How to Win a Hot Dog Eating Contest: Distributed Incremental View Maintenance with Batch Updates[*] [†]

Milos Nikolic Mohammad Dashti Christoph Koch
{milos.nikolic, mohammad.dashti, christoph.koch}@epfl.ch
École Polytechnique Fédérale de Lausanne

ABSTRACT

In the quest for valuable information, modern big data applications continuously monitor streams of data. These applications demand low latency stream processing even when faced with high volume and velocity of incoming changes and the user's desire to ask complex queries. In this paper, we study low-latency incremental computation of complex SQL queries in both local and distributed streaming environments. We develop a technique for the efficient incrementalization of queries with nested aggregates for batch updates. We identify the cases in which batch processing can boost the performance of incremental view maintenance but also demonstrate that tuple-at-a-time processing often can achieve better performance in local mode. Batch updates are essential for enabling distributed incremental view maintenance and amortizing the cost of network communication and synchronization. We show how to derive incremental programs optimized for running on large-scale processing platforms. Our implementation of distributed incremental view maintenance can process tens of million of tuples with few-second latency using hundreds of nodes.

1. INTRODUCTION

Many big data applications demand real-time analytics over high-velocity streaming data. In a growing number of domains – sensor network monitoring, clickstream analysis, high-frequency algorithmic trading, and fraud detection to name a few – applications continuously monitor rapidly-changing datasets in order to promptly detect certain patterns, anomalies, or future trends. Application users construct complex queries to reveal more information and expect to have fresh results available at all time. Such streaming applications usually have strict latency requirements, which frequently forces developers to build ad-hoc solutions in order to pull off utmost performance.

We identify three essential requirements for stream processing systems to serve these modern applications:

- *Support for complex continuous queries.* Modern streaming engines need to support continuous queries that can capture complex conditions, such as SQL queries with nested aggregates.
- *Low-latency (incremental) processing.* Online and responsive analytics enable data analysts to quickly react to changes in the underlying data. Streaming datasets evolve through frequent, small-sized updates, which makes refreshing query results using recomputation too expensive. Online systems rely on *incremental computation* to sustain high update rates and achieve low refresh latency.
- *Scalable processing.* Emerging big data applications demand scalable systems for querying and managing large datasets. Scalable processing aims to speed up query evaluation and accommodate growing memory requirements.

Modern data processing systems often fail to simultaneously address the above requirements. Traditional relational database systems support complex SQL queries but suffer from high query latencies despite their support for incremental computation of materialized views [18, 17, 14]. Conventional data stream SQL processing systems [26, 4] use incremental algorithms for evaluating continuous queries over finite windows of input data. Their usefulness is yet limited by their window semantics, inability to handle long-lived data, and lack of support for complex queries. Both classical databases and stream processing engines have limited scalability. Scalable stream processing platforms[1], like MillWheel [7], S4 [29], and Heron [25], offer low-level programming models that put the burden of expressing complex query plans on the application developer. Naiad [27] and Trill [11] focus mostly on flat LINQ-style continuous queries; encoding and efficiently incrementalizing complex queries with nested aggregates is a non-trivial task left to the user. Spark Streaming [39] supports running simple SQL queries but only over windowed data.

In this work, we study low-latency incremental processing of complex SQL queries in both local and distributed environments. Our approach derives, at compile time, delta programs that capture changes in the result for updates to the database. For efficient incremental computation, we build upon our previous work on recursive incremental view maintenance [20, 21], whose implementation inside the DBToaster system demonstrates $3-4$ orders of magnitude better performance than commercial database and stream processing engines in local settings [22].

DBToaster's recursive view maintenance can deliver μ-second latencies for single-tuple updates on standard database workloads.

[*]It is all about the bite size, and the best bite size is $1,000$ to $10,000$ tuples in local execution. We also address the distributed case.

[†]This work was supported by ERC Grant 279804.

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-3531-7/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2882903.2915246

[1]We will, throughout this paper, use stream processing and continuous queries interchangeably: We will not aim to ensure bounded state size by e.g. window semantics. This is vindicated by the common use of the term streaming in this relaxed way in recent times.

In this work, we generalize this idea to batches of updates. We present a novel technique called *domain extraction* that enables efficient incremental maintenance of complex SQL queries with nested aggregates for batch updates. We study the trade-offs between single-tuple and batched incremental processing and identify the cases where batching can improve the performance of incremental view maintenance. Our experiments show that maintenance programs specialized for single-tuple processing can outperform generic batched implementations in many cases. These results refute the widespread belief that batching always wins over tuple-at-a-time processing [30].

Batched processing enables distributed incremental view maintenance and amortizes increased costs of communication and coordination in large-scale deployments. In this paper, we show how to transform local view maintenance code into programs capable of running on large-scale processing platforms. Our implementation of distributed incremental view maintenance inside DBToaster runs on top of Spark [38] and can process tens of million of updates with few-second latency on a scale of hundreds of workers.

Our system compiles SQL workloads into aggressively specialized query engines. We rely on a modern compiler framework, called LMS [35], to generate low-level native code that is free of overheads of classical query processing engines (e.g., template-based operators). We create custom data structures for storing materialized views that are optimized for the observed access patterns. We also specialize query engines for different batch sizes, for instance, to avoid materialization when processing single-tuple updates or to use columnar mode when serializing large batches.

In this work, we make the following contributions:

1. We present techniques for the efficient recursively incremental processing of queries with nested aggregates for batch updates. This part of the paper generalizes our previous work [20, 21, 22].

2. We study the trade-offs between single-tuple and batched incremental processing in local settings. We analyze the cases when batching can greatly reduce maintenance costs.

3. We describe a novel approach for compiling incremental view maintenance code into programs optimized for running in distributed environments.

4. We present techniques for code and data-structure specialization of incremental view maintenance programs.

5. Our single-node experiments show that single-tuple processing can often outperform batched processing. Our distributed view maintenance implementation can deliver few-second latencies of processing tens of million of tuples using hundreds workers.

This paper is organized as follows. We provide an overview of incremental view maintenance in Section 2. We discuss delta processing for batch updates in Section 3 and distributed incremental view maintenance in Section 4. We present our code and data-structure specialization techniques in Section 5, experimental evaluation in Section 6, and related work in Section 7.

2. VIEW MAINTENANCE

Database systems use materialized views to speed up execution of frequently asked queries. Materialized views require maintenance to keep their contents up to date for changes in the base tables. Refreshing materialized views using recomputation is expensive for frequent, small-sized updates. In such cases, applying only incremental changes (deltas) to materialized views is usually more efficient than recomputing views from large base tables.

2.1 Classical Incremental View Maintenance

Let $(Q, M(\mathbf{D}))$ denotes a materialized view, where Q is the view definition query and $M(\mathbf{D})$ is the materialized contents for a given database $\mathbf{D}$. When the database changes from $\mathbf{D}$ to $(\mathbf{D} + \Delta\mathbf{D})$, classical incremental view maintenance *evaluates* a delta query ΔQ in order to refresh $M(\mathbf{D})$.

$$M(\mathbf{D} + \Delta\mathbf{D}) = M(\mathbf{D}) + \Delta Q(\mathbf{D}, \Delta\mathbf{D})$$

The delta ΔQ often has a simpler structure than Q (e.g., has fewer joins) and involves smaller delta updates instead of large base tables. So, computing ΔQ and refreshing $M(\mathbf{D})$ becomes cheaper than re-evaluating Q from scratch. Incremental view maintenance derives one delta query for each referenced base table. The derivation process relies on a set of change propagation rules defined for each operator of the view definition language. The derived delta query takes its role in the associated view maintenance trigger.

EXAMPLE 2.1. *Let query Q counts the tuples of a natural join of $R(A,B)$, $S(B,C)$, and $T(C,D)$ grouped by column B. Intuitively, we write the delta query for updates to R as:*

$$\Delta_R Q := Sum_{[\mathrm{B}]}(\Delta R(A,B) \bowtie S(B,C) \bowtie T(C,D))$$

Let M_R, M_S, and M_T denote the materialized base tables, then the maintenance trigger for updates to R looks as:

```
ON UPDATE R BY dR
  M_R(A,B) += dR(A,B)
  dQ(B)    := Sum_[B](dR(A,B) * M_S(B,C) * M_T(C,D))
  M_Q(B)   += dQ(B)
```

Here, `+=`, `:=`, *and $*$ denote bag union, assignment, and natural join. Under the standard assumption that $|\Delta R| \ll |R|$, incremental maintenance is cheaper than re-evaluation.* □

Incremental view maintenance is often cheaper than naïve re-evaluation but is not free. Computing deltas can be expensive, like in Example 2.1, where $\Delta_R Q$ is a non-trivial join of one (small) input update and two (potentially large) base tables.

