Delta_{+R(x,y)}Q}[x, y][]$;
foreach z, u do $M_{\Delta_{+S(z,u)}Q}[z, u][]$ += $(y < z)\ ?\ (x * u) : 0$;
foreach z, u do $M_{\Delta_{-S(z,u)}Q}[z, u][]$ += $(y < z)\ ?\ (x * u) : 0$

The remaining triggers are constructed analogously. The trigger contains an update rule for the (in this case, scalar)

data structure M_Q for the overall query result, which uses the auxiliary view $M_{\Delta_{\pm R(x,y)}Q}$ which is maintained in the update triggers for S, plus update rules for the auxiliary views $M_{\Delta_{\pm S(z,u)}Q}$ that are used to update M_Q in the insertion and deletion triggers on updates to S.

We observe that the structure of the work that needs to be done is extremely regular and (conceptually) simple. Moreover, there are no classical large-granularity operators left, so it does not make 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 adopting a compilers approach.

3.1.3 Initial value computation

The technique for constructing trigger programs by recursive delta rewriting discussed above assumes materialized views to be represented as maps M that store, for tuples $\vec{i}\vec{o}$ of input and output variables, a value (an aggregation result) $M[\vec{i}][\vec{o}]$. So far we have disregarded the question what the *domains* of these maps are, that is, for which $\vec{i}\vec{o}$ the maps are defined and store values. If we could assume that the domains contain all the tuples that could ever be formed from values occurring in the database, we could preallocate map values of zero, $M[\vec{i}][\vec{o}] := 0$. However, it is not feasible to assume such large or even infinite domains. Instead, we want to start with empty maps and extend the domains when the update triggers want to access values – for reading or writing – that are not yet represented in the map. This raises the question how to compute initialization values; in general, these will not be zero if the system has been running for a while and the maps contain data.

Take for instance the query of Example 3.2: Suppose we have only seen inserts into relation R, but S is still empty. Then the auxiliary maps $M_{\Delta_{\pm S(z,u)}Q}[z,u][]$ still have empty domains and looping over each pair $\langle z,u\rangle$ does nothing. When, later, a tuple $\langle z,u\rangle$ is inserted into S, M_Q is incremented by $M_{\Delta_{+S(z,u)}Q}[z,u][]$, which has to be computed from scratch as $\mathrm{Sum}(R \bowtie B < z \bowtie A * u)$ and will generally be nonzero.

In the current implementation of our compiler, we use the fact that if query Q does not use inequality joins, a value to be initialized in map M_Q will be zero. Otherwise, the value is computed from scratch by executing Q. In the future, this could be improved upon by further compiling subqueries of Q using recursive IVM techniques.

3.2 Query decomposition and factorization

The construction for trigger programs presented above yields materializations of high dimensionality, which will be very large and expensive to maintain. This can be avoided by being slightly less aggressive with materialization. We next discuss two query simplification techniques that are key to making recursive IVM useful, join graph decomposition and factorization of query polynomials.

Join graph decomposition makes use of the generalized distributive law [5] (which plays an important role for probabilistic inference with graphical models) to decompose sum-aggregates over disconnected join graphs. This is an important case, since computing deltas eliminates hyperedges of the join graph, disconnecting it more and more in each delta-rewriting, until all hyperedges are gone. The basic law is simple. If a query is of the form $Q_1 \times Q_2$, i.e., Q_1 and Q_2 do not share common columns, then $\mathrm{Sum}_{\vec{A}\vec{B}}(Q_1 \times Q_2) = \mathrm{Sum}_{\vec{A}}(Q_2) \times \mathrm{Sum}\vec{B}(Q_2)$ where $\vec{A}$ are columns of Q_1 and $\vec{B}$ are columns of Q_2. Consider for example the query Q

```
select sum(A*D) from R, S, T
where R.B=S.B and S.C=T.C
```

of schema $R(AB), S(BC), T(CDE)$. Then $\Delta_{+S(b,c)}Q$ is

```
select sum(A*D) from R, T where B=b and C=c
   = (select sum(A) from R where B=b) *
     (select sum(D) from T where C=c)
```

Given such a decomposition, it is better to materialize the two aggregate subexpressions and compute the product when updating, rather than to materialize the product, that is,

$$\begin{aligned} M_Q[][] \quad += \quad & M_{\text{select sum(A) from R where B=b}}[b][] \\ * \quad & M_{\text{select sum(D) from T where C=c}}[c][] \end{aligned}$$

is better than $M_Q[][] += M_{\Delta_{+S(b,c)}Q}[b,c][]$ because that way we materialize two (small) one-dimensional maps rather than one two-dimensional map.

Binary sums (unions) + can be pushed through aggregate sums ($\mathrm{Sum}(Q_1 + Q_2) = \mathrm{Sum}(Q_1) + \mathrm{Sum}(Q_2)$). In conjunction with join graph decomposition, this often leaves us with expressions that do a fair amount of adding and multiplying of subqueries. Such expressions can be optimized using the distributive law. Factorization is often particularly useful to share common subexpressions. For example, the general delta rule for $Q_1 \bowtie Q_2$ is a sum of three subexpressions $\Delta(Q_1 \bowtie Q_2) := ((\Delta Q_1) \bowtie Q_2) + (Q_1 \bowtie (\Delta Q_2)) + ((\Delta Q_1) \bowtie (\Delta Q_2))$ which can be factorized as $((Q_1 + \Delta Q_1) \bowtie (\Delta Q_2)) + ((\Delta Q_1) \bowtie Q_2)$ or as $((\Delta Q_1) \bowtie (Q_2 + \Delta Q_2)) + (Q_1 \bowtie \Delta Q_2)$. Determining which one is better is a task for a cost-based optimizer.

3.3 Materialization heuristics

We have observed experimentally (see Section 4) that under certain conditions, a less-aggressive maintenance strategy is beneficial. In this subsection, we describe challenges that we have encountered while implementing aggressive materialization, and provide heuristics for addressing them.