2.2 Recursive Incremental View Maintenance

Delta queries can be expensive despite their simpler form. For instance, a delta of an n-way join still references $(n-1)$ base tables. Instead of computing such a delta query from scratch, we could re-apply the idea of incremental processing to speed up the delta evaluation: store previously computed delta results, just as any other query result, and compute the delta of a delta query (second-order delta) to maintain the materialized delta result. That way, the second-order delta query maintains the first-order delta view, which in turn maintains the top-level view. Assuming that with each derivation deltas become simpler, we could recursively apply the same procedure until we get deltas with no references to base tables. The described procedure is known as *recursive incremental view maintenance* [20, 21, 22].

Recursive incremental view maintenance materializes the top-level view along with a set of auxiliary views that support each other's incremental maintenance. The materialization procedure starts from the top-level view and derives its delta queries for updates to base relations. For each delta query, the procedure materializes its update-independent parts such that the delta evaluation requires as little work as possible. Formally, the procedure transforms $\Delta Q(\mathbf{D}, \Delta\mathbf{D})$ into an equivalent query $\Delta Q'$ that evaluates over a set of materialized views $M_1, \ldots, M_k$ and update $\Delta\mathbf{D}$:

$$\Delta Q(\mathbf{D}, \Delta\mathbf{D}) = \Delta Q'(M_1(\mathbf{D}), M_2(\mathbf{D}), ..., M_k(\mathbf{D}), \Delta\mathbf{D})$$

But note that $M_1, \ldots, M_k$ also require maintenance, which again relies on simpler materialized views. At first, it may appear counterintuitive that storing more data can reduce maintenance costs. However, the recursive incremental maintenance scheme makes the work required to keep all views fresh extremely simple. For flat queries, each individual aggregate value can be incrementally maintained using a constant amount of work [21], which is impossible to achieve with classical incremental maintenance or recomputation.

EXAMPLE 2.2. *Let us apply recursive incremental view maintenance on the query of Example 2.1 and updates to R. Considering $\Delta_R Q$, we materialize its update-independent part $S(B,C) \bowtie T(C,D)$ as an auxiliary view $M_{ST}(B)$. We projected away C and D as they are irrelevant for the computation of $\Delta_R Q$. Repeating the same procedure for updates to T, we materialize $R(A,B) \bowtie S(B,C)$ as $M_{RS}(B,C)$ to facilitate computing of $\Delta_T Q$. For updates to S, we materialize $R(A,B) \bowtie T(C,D)$ separately as $M_R(B)$ and $M_T(C)$*[2]*.*

Next, we derive second-order deltas for M_{ST} and M_{RS}. Repeating the same delta derivation for updates to all three base relations, we materialize one additional view $M_S(B,C)$ representing the base relation S. Further derivation produces delta expressions with no base relations. Overall, recursive view maintenance materializes queries at three different levels: the top-level query M_Q, two auxiliary views M_{RS} and M_{ST}, and the base tables M_R, M_S, and M_T. The maintenance trigger for updates to R looks as:

```
ON UPDATE R BY dR
  M_Q(B)     += Sum_[B](dR(A,B) * M_ST(B))
  M_RS(B,C) += Sum_[B](dR(A,B) * M_S(B,C))
  M_R(B)     += Sum_[B](dR(A,B))
```

We similarly build triggers for updates to S and T. □

Recursive incremental view maintenance can produce triggers with lower complexity than classical maintenance triggers. In the previous example, each statement performs at most one join between the delta relation and one materialized view, which is clearly less expensive than the classical approach. In practice, DBToaster's recursive view maintenance can outperform classical view maintenance by several orders of magnitude for single-tuple updates [22].

2.3 Contributions

We summarize our contributions on our running example.

Domain extraction. Consider a modified query from Example 2.1 that reports only distinct B tuples. The delta of this query for updates to R is, unfortunately, more complex than the query itself as it reevaluates the whole query from scratch just to compute the incremental change. Clearly, that defeats the purpose of incremental computation.

In Section 3, we show how to incrementally maintain such delta queries. We observe that ΔR affects only the output tuples whose B value appears in ΔR. We present an algorithm that restricts the domain of delta evaluation to only those output tuples affected by the given input change. We also discuss the cases in which reevaluation is preferable over incremental computation.

Single-tuple vs. batch processing. We study the trade-offs between tuple-at-a-time and batched recursive incremental view maintenance in local settings. The former yields simpler maintenance code; in the trigger of Example 2.2, we can eliminate the loop around ΔR for single-tuple updates and achieve constant-time maintenance of M_Q and M_R. The latter can have positive or negative impacts on cache locality and can significantly reduce view maintenance cost; in the same example, we can pre-aggregate ΔR as $\text{Sum}_{[\text{B}]}(\Delta R(A,B))$ to keep only distinct B values along with their multiplicity. A small aggregated domain makes the maintenance code cheaper: our experiments show that batch pre-aggregation can bring up to 3 orders of magnitude performance improvements.

Distributed execution. In this work, we also study distributed execution of incremental programs. Distributed incremental view maintenance opens new problems compared to classical parallel and distributed query processing [24, 34]. First, we optimize multiple delta queries at once, where each query handles changes to one base relation. Second, during optimization we also need to consider the partitioning information of the target view being refreshed. Finally, recursive incremental view maintenance creates data-flow dependencies among statements maintaining auxiliary materialized views. These dependencies prevent arbitrary re-orderings of statements inside a trigger function. For instance, evaluating an n-th order delta relies on simpler, $(n+1)$-th order materialized views, whose contents is maintained (if necessary) in subsequent statements; in Example 2.2, we first maintain the query result, then the auxiliary view, and finally the base relation. That is, we maintain materialized views in decreasing order of their complexity. This property creates a DAG of dependencies among trigger statements, bringing more complexity to the problem.

Batch processing is essential for efficient implementation of distributed incremental view maintenance as it amortizes increased costs of network communication and synchronization. In Section 4, we show how to transform view maintenance code, like that from Example 2.2, into programs optimized for running in distributed settings. To achieve that goal, we redefine the query language with location-aware constructs, introduce new operators for exchanging data over the network, and develop a set of optimization rules that are specific to distributed environments.

Program specialization. Commercial database systems rely on embedded query evaluation mechanisms to compute deltas. In practice, they exhibit poor view maintenance performance due to overheads from components not directly related with view maintenance (e.g, logging, buffer management, concurrency control, etc.) and the use of general data structures and algorithms [22, 19, 28].

We argue that efficient view maintenance requires specialization of incremental programs. We observe that nowadays most applications have static query workloads with template-derived queries. Knowing the workload in advance allows us to tailor query processing based on the application requirements and avoid unnecessary features of database systems. Thus, we specialize incremental programs into low-level native or interpreted code and generate custom data structures to facilitate efficient maintenance operations.

EXAMPLE 2.3. *Let us show the idea of program specialization on the trigger from Example 2.2. At compile time, we analyze the access patterns inside triggers and create custom data structures with index support for storing the view contents. For instance, we optimize ΔR for scanning, while the other views may feature efficient lookup, update, and/or slice operations. Then, we transform trigger statements into function calls over these objects. For example, the R trigger may have the following functional representation:*

```
def onUpdateR(dR) {
  dR.foreach((k1,v1) =>       // key-value pairs
    mQ.update(k1.B, v1*mST.get(k1.B))
    mS.slice(k1.B, (k2,v2) =>
      mRS.update((k1.B,k2.C), v1*v2))
    mR.update(k1.B, v1)
  )
}
```

We cover the compilation process in more detail in Section 5. The take-away here is that, in the quest for maximum performance,

[2]Recursive incremental view maintenance avoids storing query results with disconnected join graphs for performance reasons [22].

we aspire to specialize high-level operators into optimized low-level implementations that enable efficient view maintenance at the speed of native code.

3. BATCHED DELTA PROCESSING

In this section, we focus on the problem of view maintenance for batch updates. First, we revisit the data model and query language from our previous work [22]; Appendix A provides more details.

3.1 Data Model and Query Language

Data Model. We materialize views as generalized multiset relations [22, 20, 21]. One relation contains a finite number of unique tuples with a non-zero, positive or negative, multiplicity. The data model generalizes tuple multiplicities from single count integers to multiple (potentially non-integer) values that correspond to different aggregate values (e.g., `SUM`, `AVG`). Traditional SQL represents these aggregates as separate columns in the query result, while this model keeps them in the multiplicities to facilitate incremental processing – updating aggregate values means changing tuple multiplicities rather than deleting and inserting tuples from the result. For simplicity, in this paper we associate each tuple with one count or aggregate multiplicity.