Nested aggregate queries. Aggregate subqueries appearing as arguments to comparison operators cannot be recursively materialized by aggressively maintaining the entire delta of the comparison expression – the delta expression is not strictly simpler than the original expression. Materializing the nested subquery rather than the delta of the comparison expression eliminates this issue. We illustrate the idea with a simple example.

EXAMPLE 3.3. Consider the following query Q over relations $R(A)$ and $S()$, and a nested aggregate query $\mathrm{Sum}(S)$

$$Q = R \bowtie (A < \mathrm{Sum}(S))$$

Applying the rules of [25] with respect to S yields the delta:

$$\begin{aligned} \Delta_S Q \quad = \quad & (R \bowtie (A \geq \mathrm{Sum}(S)) \bowtie (A < (\mathrm{Sum}(S) + 1))) \\ - \quad & (R \bowtie (A < \mathrm{Sum}(S)) \bowtie (A \geq (\mathrm{Sum}(S) + 1))) \end{aligned}$$

The aggregate appears in the delta expression, and so $\Delta_S Q$ is not structurally simpler than Q – maintaining $\Delta_S Q$ requires more work than evaluating the original query.

Instead, we subdivide this expression into three components: (a) the part of $\Delta_S Q$ independent of the nested aggregate (R), (b) the nested aggregate ($\mathrm{Sum}(S)$), and (c) the delta of the nested aggregate ($\Delta_S \mathrm{Sum}(S) \equiv 1$). Each component is individually simpler, and thus safe to materialize independently. An iteration over component (a) is required

for every insertion into S in order to apply the inequality predicate, but is less expensive than the original query.

This three-way subdivision can be applied to *any* nested aggregate delta to create a safe materialization strategy.

Input Variables. Maintaining a view with input variables can be expensive – the domain of the variable is generally infinite. When a value of this variable is encountered for the first time, the view expression must be evaluated as discussed in Section 3.1. The materialized view acts as a cache that incrementally maintains cached values rather than invalidating them.

As with caches, there is a tradeoff between the utility of caching values and the cost of doing so. In the case of materialized views, this is not only the memory cost, but also the cost of maintaining stored values – each of which is a materialized view in its own right. The amount of maintenance work required is proportional to both the domain of the input variable (the size of the cache), and the amount of work required to maintain any individual currying of the materialized view. Instead, terms where the variable appears can be pulled out of the materialization and applied as filters when the expression is evaluated.

EXAMPLE 3.4. The expression $Q = Sum_A(R \bowtie (B < D))$ over relation $R(A, B, C)$ can be materialized in two ways[4]:

$$\begin{aligned} M_{full}[D][A] &= \mathrm{Sum}_A(R \bowtie (B < D)) \\ M_{part}[][A, B] &= \mathrm{Sum}_{A,B}(R) \end{aligned}$$

Maintaining M_{part} requires only a constant amount of work per insertion, while maintaining M_{full} involves work proportional to the size of the domain of D currently being maintained.

If M_{full} already contains the cached result for a given D, then evaluating $Q(D)$ only requires a lookup. If not, the expression must be computed in its entirety. Using M_{part} to evaluate $Q(D)$ requires first applying the predicate $(B < D)$ to all $\langle A, B\rangle \in M_{part}$. Note that in lieu of the base relations, M_{part} can be used to compute initial values for M_{full}.

We can compute the relative costs of these two approaches in terms of the following variables:

$$\begin{aligned} rate_X &: \text{The rate of updates or evaluations of } X \\ p(\exists X) &: \text{The chance of } X \text{ already being cached.} \\ dom_R(X) &: \text{The domain of values of } X \text{ in } R \\ cost_{full} &= rate_Q\left[p(\exists D) + (1 - p(\exists D)) \cdot |dom_R(AB)|\right] \\ &\quad + rate_{+R}\left[|dom_Q(D)| \cdot |dom_R(A)| + 1\right] \\ cost_{part} &= rate_Q |dom_R(AB)| + rate_{+R} \end{aligned}$$

The cost of maintaining M_{full} is based on the Cartesian product of the domains of A and D, while the cost of maintaining M_{part} is based on only those pairs $\langle A, B\rangle$ that actually appear in R. Assuming comparable rates and domain sizes, it is better not to materialize the full view.

3.4 Functional compilation and optimization

To narrow the gap while translating the SQL IR directly to low-level code, our second stage IR is a small functional programming language. We perform holistic query optimization in a functional IR, exploiting a breadth of powerful program transformations including monad transformations from structural recursion [11], deforestation [28], supercompilation and fusion. Other recent works have observed the need for holistic optimization of query executors with low-level languages [27, 29]. Our functional IR is a work-in-progress that we believe will bring benefits such as simpler optimizer development, and non-first normal forms for specialized in-memory or disk layout strategies in future work.

Our functional IR includes tuples and collection types, lambdas and associative lambdas[5], conditionals, four higher-order collection transformers that we describe below, and in-place update operations on collections. Also, our functions are not recursive, and only our collection transformers use higher-order functions, that is we never pass a function as an argument to another function. Our functional IR is already low-level in the spirit of functional compiler internal IRs (i.e. after defunctionalization).