Query Language. The query language uses algebraic formulas to define queries (views) over generalized multiset relations. The language consists of relations $R(\vec{A})$, bag union $Q_1 + Q_2$, natural join $Q_1 \bowtie Q_2$, multiplicity-preserving projection $\mathrm{Sum}_{\vec{A}}(Q)$, constants, values $f(var1, var2, \ldots)$, variable assignments $(var := value)$, and comparisons. To support queries with nested aggregates, the query language generalizes the assignment operator to take on arbitrary expressions instead of just values: $(var := Q)$ defines a finite-size relation containing tuples of expression Q with non-zero multiplicities extended by column *var* holding that multiplicity. Each output tuple has the multiplicity of 1. Expression Q may be correlated with the outside as usual in SQL.

EXAMPLE 3.1. *Consider a query over $R(A,B)$ and $S(B,C)$:*

```
SELECT COUNT(*) FROM R WHERE R.A <
  (SELECT COUNT(*) FROM S WHERE R.B = S.B)
```

The nested query is $Q_n := Sum_{[]}(S(B_2,C) \bowtie (B = B_2))$, where B comes from the outer query. The whole query is $Sum_{[]}(R(A,B) \bowtie (X := Q_n) \bowtie (A < X)))$. □

Variable assignment $(var := Q)$ can also express existential quantification. Assume the query of Example 3.1 had an `EXISTS` condition instead of the comparison. Then, we would write the condition part as $(X := Q_n) \bowtie (X \neq 0)$.

Delta Queries. Delta queries capture changes in query results for updates to base relations. For any query expression Q, we can construct a delta query ΔQ using a set of derivation rules defined for each operator of the query language. Our previous work studied these rules in detail [21, 22]; in short, the delta rules for updates ΔR to R are:

$$\begin{aligned}
\Delta_R(R(A_1,A_2,\ldots)) &:= \Delta R(A_1,A_2,\ldots)\\
\Delta_R(Q_1 + Q_2) &:= (\Delta_R Q_1) + (\Delta_R Q_2)\\
\Delta_R(Q_1 \bowtie Q_2) &:= ((\Delta_R Q_1) \bowtie Q_2) + (Q_1 \bowtie (\Delta_R Q_2))\\
&\quad + ((\Delta_R Q_1) \bowtie (\Delta_R Q_2))\\
\Delta_R(\mathrm{Sum}_{[A_1,A_2,\ldots]} Q) &:= \mathrm{Sum}_{[A_1,A_2,\ldots]}(\Delta_R Q)\\
\Delta_R(var := Q) &:= (var := (Q + \Delta_R Q)) - (var := Q)
\end{aligned}$$

where $-Q$ is syntactic sugar for $(-1) \bowtie Q$. For all other kinds of expressions, $\Delta_R(\cdot) := 0$.

3.2 Delta Evaluation for Batch Updates

In this section, we focus on the techniques for efficient evaluation of delta queries for batches of updates to base relations. For flat queries in which variable assignments involve only values, the delta rules always produce a simpler expression. Here, we define the complexity of a query in terms of its degree [21], which roughly corresponds to the number of referenced base relations. Replacing large base relations by much smaller delta relations reduces evaluation cost, favoring incremental maintenance over recomputation.

For queries with nested aggregates and existential quantification, the derivation rule for variable assignment prescribes recomputing both the old and new results to evaluate the delta, which clearly costs more than recomputing the whole query expression once. In general, these classes of queries might have no benefit from incremental maintenance. But, in many cases, we can specialize this delta rule to achieve efficient maintenance.

3.2.1 Model of Computation

We describe our model of computation to understand the advantages of alternative evaluation strategies. We represent an expression as a tree of operators, which are always evaluated from left to right, in a bottom-up fashion. Information about bound variables flows from left to right through the product operation. For instance, in expression $R(A) \bowtie S(A)$, the term $R(A)$ binds the A variable which is then used to lookup the multiplicity value inside $S(A)$. The evaluation cost for such an expression is $O(|R|)$, where $|R|$ is the number of tuples with a non-zero multiplicity in R.

Our model considers in-memory hash join as a reference join implementation. In this model, the ordering of terms has an impact on query evaluation performance. For example, when $S(A)$ is smaller than $R(A)$, commuting the two terms like $S(A) \bowtie R(A)$ results in fewer memory lookups in R. Note that commuting terms is not always possible – for instance, in expression $R(A) \bowtie A$, the two terms do not commute, unless A is already bound.

3.2.2 Domain extraction

We present a technique, called *domain extraction*, for efficient delta computation of queries with nested aggregates and existential quantification. We explain the idea on the problem of duplicate elimination in bag algebra. We formalize the technique afterwards.

For clarity of the presentation, we introduce $Exists(Q)$ as syntactic sugar for $\mathrm{Sum}_{[\mathrm{sch(Q)}]}((X := Q) \bowtie (X \neq 0))$, where $sch(Q)$ denotes the schema of Q. $Exists(Q)$ changes every non-zero multiplicity inside Q to 1. The delta rule for $Exists$ is $\Delta_R(Exists(Q)) = Exists(Q + \Delta_R Q) - Exists(Q)$.

First, we introduce the notion of domain expressions. A domain expression binds a set of variables with the sole purpose of speeding up the downstream query evaluation. All domain expression tuples have the multiplicity of one. For instance, we can write $R(A,B)$ as $Exists(R(A,B)) \bowtie R(A,B)$ without changing the original query semantics. Note that now $Exists(R(A,B))$ defines the iteration domain during query evaluation rather than $R(A,B)$.

EXAMPLE 3.2. *Consider an SQL query over $R(A,B)$*

```
SELECT DISTINCT A FROM R WHERE B > 3
```

or equivalently $Q := Exists(Sum_{[A]}(R(A,B) \bowtie (B > 3)))$. Let Q_n denotes the nested sum, then $\Delta Q_n := Sum_{[A]}(\Delta R(A,B) \bowtie (B > 3))$ and $\Delta Q := Exists(Q_n + \Delta Q_n) - Exists(Q_n)$. Note that ΔQ recomputes Q twice, once to insert new tuples and then to delete the old ones, which clearly defeats the purpose of incremental computation. Also, $(Q_n + \Delta Q_n)$ might leave unchanged many tuples in Q_n, so deleting those tuples and inserting them again is wasted work.

```
def extractDom(e: Expr): Expr = e match {
  case Plus(A, B) =>
    interDoms(extractDom(A), extractDom(B))
  case Prod(A, B) =>
    unionDoms(extractDom(A), extractDom(B))
  case Sum(gb, A) =>
    val domA  = extractDom(A)
    val domGb = inter(sch(domA), gb)
    if (domGb == gb) domA
    else if (domGb == Nil) 1
    else Exists(Sum(domGb, domA))
  case Assign(v, A) if A.hasRelations =>
    extractDom(A)
  case Rel(_)    =>
    if (e.hasLowCardinality) Exists(e) else 1
  case _         => e
}
```

Figure 1: The domain extraction algorithm. `interDoms` extracts common domains, `unionDoms` merges domains, `inter` is set intersection, and `sch(A)` denotes the schema of expression `A`.

Our goal is to transform ΔQ into an expression that changes only relevant tuples in the delta result. We want to iterate over only those tuples in Q_n whose A values appear in $\Delta R(A,B)$; other tuples are irrelevant for computing ΔQ. To achieve that goal, we capture the domain of A values in ΔR using $Exists(\Delta R(A,B))$[3]. To express only distinct values of A in ΔR, we write:

$$Q_{dom} := Exists(Sum_{[A]}(Exists(\Delta R(A,B)))).$$

The outer Exists keeps the multiplicity of 1. We prepend Q_{dom} to ΔQ in order to restrict the iteration domain.

$$\Delta Q' := Q_{dom} \bowtie (Exists(Q_n + \Delta Q_n) - Exists(Q_n))$$

We can make Q_{dom} more strict by including $(B > 3)$. □

Figure 1 shows the algorithm for domain extraction. The algorithm recursively pushes extracted domains up through the expression tree to bound variables in even larger parts of the expression. At leaf nodes, it identifies relations with low cardinalities that can restrict the iteration domain. We can either use cardinality estimates for each relation or rely on heuristics. We also include terms that can further restrict the domain size, like comparisons, values, and variable assignments over values.

For Sum aggregates, we recursively compute the domain of the subexpression and then, if necessary, reduce its schema to match that of the Sum aggregate. For instance, in Example 3.2, we project $Exists(\Delta R(A,B)) \bowtie (B > 3)$ on column A. If the domain schema is reduced, we enclose the expression with *Exists* to preserve the domain semantics; if the schema is empty, the extracted domain bounds no column and has little effect. When dealing with union, we intersect two subexpressions to find the maximum common domain that can be propagated further up in the tree; for the product operation, we union subexpressions into one common domain.

The domain extraction procedure allows us to revise the delta rule for variable assignments as:

$$\Delta(var := Q) := Q_{dom} \bowtie ((var := Q + \Delta Q) - (var := Q))$$

where $Q_{dom} := \texttt{extractDom}(\Delta Q)$.