For our higher-order collection transformers, we extend the `map` and `flatten` constructs from Kleisli [11] with aggregations: `agg` and `groupagg`. Their type signatures are:

$$\begin{aligned} &\texttt{map} &&: (\sigma \to \tau) \to \sigma\ \mathcal{C} \to \tau\ \mathcal{C} \\ &\texttt{flatten} &&: (\sigma\ \mathcal{C})\ \mathcal{C} \to \sigma\ \mathcal{C} \\ &\texttt{agg} &&: (\sigma \to \tau \to \tau) \to \tau \to \sigma\ \mathcal{C} \to \tau \\ &\texttt{groupagg} &&: (\sigma \to \tau \to \tau) \to (\sigma \to \upsilon) \to \tau \to \sigma\ \mathcal{C} \to \langle \upsilon, \tau \rangle\ \mathcal{C} \end{aligned}$$

Above $\sigma\ \mathcal{C}$ indicates the type of a collection containing elements of type σ, and $\langle\rangle$ is a tuple constructor. The `map` transformer applies a function to every element of a collection, while `flatten` reduces a two-level nested collection to a flat collection. The `agg` and `groupagg` transformers applies the accumulator function given as its first argument over a collection. Both are given an initial value, and `groupagg` is additionally given a grouping or partitioning function as its second argument. Our framework is agnostic to the form of collection, which may be sets, bags, and lists, and can vary in their underlying implementation (e.g. tree or hash sets, linked lists or vectors). Intelligent datastructure selection is future work, we currently use vectors and hashtables.

Our functional IR supports several optimizations that are not captured by the SQL IR, including:

If-lifting. This is a transformation to a canonical form for our functional IR that involves lifting every conditional to its minimal binding point. Consider an expression:

$$\texttt{apply}(\lambda x.\texttt{apply}(\lambda y.\ \texttt{if}\ x < 0\ \texttt{then}\ y/a\ \texttt{else}\ y{*}3, a), b)$$

The conditional above is independent of y and we can lift to:

$$\begin{aligned} &\texttt{apply}(\lambda x.\ \texttt{if}\ x < 0\ \ \texttt{then apply}(\lambda y.y/a, a) \\ &\qquad\qquad\qquad\qquad\ \ \texttt{else apply}(\lambda y.y * 3, a),\ b) \end{aligned}$$

This canonical form admits maximal optimization of the conditional's then-else blocks by pushing in expressions from outside the conditional at the expense of code size. However, queries are often small in code size, and if-lifting enables joint optimization of IVC expressions and delta expressions.

Deforestation and fusion. This transformation converts functional programs to treeless forms [28], that is, it eliminates redundant intermediate datastructures. The core subset of our deforestation rewrite rules are:

$$\begin{aligned} &\texttt{map}(f, \texttt{map}(f', c)) &&\texttt{:- map}(f \circ f', c) \\ &\texttt{map}(f, \texttt{flatten}(c)) &&\texttt{:- flatten}(\texttt{map}(\lambda c'.\ \texttt{map}(f, c'), c)) \\ &\texttt{agg}(f^a, i, \texttt{map}(f', c)) &&\texttt{:- agg}(f^a \circ f', i, c) \\ &\texttt{agg}(f^a, i, \texttt{flatten}(c)) &&\texttt{:- agg}(f, i, \texttt{map}(\lambda c'.\ \texttt{agg}(f, i, c'), c)) \\ &\texttt{groupagg}(f^a, g, i, \texttt{map}(f', c)) &&\texttt{:- groupagg}(f^a \circ f', g \circ f', i, c) \\ &\texttt{groupagg}(f^a, i, \texttt{flatten}(c)) &&\texttt{:-} \\ &\quad \texttt{groupagg}(f, \texttt{fst}, i, \texttt{flatten}(\texttt{map}(\lambda c'.\ \texttt{groupagg}(f, g, i, c'), c))) \end{aligned}$$

Above $\circ$ is function composition and `fst` the left projection of a pair. Additional deforestation rules for iterative usage

[4]This simple, illustrative example, can also be materialized as a range tree.

[5]Determining properties such as associativity, commutativity and idempotence via program analyses is undecidable [11]

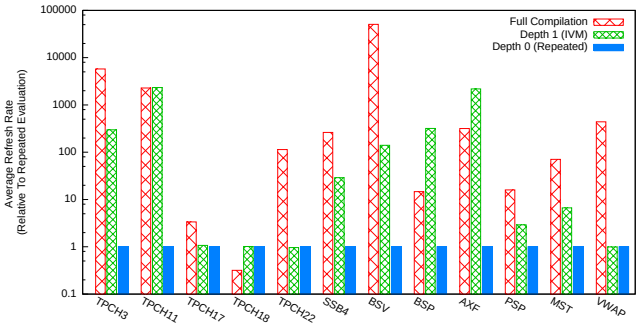


Figure 3: Performance improvement achievable by our compiler. Note the logscale on the y-axis.

of in-place updates are straightforward to derive.

The first rule for `map` eliminates the intermediate collection constructed after applying f', instead pipelining the results through the application of both functions f and f'. Since both f and f' can themselves contain `map` invocations, essentially performing joins, this rule can alter the arity of the join operation, constructing fully pipelined n-way joins. The second rule above is a form of late tuple construction [3], where by lifting the `flatten` expression, we carry around nested intermediate collection during processing. Repeating this facilitates deeply nested representations, in the same vein as non-first normal forms [31]. Aggregate deforestation also uses composition to realize pipelining and simplification, and the variants with `flatten` are partial aggregations that push an associative aggregation inside a nested collection.

Code generation Following functional optimization, we synthesize an imperative IR for easy code generation as the final step of compilation. The imperative IR supports external functions and types to enable the usage of library code. We implemented a C++ code generator based around the STL and its iterator-oriented collection APIs (e.g. `begin`, `end`, `find`, `erase`, `clear`, etc.). We omit further details due to space constraints. Our imperative optimizer and its datastructure and storage layer concerns is a work-in-progress.

4. EXPERIMENTAL RESULTS

We now analyze the performance of our compilation techniques and compare them with the results of Section 2. Experiments are run on Redhat Enterprise Linux with 16 GB of RAM, and 2x4 core Intel Xeon E5620 2.4 GHz processors allocated to it. A query workload overview is presented in Section 2, and as SQL in Appendix A. Queries were run over a replayed stream for 1 hour and then terminated. The results presented are computed based on the number of tuples processed within the 1 hour time limit. Memory measurements are taken using google-perftools, and count memory allocated to the materialized views and not transient data.