3.2.3 Re-evaluation vs. Incremental Computation

For non-nested queries, recursive incremental maintenance has lower parallel complexity than non-incremental evaluation [21], which often reflects in faster sequential execution in practice [22]. For queries with nested aggregates, there are cases when recomputing from scratch is better than computing the delta.

[3]ΔR can contain both insertions and deletions.

Example 3.3. *Consider a query over $R(A,B)$ and $S(B,C)$.*

```
SELECT COUNT(*) FROM R
WHERE R.A < (SELECT COUNT(*) FROM S) AND R.B=10
```

The nested query computes an aggregate value and has no correlation with the outer query. The domain extraction procedure cannot restrict the domain of the delta query for updates to S, so re-evaluation is a better option. We can still speed up the computation by materializing the query piecewise, for instance, by precomputing and maintaining the expression $Sum_{[A]}(R(A,B) \bowtie (B = 10))$. Note that we can incrementally maintain the top-level query for updates to R by materializing the nested query result as a single variable.

Nested queries are often correlated with the outside query. When the correlation involves equality predicates, extracting the domain of the inner query might restrict some of the correlated variables. This range restriction can reduce the maintenance cost. In general, the decision on whether to incrementally maintain or recompute the query result requires a case-by-case cost analysis. In our experiments, we incrementally maintain a query whenever the extracted nested domain binds at least one equality-correlated variable.

3.3 Single-tuple vs. Batch Updates

Incremental programs for single-tuple updates are simpler and easier to optimize than batched incremental programs. The parameters of single-tuple triggers match the tuple's schema to avoid boxing and unboxing of the primitive data types in the input. We can inline these parameters into delta expressions and eliminate one-element loops. In local environments, we can process one batch of updates via repeated calls to single-tuple triggers. Aside from the increased invocation overhead, we identify three reasons why such an approach can be suboptimal.

Preprocessing batches. Preprocessing a batch of updates can merge or eliminate changes to the same input tuples. Static analysis of the query can identify subexpressions that involve solely batch updates (e.g., domain expressions). Then, we can precompute a batch aggregate by keeping only relevant batch columns of the tuples that match query's static conditions. Batch pre-aggregation can produce much fewer tuples, which can significantly decrease the cost of trigger evaluation; smaller batches also reduce communication costs in distributed environments. Batch pre-aggregation can have a limited effect when there is no filtering condition and there is a functional dependency between the aggregated columns and the primary key of the delta relation; then, we can eliminate only updates targeting the same key.

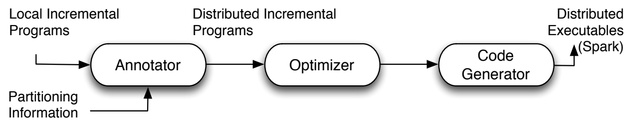
Skipping intermediate views states. When a maintenance trigger evaluates the whole query from scratch, batching can help us to avoid recomputation of intermediate query results. For instance, considering the query of Example 3.3 and updates to S, processing one batch of updates refreshes the inner query result once and triggers one recomputation of the outer query. In contrast, the tuple-at-a-time approach evaluates the outer query on every update.

Cache locality. Processing one update batch, in a tight loop, can improve cache locality and branch prediction for reasonably-sized batches; too large batches can have negative impacts on locality.

In our experiments, we evaluate these trade-offs between single-tuple and batch maintenance for different update sizes.

4. DISTRIBUTED VIEW MAINTENANCE

In this section, we present our compiler that transforms maintenance triggers generated for local execution into maintenance programs optimized for running on large-scale processing platforms with synchronous execution, like Spark or Hadoop. Our approach



Local Incremental Programs
Distributed Incremental Programs
Distributed Executables (Spark)
Annotator
Optimizer
Code Generator
Partitioning Information

Figure 2: Compilation of incremental programs

is general and applies to any input program formed using our query language, not just recursive view maintenance programs.

Distributed Execution Model. We assume a synchronous execution model where one driver node orchestrates job execution among workers, like in Spark or Hadoop. Processing one batch of updates may require several computation stages, where each stage runs view maintenance code in parallel. All workers are stateful, preserve data between stages, and participate in every stage.

Our approach naturally leverages the fault tolerance mechanisms of the underlying execution platform, in our case, the Spark computing framework. Using data checkpointing, we can periodically save intermediate state to reliable storage (HDFS) in order to shorten recovery time. Checkpointing may have detrimental effects on the latency of processing, so the user needs to carefully tune the frequency of checkpointing based on application requirements.

Types of Materialized Views. We classify materialized views depending on the location of their contents. *Local views* are stored and maintained entirely on the driver node. They are suitable for materializing top-level aggregates with small output domains. *Distributed views* have their contents spread over all workers to balance CPU and memory pressure. Each distributed view has an associated partitioning function that maps a tuple to a non-empty set of nodes storing its replicas.

4.1 Compilation Overview

Figure 2 shows the process of transforming a local incremental program into a functionally equivalent, distributed program. The input consists of statements expressed using our query language. To distribute their execution, the compiler relies on partitioning information about each materialized view. Deciding on the optimal view partitioning scheme happens outside of the compilation process; we assume that such partitioning information is available (e.g., provided by the user).

The compilation process consists of three phases. First, we annotate a given input program with partitioning information and, if necessary, introduce operators for exchanging data among distributed nodes in order to preserve the correctness of query evaluation. Next, we optimize the annotated program using a set of simplification rules and heuristics that aim to minimize the number of jobs necessary for processing one update batch. Finally, we generate executable code for a specific processing platform. Only the code generation phase depends on the target platform.

4.2 Well-formed Distributed Programs

The semantics of the query operators, presented in Section 3.1 and Appendix A, cannot be directly translated to distributed environments. For instance, unioning one local and one distributed view has no clear meaning. Even among views of the same type, naïvely executing query operators at each distributed node might yield wrong results. For instance, a natural join between two distributed views produces a correct result only if the views are identically partitioned over the join (common) keys; otherwise, one or both operands need to be repartitioned.

We extend our query language with new location-aware primitives that allow us to construct *well-formed* query expressions. Such expressions preserve the correctness of distributed query evaluation for the given partitioning strategy.

Location Tags. To reason about the semantics and correctness of query evaluation, we annotate query expressions with location tags: 1) `Local` tag denotes the result is located on the driver node; 2) `Dist(`$\mathcal{P}$`)` tag marks the result is distributed among all workers according to partitioning function $\mathcal{P}$; and 3) `Random` tag denotes the result is randomly distributed among all workers.

A relation (materialized view) can take on a `Local`, `Dist`, or `Random` tag. Constants, values, comparisons, and variable assignments involving values are interpreted (virtual) relations whose contents is deterministic and never materialized. The location of such terms is irrelevant from the perspective of query evaluation, and they can freely participate in both local and distributed expressions; in distributed settings, one can view these terms as fully replicated.

Location Transformers. To support distributed execution, we extend the language with new operators for manipulating location tags and exchanging data over the network.

- $\texttt{Repart}_{\mathcal{P}_2}(Q^{\{\texttt{Dist}(\mathcal{P}_1),\texttt{Random}\}}) = Q^{\texttt{Dist}(\mathcal{P}_2)}$
  Partition the distributed result of Q using function $\mathcal{P}_2$.
- $\texttt{Scatter}_{\mathcal{P}}(Q^{\texttt{Local}}) = Q^{\texttt{Dist}(\mathcal{P})}$
  Partition the local result of Q using function $\mathcal{P}$.
- $\texttt{Gather}(Q^{\{\texttt{Dist}(\mathcal{P}),\texttt{Random}\}}) = Q^{\texttt{Local}}$
  Aggregate the distributed result of Q on the driver node.

The location transformers are the only mechanism for exchanging data among nodes. `Repart` and `Gather` operate over distributed expressions, while `Scatter` supports only local expressions.

Distributed Query Operators. We extend the semantics of our query operators with location tags.

- Relation $R(A_1, A_2, ...)^T$ stores the contents at location T.
- Bag union $Q_1^T + Q_2^T$ merges tuples either locally or in parallel on every node, and the result retains tag T. Requires the same location tag for both operands.
- Natural join $Q_1^T \bowtie Q_2^T$ has the usual semantics when $T = \texttt{Local}$. For distributed evaluation, both operands need to be partitioned on the join keys; the result is distributed and consists of locally evaluated joins on every node. Joins on `Random` are disallowed.
- $\text{Sum}_{[A_1,A_2,...]} Q^T$, when $T = \texttt{Dist}(\mathcal{P})$, computes partial aggregates on every node. The result has tag T only if Q is key partitioned on one of the group-by columns; otherwise, we annotate with a `Random` tag. In other cases, the result retains tag T.

All other language constructs – constants, values, comparisons, and variable assignments – are location independent and their semantics remain unchanged.

Next, we present an algorithm for transforming a local program into a well-formed distributed program based on the given partitioning information. The algorithm annotates and possibly extends the expression trees of the statements to preserve the semantics of each operator. For each statement, we start by assigning location tags to all relational terms in the expression tree. Then, in a bottom-up fashion, we annotate each node of the tree with a location tag. We introduce location transformers where necessary to preserve the semantics and correctness of each query operator, as discussed above. Upon reaching the root node, we ensure that the RHS query expression evaluates at the same location where the target LHS view is materialized, and, if necessary, introduce a `Gather` or `Scatter` transformer. To obtain a well-formed distributed program, we apply this procedure to every input statement.

EXAMPLE 4.1. *Let us construct a well-formed statement for*

$$M(A) \mathrel{+}= Sum_{[A]}(M_1(A,B) \bowtie M_2(A,B))$$

when $M(A)$ and $M_1(A,B)$ are partitioned by A, while $M_2(A,B)$ is partitioned by B. We associate location tags to the materialized views. We use $M(A)^{[A]}$ to denote that $M(A)$ is partitioned on column A. Since the join operands have incompatible location tags, we introduce `Repart` *around one of the operands (e.g., left):*

$$\texttt{Repart}_{[B]}(M_1(A,B)^{[A]})^{[B]} \bowtie M_2(A,B)^{[B]}$$

The join result remains partitioned on B. The Sum expression computes partial aggregates grouped by column A. Such an expression cannot have location tag $[B]$ since B is not in the output schema; so, we assign a `Random` *tag to the expression. Now, we have reached the root of the expression tree. We need to ensure that the RHS expression has the same location tag as the target view. So, adding a* `Repart` *transformer produces a well-formed statement:*

$$M(A)^{[A]} \mathrel{+}= \texttt{Repart}_{[A]}(Sum_{[A]}(\texttt{Repart}_{[B]}(M_1(A,B)^{[A]})^{[B]} \bowtie M_2(A,B)^{[B]})^{Random})^{[A]} \quad \square$$

The algorithm for constructing well-formed expressions has no associated cost metrics and might produce suboptimal solutions. In the above example, executing the final statement requires two communication rounds (`Repart`s) between the driver and workers. Such communication overhead is unnecessary – if we repartition the other join operand, we produce an equivalent, less expensive well-formed statement with only one communication round:

$$M(A)^{[A]} \mathrel{+}= \text{Sum}_{[A]}(M_1(A,B)^{[A]} \bowtie \texttt{Repart}_{[A]}(M_2(A,B)^{[B]})^{[A]})^{[A]}$$

To optimize well-formed programs we rely on a set of simplification and heuristic rules, which we present next.

4.3 Optimizing Distributed Programs

In this section, we describe how to optimize well-formed programs in order to minimize communication and processing costs. We divide this task into two stages. The first stage relies on a simple cost-based model to simplify each individual statement. The second stage exploits commonalities among statements to avoid redundant communication and minimize the number of jobs needed to execute the given program.

4.3.1 Intra-Statement Optimization

Intra-statement optimization aims to minimize the amount of network traffic required to execute one well-formed statement. Our cost model takes the number of communication rounds as the basic metric for cost comparison. Each location transformer (`Repart`, `Scatter`, and `Gather`) shuffles data over the network, so statements containing fewer of them require less communication. Our cost model also relies on few heuristics to resolve ties between statements with the same cost. We reshuffle expressions that involve batch updates rather than whole materialized views as the formers are usually smaller in size; we favor expressions with fewer `Gather` transformers to distribute computation as much as possible.

The optimizer uses a trial and error approach to recursively optimize a given statement. It tries to push each location transformer down the expression tree, following the rules from Figure 3. Note that these rules might produce more expensive expressions, in which case the algorithm backtracks. During each optimization step, the compiler tries to simplify the current expression using the rules from Figure 4. Each simplification rules always produces an equivalent expression with fewer location transformers.

$$\begin{aligned}
\texttt{Repart}_{\mathcal{P}}(Q_1 \bowtie Q_2) &\Leftrightarrow \texttt{Repart}_{\mathcal{P}}(Q_1) \bowtie \texttt{Repart}_{\mathcal{P}}(Q_2)\\
\texttt{Repart}_{\mathcal{P}}(Q_1 + Q_2) &\Leftrightarrow \texttt{Repart}_{\mathcal{P}}(Q_1) + \texttt{Repart}_{\mathcal{P}}(Q_2)\\
\texttt{Repart}_{\mathcal{P}}(\text{Sum}_{[A_1,A_2,\ldots]}(Q)) &\Leftrightarrow \text{Sum}_{[A_1,A_2,\ldots]}(\texttt{Repart}_{\mathcal{P}}(Q))\\
\texttt{Repart}_{\mathcal{P}}(var := Q) &\Leftrightarrow (var := \texttt{Repart}_{\mathcal{P}}(Q))
\end{aligned}$$

Figure 3: Bidirectional optimization rules for `Repart`. The same rules apply to `Scatter` and `Gather`.

$$\begin{aligned}
\texttt{Repart}_{\mathcal{P}}(Q^{\texttt{Dist}(\mathcal{P})}) &\Rightarrow Q^{\texttt{Dist}(\mathcal{P})}\\
\texttt{Gather}(Q^{\texttt{Local}}) &\Rightarrow Q^{\texttt{Local}}\\
\texttt{Repart}_{\mathcal{P}_1} \circ \texttt{Repart}_{\mathcal{P}_2} &\Rightarrow \texttt{Repart}_{\mathcal{P}_1}\\
\texttt{Repart}_{\mathcal{P}_1} \circ \texttt{Scatter}_{\mathcal{P}_2} &\Rightarrow \texttt{Scatter}_{\mathcal{P}_1}\\
\texttt{Gather} \circ \texttt{Repart}_{\mathcal{P}} &\Rightarrow \texttt{Gather}\\
\texttt{Gather} \circ \texttt{Scatter}_{\mathcal{P}} &\Rightarrow \texttt{Gather}\\
\texttt{Scatter}_{\mathcal{P}} \circ \texttt{Gather} &\Rightarrow \texttt{Repart}_{\mathcal{P}}
\end{aligned}$$

Figure 4: Simplification rules for location transformers. The $\circ$ sign denotes operator composition.

In Example 4.1, the optimizer pushes the outer $\texttt{Repart}_{[A]}$ through Sum and $\bowtie$, and then simplifies $\texttt{Repart}_{[A]} \circ \texttt{Repart}_{[B]}$ to $\texttt{Repart}_{[A]}$. The optimized statement requires only one communication round.

4.3.2 Inter-Statement Optimization

Location transformers represent natural pipeline breakers in query evaluation since they need to materialize and shuffle their contents before continuing processing. In this section, we show how to analyze inter-statement dependencies to minimize the number of pipeline breakers and their communication overhead.

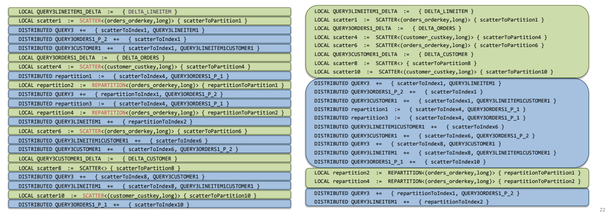
Single Transformer Form. To facilitate inter-statement analysis, we first convert a given program into single transformer form where each statement has at most one location transformer. The transformer, if present, always references one materialized view. In other words, we normalize the input program by: 1) materializing the contents being transformed (if not already materialized), 2) extracting the location transformers targeting the materialized contents into separate statements, and 3) updating the affected statements with new references. To achieve that, we recursively bottom-up traverse the expression tree of every statement.

Single transformer form sets clear boundaries around the contents that needs to be communicated, which eases the implementation of further optimizations. We apply common subexpression elimination and dead code elimination to detect expressions shared among statements and to eliminate redundant network transfers. In contrast to the classical compiler optimizations, our routines are aware of the location where each expression is executed.

Statement Execution Mode. The location tag associated with each statement determines where and how that statement is going to be executed. Statements targeting local materialized views are, as expected, executed at the driver node in *local mode*. Statements involving distributed views run in *distributed mode*, where the driver initiates the computation.

All location transformations run in local mode since the driver governs their execution. But, materializing the contents to be reshuffled can happen in both local and distributed mode. For instance, preparing contents for `Scatter` takes place on the driver node, while for `Repart` and `Gather` happens on every worker node.