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 a schema of a timestamp, an order id, a broker id, a price, and a volume.

Queries from the TPC-H benchmark (including SSB4) were run on a scaling-factor 0.1 (100 MB) database generated by dbgen[34]. Additional scaling experiments were carried out on TPCH3 and TPCH11 at scaling factors 0.5, 1, 5, and 10, these results are presented in Section 4.3. Insertions are drawn directly from the table files generated by dbgen and interleaved in random order. Smaller datafiles finish earlier in the stream. This provides insight into the performance characteristics of operations on different tables.

To evaluate our compilation algorithm with a common performance baseline, we use a depth-limited instantiation of our compilation algorithm: Instead of recursively computing the entire materialization plan, the compiler stops after a fixed number of recursive steps. Beyond this stage, queries are not materialized and instead computed directly from the base relations. Compilation at depth-1 corresponds to traditional IVM techniques, and depth-0 corresponds to re-evaluating the query on every update. Figure 3 provides end-to-end results of this comparison on our workload.

Comparison with Existing Systems. Figure 4 compares multilevel IVM to the DBMS and stream processors presented in Section 2. Recall that CSPE and CDB were implemented using repeated evaluation (i.e., depth-0) to reflect development effort without recursive incrementalization. Comparing refresh rates, multilevel IVM spans the full gamut, from being three orders of magnitude faster to one order slower than CSPE and CDB. Looking at our C++ code next to their plans, there is clear room for improvement in our functional and imperative IR optimizations. Our engineering efforts thus far have been on developing a flexible compiler framework that can perform recursive compilation and materialization alongside optimization rather than reinvent the wheel with well-established cost-based optimizations. With future work on optimization designed specifically for our form of side-effecting incremental programs and a more mature compiler backend, we believe we can retain the demonstrated advantages of multilevel IVM while overcoming performance gaps for current adverse cases. From here on we discuss an apples-to-apples comparison of view maintenance in our framework alone.

4.1 Equijoins

We first analyze the performance of our compiler on three equijoin queries without nested aggregates. Both TPCH3 and SSB4 (Figures 5a, and 5c) demonstrate a substantial performance increase over IVM. The one-to-one and bounded fanout one-to-many relationships between elements of many of these queries are actually advantageous to the IVM implementation – each insertion only triggers a limited number of reads. In spite of this, incrementally maintaining the (aggregate) delta queries results in a net reduction in the amount of work required – especially in a large query like SSB4.

Also note the memory usage of TPCH3. Starting by the 40% marker, all streams have been exhausted except for LINEITEM. The final aggregate's group-by columns are drawn purely from ORDERS, so insertions into LINEITEM only update aggregate values for which entries have already been allocated by the corresponding ORDERS. Thus, memory usage plateaus for full compilation, while the IVM implementation must continue to store each row.

This is not always true. For extremely large queries like SSB4 (a 7-way join), the number of intermediate materialized views created is quite large. However, any individual update modifies only a small amount of that state – the fully compiled query is substantially faster as long as the system has enough memory. Memory usage is an important part of the cost/benefit tradeoff of full compilation, and we explore several other points in this space in Section 4.3.

Our compiler recurs only once on TPCH11 (Figure 5b), and the result is nearly equivalent to IVM. The fully compiled query materializes a projected and aggregated repre-

Query	Depth 0	CSPE	CDB	Full Compilation
TPCH3	4.75	59.69	38.34	27342.14
TPCH11	19.51	13.11	19.64	44597.8
TPCH17	19.76	32.58	8.48	66.27
TPCH18	6.13	21.13	38.07	1.95
TPCH22	2.4	39.82	45.83	273.37
SSB4	0.19	69.19	103.29	50.4
BSV	2.18	22.94	32.6	110261.68
BSP	0.63	13.79	31.47	9.24
AXF	0.17	12.28	31.1	54.15
PSP	0.67	10.52	13.32	10.69
MST	0.13	7.51	30.88	9.1
VWAP	7.4	9.31	12.89	3259.47

Figure 4: Comparison between the performance of our compiler and the two commercial query engines (in # of refreshes per second) from Section 2. These engines conceptually do the same amount of work as depth-0, but perform better as a result of commercial-grade optimization. We expect a similar improvement to be possible for full compilation.

sentation of SUPPLIER and PARTSUPP, but this provides only a minor improvement at this scale. Further scaling experiments can be seen in Figure 10. The performance of BSV (Figure 5d) is similar to TPCH3. Here, unlike TPCH11, the join relationship is many-to-many, and the benefits of maintaining the join result as an aggregate grow over time.

The takeaway with equijoin queries is that multilevel IVM is at least as good as, and mostly significantly dominates repetitive processing and IVM, and can greatly benefit fast refresh analytics for specific entities (e.g. a single broker or customer) which perform equality filtering and aggregation.

4.2 Nested subqueries

Figures 6a-d illustrate the performance of our compiler on several queries with nested aggregates. The lookup over ORDERS in TPCH22 can be evaluated in constant time both using IVM and full compilation. However each insertion into ORDERS incurs a nested aggregate on CUSTOMER, while full compilation materializes the aggregate result instead.

In IVM, insertions into CUSTOMER require two complete iterations over the customer table: once to compute the aggregate and once to figure out for which customers the state of the comparison changes. The latter iteration cannot be eliminated by full compilation. However, it need only iterate over the contents of the materialized view, which is already aggregated and projected down.

As with TPCH22, VWAP's uncorrelated aggregate can be evaluated efficiently if the query optimizer spots it – the inequality-correlated aggregate is of more interest. Because the domain of the correlating variable (price) is determined outside the nested aggregate, the nested subquery must be re-evaluated every time a new price is encountered. However, the aggregate value can then be stored and incrementally maintained from that point on. The domain of prices is bounded in practice, so after an initial ramp up process (that occurs while the size of the table is small) the fully compiled version can incrementally maintain the query output in (close to) constant time.

As in the last several queries, incrementally maintaining the nested aggregate of TPCH17 makes insertions into PART constant-time rather than linear. Even in the fully compiled version, insertions into the LINEITEM table must still iterate over the results of the join – under full compilation this join has already been materialized.

Depth	Avg Rate (refreshes/s)	Avg Memory per Tuple	Lines of Code	Number of Views
1	5.91	98.5 B	3174	6
2	0.373	167.0 B	12015	18
3	0.7	405.0 B	16517	36
4	12.7	24.7 KB	13215	45
5	51.5	62.2 KB	10998	45
Full	50.4	61.0 KB	10431	39

Figure 8: Statistics for different compilation depths on SSB4. Depth-5 is equivalent to full compilation, but also maintains each of the 6 base relations.

We touch on TPCH18 (Figure 6d) and Figures 7a-d in Section 4.4 to describe the implied limitations of our current approach. The takeaway for these queries is that multilevel IVM does offer improvements (PSP, MST) over repetitive and IVM, but the situation is less clear cut (e.g. AXF, BSP). In particular, we observe that views with large domains for their input variables can be costly, with entries being added and rarely reused, while still incurring maintenance work. This motivates advanced partial materialization and garbage collection strategies alongside multilevel IVM.

4.3 Other metrics

Limited Recursion. We now explore the space of limited recursive compilation beyond IVM. Figure 8 illustrates the effects of limiting compilation to depths between 0 and 5. Recall that the maximum recursive depth is one less than the join width of the query. Thus for SSB4 (which has a join width of 6), compilation to depth-5 is equivalent to full compilation, save that the base relations are materialized.

At depth-1, the compiled query materializes only the base relations and no intermediate tables. It must still perform a 5-way join on every insertion, but only once per update. The 6 materialized views that it maintains are the 6 base relations from the query.

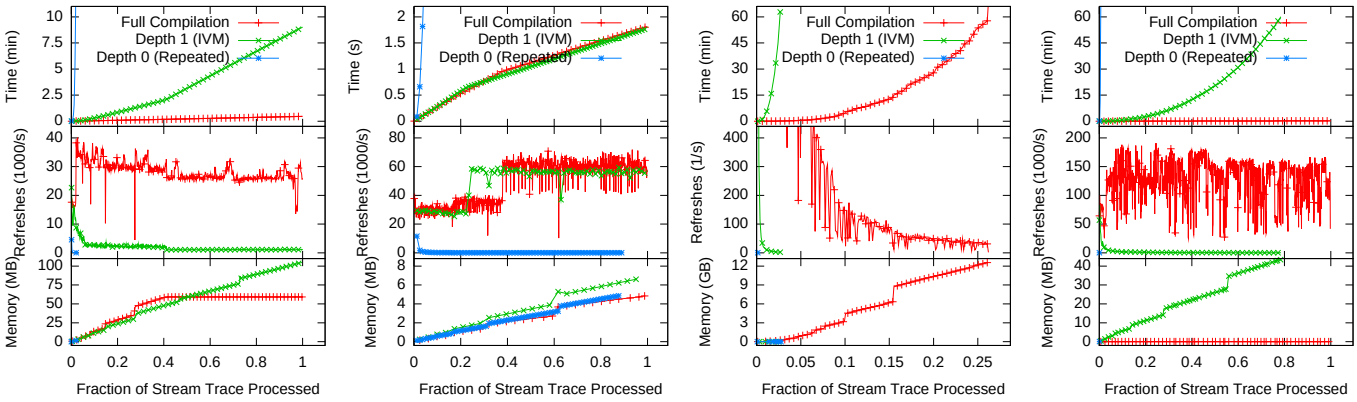
At depth-2, the compiled query must now maintain 12 intermediate materialized views, several of which require a 4-way join to maintain. The net cost of maintaining these additional views does not begin to pay off until depth-4 (where maintenance operations are reduced to at most 2-way joins). By this point, decomposition has already resulted in the instantiation of all intermediate materializations relevant to the query, so extra and unnecessary work is being done.

The effectiveness of this approach at depth-4 (in spite of the extra work) suggests that a more effective approach to reducing memory consumption might be to materialize not just the set of views closest to the root, but rather a subset of the possible materialized views. However, the space of materialization strategies is exponential, and a cost based optimizer is future work.

Scaling. Figure 10 analyzes how two queries: TPCH3 and TPCH11 scale to larger datasets. Note that on TPCH3, the depth-1 implementation does not successfully complete the entire workload within the 1-hour time limit on any scaling factor greater than 0.1. With full compilation, Query 3 consumes a large fraction of system memory on the 10 GB dataset, but performance remains constant throughout. Query 11's performance also remains constant – note the nearly 50% improvement over IVM at higher scales, an effect of pushing aggregation into the materialized views.