Statement Blocks. Distributed statements are more expensive to execute than local statements. To run a distributed statement, the driver needs to serialize the task closure, ship it to all the work-



```
LOCAL QUERY3LINEITEM1_DELTA  :=   { DELTA_LINEITEM }
LOCAL scatter1  :=  SCATTER<(orders_orderkey,long)> { scatterToPartition1 }
DISTRIBUTED QUERY3  +=   { scatterToIndex1, QUERY3LINEITEM1 }
DISTRIBUTED QUERY3ORDERS1_P_2  +=   { scatterToIndex1 }
DISTRIBUTED QUERY3CUSTOMER1  +=   { scatterToIndex1, QUERY3LINEITEM1CUSTOMER1 }
LOCAL QUERY3ORDERS1_DELTA  :=   { DELTA_ORDERS }
LOCAL scatter4  :=  SCATTER<(customer_custkey,long)> { scatterToPartition4 }
DISTRIBUTED repartition1  :=   { scatterToIndex4, QUERY3ORDERS1_P_1 }
LOCAL repartition2  :=  REPARTITION<(orders_orderkey,long)> { repartitionToPartition1 }
DISTRIBUTED QUERY3  +=   { repartitionToIndex1, QUERY3ORDERS1_P_2 }
DISTRIBUTED repartition3  :=   { scatterToIndex4, QUERY3ORDERS1_P_1 }
LOCAL repartition4  :=  REPARTITION<(orders_orderkey,long)> { repartitionToPartition2 }
DISTRIBUTED QUERY3LINEITEM1  +=   { repartitionToIndex2 }
LOCAL scatter6  :=  SCATTER<(orders_orderkey,long)> { scatterToPartition6 }
DISTRIBUTED QUERY3LINEITEM1CUSTOMER1  +=   { scatterToIndex6 }
DISTRIBUTED QUERY3CUSTOMER1  +=   { scatterToIndex6, QUERY3ORDERS1_P_2 }
LOCAL QUERY3CUSTOMER1_DELTA  :=   { DELTA_CUSTOMER }
LOCAL scatter8  :=  SCATTER<> { scatterToPartition8 }
DISTRIBUTED QUERY3  +=   { scatterToIndex8, QUERY3CUSTOMER1 }
DISTRIBUTED QUERY3LINEITEM1  +=   { scatterToIndex8, QUERY3LINEITEM1CUSTOMER1 }
LOCAL scatter10  :=  SCATTER<(customer_custkey,long)> { scatterToPartition10 }
DISTRIBUTED QUERY3ORDERS1_P_1  +=   { scatterToIndex10 }
```

```
LOCAL QUERY3LINEITEM1_DELTA  :=   { DELTA_LINEITEM }
LOCAL scatter1  :=  SCATTER<(orders_orderkey,long)> { scatterToPartition1 }
LOCAL QUERY3ORDERS1_DELTA  :=   { DELTA_ORDERS }
LOCAL scatter4  :=  SCATTER<(customer_custkey,long)> { scatterToPartition4 }
LOCAL scatter6  :=  SCATTER<(orders_orderkey,long)> { scatterToPartition6 }
LOCAL QUERY3CUSTOMER1_DELTA  :=   { DELTA_CUSTOMER }
LOCAL scatter8  :=  SCATTER<> { scatterToPartition8 }
LOCAL scatter10  :=  SCATTER<(customer_custkey,long)> { scatterToPartition10 }
DISTRIBUTED QUERY3  +=   { scatterToIndex1, QUERY3LINEITEM1 }
DISTRIBUTED QUERY3ORDERS1_P_2  +=   { scatterToIndex1 }
DISTRIBUTED QUERY3CUSTOMER1  +=   { scatterToIndex1, QUERY3LINEITEM1CUSTOMER1 }
DISTRIBUTED repartition1  :=   { scatterToIndex4, QUERY3ORDERS1_P_1 }
DISTRIBUTED repartition3  :=   { scatterToIndex4, QUERY3ORDERS1_P_1 }
DISTRIBUTED QUERY3LINEITEM1CUSTOMER1  +=   { scatterToIndex6 }
DISTRIBUTED QUERY3CUSTOMER1  +=   { scatterToIndex6, QUERY3ORDERS1_P_2 }
DISTRIBUTED QUERY3  +=   { scatterToIndex8, QUERY3CUSTOMER1 }
DISTRIBUTED QUERY3LINEITEM1  +=   { scatterToIndex8, QUERY3LINEITEM1CUSTOMER1 }
DISTRIBUTED QUERY3ORDERS1_P_1  +=   { scatterToIndex10 }
LOCAL repartition2  :=  REPARTITION<(orders_orderkey,long)> { repartitionToPartition1 }
LOCAL repartition4  :=  REPARTITION<(orders_orderkey,long)> { repartitionToPartition2 }
DISTRIBUTED QUERY3  +=   { repartitionToIndex1, QUERY3ORDERS1_P_2 }
DISTRIBUTED QUERY3LINEITEM1  +=   { repartitionToIndex2 }
```

Figure 5: The block fusion effect in TPC-H Q3: before and after. Green blocks are local, blue blocks are distributed.

ers, and wait for the completion of each one of them. For short-running tasks, and recursive view maintenance is often such, non-processing overheads can easily dominate in the execution time.

To amortize the cost of executing distributed statements, we pack them together into processing units called *statement blocks*. A statement block consists of a sequence of distributed statements that can be executed at once on every node without comprising the program correctness. Apart from distributed blocks, we also introduce blocks of local statements whose purpose is to determine which network operations can be batched together (the driver initiates `Repart`, `Scatter`, and `Gather` in local mode).

Data-flow dependencies among statements prevent arbitrary reorderings of statements and blocks. In Appendix C.3, we show methods for checking the commutativity of statements and blocks.

Block Fusion Algorithm. We prefer program execution plans with as few statement blocks as possible. Here, we describe an algorithm that reorders and merges together consecutive blocks to minimize their number. Appendix C.3 presents the algorithm.

First, we promote each statement into a separate block that keeps statement's execution mode (local or distributed). Then, the algorithm tries to fuse together the first block with the others that share the same execution mode and commute with all intermediate blocks. On success, the algorithm merges the new block sequence recursively; otherwise, it handles the remaining blocks recursively.

Figure 5 visualizes the effects of block fusion on the incremental program of TPC-H Q3. Before running the algorithm, the annotated input program contained 10 local and 12 distributed statement blocks. After reordering and merging these blocks, the algorithm outputs only 2 local and 2 distributed compound blocks.

4.4 Code Generation

Statement blocks considerably simplify code generation. Isolating distributed blocks enables workers to safely run code generated for single-node execution on their local data partitions. Pure local statements without transformers also correspond to unmodified single-node code. Distributed code generation, thus, relies to a great extent on single-node code generation.

Code generated for location transformers uses platform-specific communication primitives for exchanging data. To minimize network overhead, we encapsulate transformers of the same type into one compound request per block. For instance, we coalesce multiple `Scatter` transformers and their materialized data into just one `Scatter` request that uses a container data structure.

5. IMPLEMENTING VIEW MAINTENANCE

In this section, we describe how to compile queries of our language into low-level code that uses specialized data structures.

5.1 From Queries to Native Code

DBToaster compiles maintenance programs into imperative or functional code optimized for the execution in local mode (C++) or distributed mode (Scala for Spark). The compiler relies on the model of computation described in Section 3. As discussed, the query language exploits the notion of information flow, which is common in programming languages. The information about bound variables always flows from left to right during query evaluation, which eases compilation of the language construct. In generated code, we replace all high-level operators, like natural joins, unions, etc., with concrete operations over the underlying data structures.

When compiling a relational term, we distinguish several cases: (1) if all its variables are free, we transform it into a `foreach` loop that iterates over the whole collection; (2) if all its variables are bound, we replace it with a `get` (lookup) operation; (3) otherwise, we form a `slice` operation that iterates over only the elements matching the current values of the bound variables. Note that cases (2) and (3) may benefit from specialized index structures.

We use continuation passing style [8] to facilitate code generation and avoid intermediate materializations, such as redundant computations of bag unions and aggregates. The compilation process relies on an extensible compiler library, called LMS [36], to perform both classical optimizations, like common subexpression elimination, dead code elimination, loop unrolling and fusion, and also domain-specific optimizations, like data-structure specialization and automatic indexing.

Our compiler optimizes incremental programs for single-tuple processing. It specializes the parameters of single-tuple triggers to the concrete primitive types of the updated relation. Then, the native compiler can treat such parameters as constants and move them out of loops and closures when possible. Our compiler eliminates loops around one-element batches and uses primitive type variables rather than maps to store intermediate materializations.

5.2 Data Structure Specialization

The design of the data structure for storing materialized views depends on whether the view contents can change over time and the types of operations that need to be supported. Materialized views defined over static base relations are immutable collections of records stored in fixed-size arrays with fast lookup and scan operations. In typical streaming scenarios, however, updates to mutable base relations can be fast-moving and unpredictable, indicating that any array-based solution for storing tuples of dynamic materialized views might be expensive either in terms of the memory usage or maintenance overhead.

We materialize the contents of dynamic materialized views inside record pools, shown in Figure 6. One record pool stores records of the same format inside main memory and dynamically adapts its size to match the current working set. The pool keeps track of available free slots to facilitate future memory allocations and ease memory management. We specialize the format of pool records at compile time. Each record contains key fields corresponding to the schema of the materialized expression and value fields storing tuple multiplicities. The key fields uniquely identify each record.

5.2.1 *Automatic Index Support*

We can associate multiple index structures with one record pool, as shown in Figure 6. Each index structure provides a fast path to the records that match a given condition. We use unique hash indexes to provide fast lookups and non-unique hash indexes for slice operations[4]. Both indexes maintain an overflow linked list for each bucket. To shorten scanning in one bucket, non-unique hash indexes cluster records that share the same key. Pool records keep

[4]Studying other index types, like B++ trees (for range operations) or binary heaps (for min/max), we leave for future work.

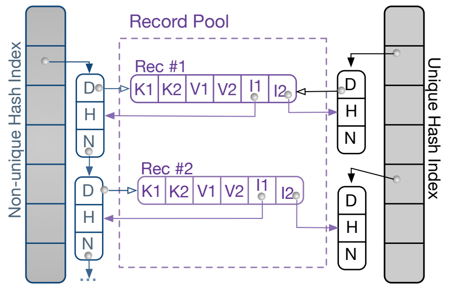


Figure 6: Multi-indexed data structure used for materialization. Each bucket (shaded cells) has a linked list of collisions. Legend: (D)ata, (H)ash, (N)ext, (K)ey, (V)alue, and (I)ndex.

back-references to their indexes to avoid hash re-computation and additional lookups during update and delete operations.

Our compiler analyzes the following access patterns: 1) scan over the entire collection (`foreach`), 2) lookup for a given unique key (`get`, `update`, `delete`), and 3) index scan for a given non-unique key (`slice`). We build a unique hash index for `get` operations, that is, when all lookup keys are bound at evaluation time. The same rule applies to `update`, `insert`, and `delete` operations. For `slice` operations, we create a non-unique index over the variables bound at evaluation time. When the access pattern analysis detects only `foreach` operations, we omit index creation.

DBToaster creates all relevant index structures. From our experience, most data structures produced during recursive compilation have only few indexes. For instance, the materialized views produced compiling the TPC-H queries usually have zero or one secondary indexes, with rare exceptions of up to three non-unique indexes. Our empirical results indicate that the benefit of creating these indexes greatly outperforms their maintenance overheads.

5.2.2 *Column-oriented layout*

Record pools keep tuples in a row-oriented format. Materialized views store aggregated results in which all unused attributes are projected away during query compilation. Thus, during query evaluation, each access to one record likely references all its fields.

The row-oriented layout is, however, unsuitable for efficient data serialization and deserialization, which are important considerations in distributed query evaluation. To speed up these operations, the compiler creates array-based data structures for storing serializable data in columnar mode. It also generates specialized transformers for switching between row- and column-oriented formats.

We also use columnar data structures for storing input batches. Batched delta processing often starts by filtering out tuples that do not match query's static conditions. As these conditions are often simple (e.g., $A > 2$), using a columnar representation can improve cache locality. After filtering, we typically aggregate input batches to remove unused columns and store the result in a record pool.

6. EXPERIMENTS

We evaluate the performance of recursive incremental view maintenance for batch updates of different sizes in local and distributed settings. In the single-node evaluation, we analyze the throughput and cache locality of C++ incremental programs generated using the DBToaster Release 2.2 [2]. In the distributed evaluation, we scale out incremental view maintenance to hundreds of Spark workers. Our experimental results show that:

- In local mode, view maintenance code specialized for tuple-at-a-time processing can outperform or be on par with batched programs for almost half of our queries.
- Preprocessing input batches can boost the performance of incremental computation by multiple orders of magnitude.
- Large batches can have negative impacts on cache locality. In local mode, the throughput of most of our queries peaks for batches with $1,000 - 10,000$ tuples.
- In distributed mode, we show that our approach can scale to hundreds of workers for queries of various complexities, while processing input batches with few second latencies.

Experimental Setup. We run single-node experiments on an Intel Xeon E5-2630L @ 2.40GHz server with 2×6 cores, each with 2 hardware threads, 15MB of cache, 256GB of DDR3 RAM, and Ubuntu 14.04.2 LTS. We compile generated C++ programs using GCC 4.8.4. For distributed experiments, we use 100 such server instances connected via a full-duplex 10GbE network and running Spark 1.6.1 and YARN 2.7.1. We generate Scala programs for running on Spark and compile using Scala 2.10.4.

Query and Data Workload. Our query workload consists of the TPC-H queries that are modified for streaming scenarios [22] and a subset of TPC-DS queries from [23] (excluding four queries with the OVER clause, which we currently do not support). We run these queries over data streams synthesized from TPC-H and TPC-DS databases by interleaving insertions to the base relations in a round-robin fashion. We process 10GB streams in local mode and 500GB streams in distributed mode. We run experiments with a one-hour timeout on query execution, not counting loading of streams into memory and forming input batches of a given size.

6.1 Single-node Evaluation

Our local experiments compare tuple-at-a-time and batched incremental programs. The former have triggers with tuple fields as function parameters, which can be inlined into delta computation; the latter consist of triggers accepting one arbitrary-sized columnar batch. In both cases, we generate single-threaded C++ code.

Batched incremental programs have extra loops inside triggers for processing input batches. To avoid redundant iterations over the whole batch, we materialize input tuples that match query's static conditions, retaining only the attributes (columns) used in incremental evaluation. These pre-aggregated batches are smaller in size due to having fewer attributes and, potentially, fewer matching tuples, which can significantly affect view maintenance costs. In contrast, single-tuple triggers avoid materialization of input batches.

6.1.1 *Batch Size vs. Throughput*

Figure 7 shows the normalized throughput of batched incremental processing of the TPC-H queries for different batch sizes using the tuple-at-a-time performance as the baseline. For ease of presentation, we use two graphs with different y-axis scales.

For almost half of our queries, batched incremental processing performs worse or just marginally better than specialized tuple-at-a-time processing. This result comes at no surprise once we start analyzing the pre-aggregated batches of these queries for updates to their largest (and usually most expensive) relation. For instance, Q5, Q9, and Q18 aggregate `LINEITEM` deltas by `orderkey` and, possibly, some other fields; Q16 aggregates `PARTSUPP` deltas by the primary key (`partkey`, `suppkey`). In these cases, batch pre-aggregation retains (almost) the same number of input tuples, bringing no performance improvements; on the contrary, it introduces extra materialization and looping overheads. For two-way join queries, Q4, Q12, and Q13, the simplicity of their view maintenance code makes these overheads particularly pronounced.

Batch pre-aggregation keeps only input tuples that match static query conditions. This step can speed up the rest of view maintenance code, bringing improvements that depend on the selectivity of query predicates. For instance, preprocessing in Q3, Q7, Q8, Q10, and Q14 filters out tuples of `LINEITEM` and `ORDERS` batches and yields improvements from 19% in Q7 to 306% in Q8.

Batch pre-aggregation can project input tuples onto a set of attributes with small active domains. For instance, Q1 projects a `LINEITEM` batch onto columns containing only few possible values, which enables cheap maintenance of the final 8 aggregates. So, the single-tuple implementation performs more maintenance work per input tuple. For Q2 and Q19, batch filtering and projection can give up to 1.3x and 5.1x better performance. This benefit can increase for queries with more complex trigger functions. For instance, Q22 filters and projects an `ORDERS` batch on `custkey`, while Q20 filters and projects `PARTSUPP` and `LINEITEM` batches on `suppkey`. In both cases, the projected columns have much smaller domains, bringing significant improvements, $2,243$x in Q20 and $4,319$x in Q22.

Incremental programs for single-tuples updates are easier to optimize than their batched counterparts due to having simpler input and fewer loops in trigger bodies. This virtue emerges in Q17 and Q21. For these queries, our compiler fails to factorize common subexpressions as efficiently as during the single-tuple compilation, which causes same expressions to be evaluated twice.

For Q11 and Q15, incremental view maintenance is more expensive than re-evaluation due to inequality-based nested aggregates. Increasing batch sizes results in fewer re-evaluations, which increases the overall throughput at the expense of higher latencies.

Bulk processing amortizes the overhead of invoking trigger functions. For instance, maintaining a single aggregate over `LINEITEM` in Q6 with batches of $10,000$ tuples can bring up to 2x better performance than the single-tuple execution. However, these results hold only when function inlining is disabled. Since the single-tuple trigger body of Q6 consists of just one conditional statement, the C++ compiler usually decides to inline this computation, causing single-tuple execution to always outperform batched execution.

Comparison with PostgreSQL. Figure 8 compares the throughput of recursive incremental processing in C++ and re-evaluation and classical incremental view maintenance in PostgreSQL for TPC-H Q17. We implement incremental processing in PostgreSQL using the domain extraction procedure described in Section 3.2. The results show that the generated code outperforms PostgreSQL re-evaluation from 233x to $14,181$x and classical incremental view maintenance from 120x to $10,659$x, for different batch sizes.