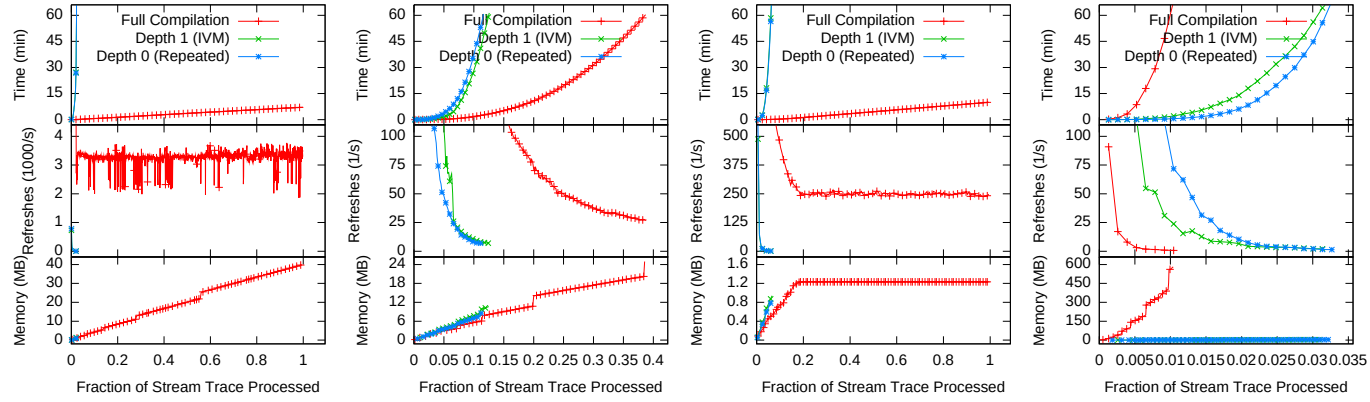
4.4 Memory, extraction, and future work

The above experiments reveal several cases where the per-



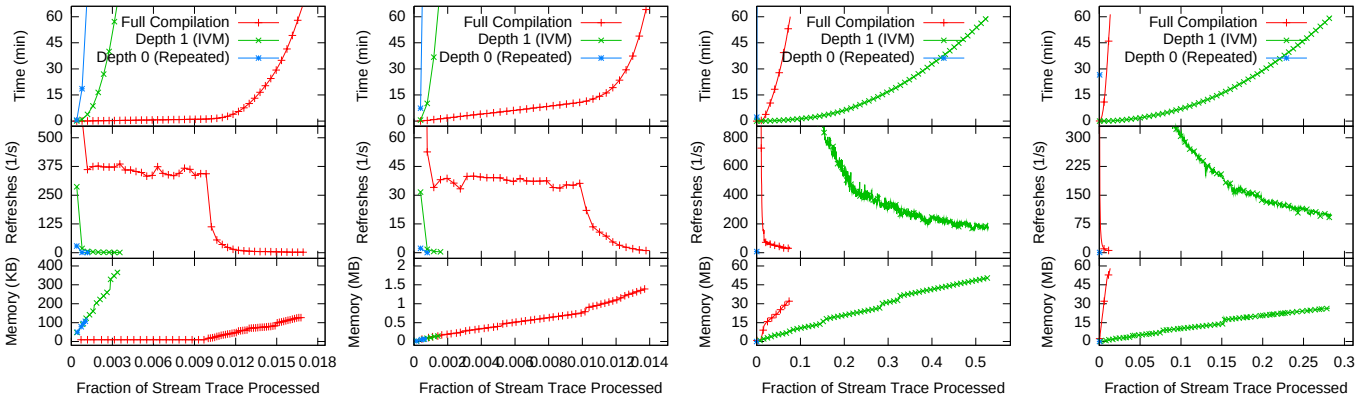
(a) (b) (c) (d)

Figure 5: TPCH3 (a), TPCH11 (b), SSB4 (c), and BSV (d): (a) By the 40% marker, all streams except LINEITEM have completed, and the remaining tuples consume no additional memory. (b) For simple two-way joins, full compilation is virtually identical to depth-1 and takes under 2 seconds, while depth-0 takes over an hour. (c) Full compilation is an order of magnitude faster than in IVC, although performance drops once the system begins running out of memory around the 27% marker. (d) The many-to-many relationship on the join term forces IVM to perform linear work on each insertion, which full compilation avoids.



(a) (b) (c) (d)

Figure 6: VWAP (a), TPCH17 (b), TPCH22 (c) and TPCH18 (d): (a) IVM repeatedly re-evaluates the nested (parameterized) sub-query, while full compilation maintains a cache of sub-query results. (b) Due to the nested aggregate, IVC requires a nested loop, while full compilation requires only a single scan. (c) The small CUSTOMER stream completes at the 10% marker, while the remaining ORDERS tuples require only linear time with full compilation. (d) A badly chosen join ordering prevents full compilation from effectively exploiting foreign key dependencies in the TPC-H schema.



(a) (b) (c) (d)

Figure 7: PS (a), MST (b), AXF (c) and BSP (d): (a,b) The performance and memory plateaus result from a portion of the trace from about 0.001% to 0.01%, where a single order is repeatedly placed and revoked. (c,d) Full compilation's aggressive materialization strategy results in the caches growing too large to be efficiently maintained.

	Full compilation	Depth 1	Depth 0
TPCH3	2509	2855	4198
TPCH11	531	596	616
TPCH17	928	1158	1478
TPCH18	3668	3538	4631
TPCH22	777	1135	754
SSB4	10995	8954	7904
BSV	342	327	347
BSP	45625	567	729
AXF	2169	553	1394
PSP	1442	1878	1890
MST	5457	2870	2434
VWAP	533	466	341

Figure 9: Lines of C++ generated for each query.

formance of multilevel IVM leaves room for improvement. We identify three challenges which are non-trivial and open avenues for substantial future work.

Join Ordering. Despite the simplicity of TPCH18 (Figure 6d), the query performs poorly – repeated evaluation is faster. This occurs due to join ordering: to update query results, we must join the delta of the extracted nested subquery (aggregated over orderkey) and a materialized representation of CUSTOMER ⋈ ORDER ⋈ LINEITEM. We iterate over the materialized 3-way join first.

We mentioned the focus of our engineering efforts when comparing to CSPE and CDB, and plan to implement full cost-based materialization and query planning, and improve the functional and imperative IR. These efforts are outside the scope of this paper, yet recent works on holistic optimization [27] and compilation [29] have shown there is room for improvement over existing engines. Backend compiler design will require a deep understanding of the compilers literature, and must be married to the higher-level recursive delta compilation we perform.

Domain Maintenance. Both PS and MST (Figures 7a, and 7b respectively) do not perform as well as possible – Apart from a stretch of updates (0.001% to 0.01% in the trace) in the stock market trace where the same order is repeatedly placed and revoked, query performance decreases polynomially over time. This slowdown is due to overly aggressive caching. We do not connect the domain of a materialized view's output variables with the domains of its base relations. The effect of removing a row in the base relation is propagated to the multiplicity of the corresponding row(s) of the view. However, the row itself is not garbage collected, and remains a part of iterations over the view.

Garbage collection and finalization of map entries (in addition to initialization) will require rigorous analysis of delta queries to understand their effect on a variable's domain. Such issues are not typically the concern of DBMS engines, where query processing temporaries are discarded after an operator's set-at-a-time computation completes or at transaction boundaries. On the other hand, generational garbage collectors as seen in modern VMs such as the JVM and Microsoft's CLR apply to general purpose programs rather than the incremental programs we have, where one can exploit data properties and delta queries that touch a small fraction of the entire database state.

Map Extraction. The final case of performance issues is seen in both AXF (Figure 7c) and BSP (Figure 7d). Our aggressive extraction heuristic attempts to materialize the entire delta query, which for inequality joins includes an unbound variable. The domain of this unbound variable is

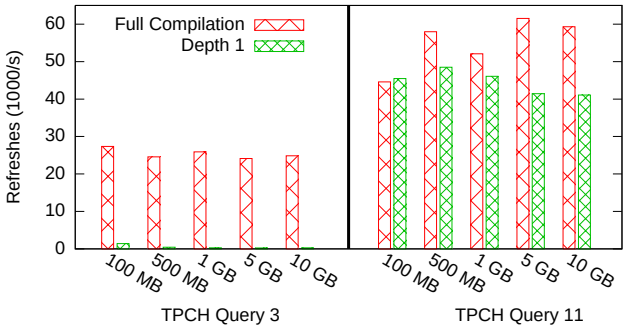


Figure 10: Performance scaling with respect to dataset size for TPCH 3 and 11 respectively. For these queries, performance remains roughly constant as long as sufficient memory is available.

as large as either input table – insertions into either table trigger maintenance work that is linear in the number of prior insertions onto the other table. Worse still, the work saved by doing this is minimal – the aggregate value must be computed from scratch on every insertion.

An improved, data-dependent extraction heuristic can identify such situations and compute the inequality join inline. This is precisely what IVM is already doing – hence the performance improvement. Alternatively, the entire materialized delta could be incrementally maintained more efficiently using datastructures suited to computing aggregates over ranges (e.g., Range Trees[21]).

5. CONCLUSION

We presented multilevel IVM, motivated the need to compile its maintenance work based on aggressive simplification of recursive delta queries, and presented the design of a compiler framework that incorporates a plethora of optimization techniques to make recursive IVM viable.

Our compilation technique is effective on select-project-join-aggregate queries involving equi-joins and nested subqueries which are uncorrelated, correlated through an equality comparison, or correlated on a variable (or variables) with a small domain. It is especially good on queries with small result sets (but large inputs). Our technique is less effective on inequality joins and nested aggregates correlated through an inequality – although both are still handled efficiently if the domain of the values being compared is small. In a similar vein, we do not optimize to take advantage of, or avoid problems caused by data-dependent characteristics (e.g., foreign keys, large domains).

We have presented guidelines for addressing these issues in future work by applying both heuristics and cost-based optimizers to the space of possible materialization plans. With these optimizations in place, our compilation technique has the potential to dominate existing systems in this space, and to enable a broad new class of monitoring applications.

6. REFERENCES

[1] Microsoft SQL Server 2008 indexed view documentation, view restrictions.
[2] Oracle Database 11g Release 1 Documentation.
[3] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. Madden. Materialization strategies in a column-oriented DBMS. In *ICDE*, 2007.
[4] Y. Ahmad and C. Koch. DBToaster: A SQL compiler for high-performance delta processing in main-memory

databases. *PVLDB*, 2(2):1566–1569, 2009.

[5] S. Aji, R. McEliece. The generalized distributive law. *IEEE Trans. Inf. Theory*, 46(2):325–343, 2000.

[6] E. Baralis and J. Widom. Using delta relations to optimize condition evaluation in active databases. In *Rules in Database Systems*, 1995.

[7] Basel Committee on Banking Supervision. International convergence of capital measurements and capital standards: A revised framework. June 2006.

[8] J. Blakeley, P. Larson, F. Tompa. Efficiently updating materialized views. In *SIGMOD*, 1986.

[9] I. Botan, R. Derakhshan, N. Dindar, L. M. Haas, R. J. Miller, and N. Tatbul. Secret: A model for analysis of the execution semantics of stream processing systems. *PVLDB*, 3(1):232–243, 2010.

[10] P. Buneman and E. K. Clemons. Efficiently monitoring relational databases. *ACM TODS*, 4(3):368–382, 1979.

[11] P. Buneman, S. A. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theor. Comp. Sci.*, 149(1):3–48, 1995.

[12] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *VLDB*, 1991.

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

[14] B. Gedik, H. Andrade, K.-L. Wu, P. Yu, and M. Doo. SPADE: the System S declarative stream processing engine. In *Proc. SIGMOD*, 2008.

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

[16] T. M. Ghanem, A. K. Elmagarmid, P.-Å. Larson, and W. G. Aref. Supporting views in data stream management systems. *ACM TODS*, 35(1), 2010.

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

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

[19] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, 1993.

[20] A. Hey, S. Tansley, and K. Tolle. *The fourth paradigm: data-intensive scientific discovery*. Microsoft Research Redmond, WA, 2009.

[21] C.-T. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in olap data cubes. In *SIGMOD*, 1997.

[22] IBM. IBM DB2 Database Version 9.5 Documentation.

[23] N. Jain, S. Mishra, J. Gehrke, J. Widom, H. Balakrishnan, U. Çetintemel, M. Cherniack, R. Tibbetts, S. Zdonik. Towards a streaming SQL standard. *PVLDB*, 1(2):1379–1390, 2008.

[24] O. Kennedy, Y. Ahmad, and C. Koch. DBToaster: Agile views for a dynamic data management system. In *Proc. CIDR*, 2011.

[25] C. Koch. Incremental query evaluation in a ring of databases. In *PODS*, 2010.

[26] Y. Kotidis and N. Roussopoulos. A case for dynamic view management. *TODS*, 26(4):388–423, 2001.

[27] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE*, 2010.

[28] S. Marlow and P. Wadler. Deforestation for higher-order functions. In *Func. Programming*, 1992.

[29] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[30] N. Roussopoulos. An incremental access method for ViewCache: Concept, algorithms, and cost analysis. *TODS*, 16(3):535–563, 1991.

[31] H.-J. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Inf. Syst.*, 11(2):137–147, 1986.

[32] C. Scholz, L. Sykes, and Y. Aggarwal. Earthquake prediction: a physical basis. *Science*, 181(4102), 1973.

[33] O. Shmueli and A. Itai. Maintenance of views. In *Proc. SIGMOD*, 1984.

[34] Transaction Processing Performance Council. TPC-H benchmark spec: `http://www.tpc.org/hspec.html`.

[35] W. P. Yan and P.-Å. Larson. Eager aggregation and lazy aggregation. In *VLDB*, 1995.

APPENDIX

A. QUERIES

AXF

```
SELECT   b.broker_id, sum(a.volume-b.volume) FROM bids b, asks a
WHERE    b.broker_id = a.broker_id
  AND    ( (a.price - b.price > 1000) OR (b.price - a.price > 1000) )
GROUP BY b.broker_id;
```

BSP

```
SELECT x.broker_id, SUM(x.volume * x.price - y.volume * y.price)
FROM   bids x, bids y
WHERE  x.broker_id = y.broker_id AND x.t > y.t GROUP BY x.broker_id;
```

BSV

```
SELECT x.broker_id, SUM(x.volume * x.price * y.volume * y.price * 0.5)
FROM   bids x, bids y
WHERE  x.broker_id = y.broker_id GROUP BY x.broker_id;
```

MST

```
SELECT b.broker_id, sum(a.price*a.volume - b.price*b.volume)
FROM   bids b, asks a
WHERE 0.25*(select sum(a1.volume) from asks a1) >
      (select sum(a2.volume) from asks a2 where a2.price > a.price)
AND   0.25*(select sum(b1.volume) from bids b1) >
      (select sum(b2.volume) from bids b2 where b2.price > b.price)
GROUP BY b.broker_id;
```

PSP

```
SELECT sum(a.price - b.price) FROM bids b, asks a
WHERE ( b.volume>0.0001*(select sum(b1.volume) from bids b1) )
AND   ( a.volume>0.0001*(select sum(a1.volume) from asks a1) );
```

VWAP

```
SELECT sum(b1.price * b1.volume) FROM bids b1
WHERE  0.25 * (select sum(b3.volume) from bids b3) >
       (select sum(b2.volume) from bids b2 where b2.price > b1.price);
```

Q3

```
SELECT ORDERS.orderkey, ORDERS.orderdate, ORDERS.shippriority,
       SUM(extendedprice * (1 - discount))
FROM   CUSTOMER, ORDERS, LINEITEM
WHERE  CUSTOMER.mktsegment = 'BUILDING'
  AND  ORDERS.custkey = CUSTOMER.custkey
  AND  LINEITEM.orderkey = ORDERS.orderkey
  AND  ORDERS.orderdate < DATE('1995-03-15')
  AND  LINEITEM.SHIPDATE > DATE('1995-03-15')
GROUP BY ORDERS.orderkey, ORDERS.orderdate, ORDERS.shippriority;
```

Q11

```
SELECT ps.partkey, sum(ps.supplycost * ps.availqty)
FROM   partsupp ps, supplier s
WHERE  ps.suppkey = s.suppkey GROUP BY ps.partkey;
```

Q17

```
SELECT sum(l.extendedprice) FROM lineitem l, part p
WHERE  p.partkey = l.partkey
AND    l.quantity < 0.005 * (SELECT sum(l2.quantity)
          FROM lineitem l2 WHERE l2.partkey = p.partkey);
```

Q18

```
SELECT c.custkey, sum(l1.quantity)
FROM customer c, orders o, lineitem l1
WHERE 1 <= (SELECT sum(1) FROM lineitem l2
            WHERE l1.orderkey = l2.orderkey
            AND 100 < (SELECT sum(l3.quantity) FROM lineitem l3
                       WHERE l2.orderkey = l3.orderkey))
AND c.custkey = o.custkey AND o.orderkey = l1.orderkey
GROUP BY c.custkey;
```

Q22

```
SELECT c1.nationkey, sum(c1.acctbal) FROM customer c1
WHERE c1.acctbal <
    (SELECT sum(c2.acctbal) FROM customer c2 WHERE c2.acctbal > 0)
AND 0 = (SELECT sum(1) FROM orders o WHERE o.custkey = c1.custkey)
GROUP BY c1.nationkey
```

SSB4

```
SELECT sn.regionkey, cn.regionkey, PART.type, SUM(LINEITEM.quantity)
FROM   CUSTOMER, ORDERS, LINEITEM, PART, SUPPLIER, NATION cn, NATION sn
WHERE  CUSTOMER.custkey = ORDERS.custkey
  AND  ORDERS.orderkey = LINEITEM.orderkey
  AND  PART.partkey = LINEITEM.partkey
  AND  SUPPLIER.suppkey = LINEITEM.suppkey
  AND  ORDERS.orderdate >= DATE('1997-01-01')
  AND  ORDERS.orderdate <  DATE('1998-01-01')
  AND  cn.nationkey = CUSTOMER.nationkey
  AND  sn.nationkey = SUPPLIER.nationkey
GROUP BY sn.regionkey, cn.regionkey, PART.type
```