Appendix B contains the performance numbers for the TPC-H and TPC-DS queries from our workload. The results show that, in all but four cases, recursive view maintenance outperforms classical view maintenance by orders of magnitude, even when processing large batches, for which the database system is optimized for.

Memory requirements. Recursive incremental view maintenance materializes auxiliary views in order to speed up the work required to keep all views fresh. The sizes of these auxiliary views created to maintain a given query, in general, depend on the query structure. Our query workloads, TPC-H and TPC-DS, are based on the star schema with one large fact table and several dimension tables. In such cases, auxiliary materialized views cannot have more tuples than the fact table due to integrity constraints. In practice, the sizes of materialized views are much smaller than the size of the fact table as their view definition queries often involve static filtering conditions. In addition, materialized views discard columns unused in a given query and aggregate over the remaining columns.

6.2 Distributed Evaluation

For distributed experiments, we use Spark [38] for parallelizing the execution of incremental view maintenance code. Spark offers a synchronous model of computation in which the driver governs job execution and coordination with workers.

We execute a subset of the TPC-H queries with different complexities on a 500GB stream of tuples. We chunk the input stream into batches of a given size, and, for each batch, we run one or more Spark jobs to refresh the materialized view. To avoid scalability bottlenecks caused by the driver handling all input data, we simulate a system in which every worker receives, independently of the driver and other workers, a fraction of the input stream. In our experiments, we ensure that each worker gets a roughly equal *random* partition of every batch. Each worker preloads its batch partitions before starting the experiment.

Materialized views are either stored locally on the driver or distributed among workers. The decision on how to partition materialized views in order to minimize their maintenance costs is a challenging problem, which might benefit from previous work on database partitioning [15, 31]. We leave this question for future work. For this paper, we rely on a simple heuristics rule: we partition materialized views on the primary key of a base table appearing in the view schema (e.g., `orderkey`); if there are multiple such primary keys of base tables (e.g., `orderkey`, `custkey`), we partition on the one with the highest cardinality (`orderkey`); otherwise, if there are no primary keys in the view schema, we assume the final aggregate has a small domain and can be stored on the driver.

Query Complexity in Spark. Generated Spark code runs a sequence of jobs in order to perform incremental view maintenance. Each job consists of multiple stages (e.g., map-reduce phases), and each stage corresponds to one block of distributed statements, defined in Section 4.3. The number of jobs and stages for a given query depends on the query structure and the provided partitioning information. Appendix C.1 shows the complexity of the TPC-H queries in terms of the number of jobs and stages necessary for processing one update batch, for the above partitioning strategy.

6.2.1 Weak Scalability

We evaluate the scalability of our approach when each worker receives batch partitions of size $100,000$. Figure 9 shows the measured latency and throughput for a subset of the TPC-H queries.

Q6 computes a single aggregate over `LINEITEM`. Given the initial random distribution of batches and small query output size, we create one stage during which each worker computes a partial aggregate of its batch partition, and then, we sum these values up to update the final result at the driver. The purpose of running Q6 is to measure Spark synchronization overheads as a function of the number of workers. The query requires minimal network communication as each worker sends one 64-bit value per batch. Also each worker spends negligible time aggregating $100,000$ tuples (6 ms on average), thus the results from Figure 9a are close to pure synchronization overheads of Spark. The median latency of processing a batch of size $(100,000 \times \#\text{workers})$ increases from 65 ms for 50 workers to 386 ms for $1,000$ workers, while the throughput rises up to 267 million tuples per sec for 600 workers. Both metrics suffer from synchronization costs, which increase with more workers.

Q17 computes a two-way join with an equality-correlated nested aggregate. Incremental computation of Q17 relies on the domain extraction procedure described in Section 3.2. We partition both base relations on `partkey` and store the result at the driver. The execution graphs consists of two stages. The first stage pre-aggregates batch partitions and shuffles the result on `partkey`. The second

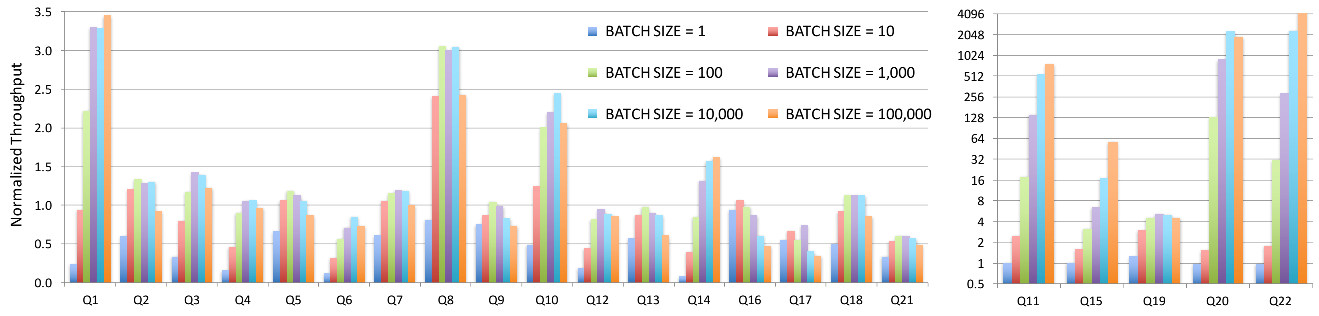


Figure 7: Normalized throughput of the TPC-H queries for different batch sizes with single-tuple execution as the baseline.

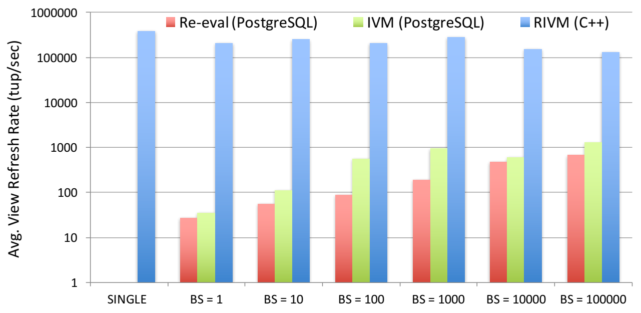


Figure 8: Throughput comparison for TPC-H Q17 of re-evaluation and incremental maintenance in PostgreSQL and recursive incremental maintenance in generated C++ code for different batch sizes. SINGLE denotes specialized single-tuple processing in C++.

stage refreshes the base relations and aggregates partial results at the driver to update the final result.

Figure 9b shows the performance of incremental maintenance of Q17. The throughput rises up to 600 nodes, while the median latency increases from 1.3s for 50 workers to 4s for 800 workers. Q17 achieves higher latency than Q6 due to several reasons: 1) workers perform more expensive view maintenance (526 ms on average), 2) the shuffling phase requires serialization and deserialization of data, writing to local disks, and reading from remote locations, and 3) more processing stages incur more synchronization overheads. Pre-aggregation of input batches reduces the amount of shuffled data from 7.6 MB to 1.8 MB per worker. This amount remains constant regardless of the number of workers.

Figure 9c shows that the average throughput of Q3 increases up to 400 workers. Compared with Q17, the median latency of Q3 is lower at smaller scales and almost identical when using more workers. Q3 uses one additional stage to replicate pre-aggregated `CUSTOMER` deltas and join them with materialized views partitioned over `orderkey`. The amount of shuffled data per worker grows with the batch size, from 439 KB for 50 workers to 2.4 MB for 1,000 workers. The trigger processing time per worker (excluding all other overheads) changes on average from 120 ms for 50 workers to 305 ms for 1,000 workers, with less than 10% deviation among workers in both cases. Thus, at larger scales, the increased shuffling cost dominates the processing time.

Q7 is one of the most complex queries in our workload from the perspective of incremental view maintenance. The driver stores the top-level result and runs three jobs to process one batch of updates. The median latency grows more rapidly compared to other queries, from 1.5s on 50 workers to 16.9s on 800 workers. The average running time of the update triggers per worker also increases but more steadily, from 0.6s to 3.7s, while the amount of shuffled data per worker grows from 2.1 MB to 8.4 MB. This increased network communication induces higher latencies and brings down the average throughput beyond 400 workers.

In all these cases, using more workers increases the variability of latency. From our experience with using Spark, stragglers can often prolong stage computation time by a factor of $1.5-3$x despite almost perfect load balancing. We also observe that the straggler effect is more pronounced with queries shuffling relatively large amounts of data, such as Q3 and Q7. Examining logs and reported runtime metrics gives no reasonable explanation for such behavior.

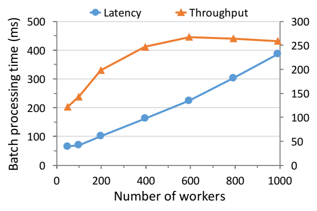
6.2.2 *Strong Scalability*

We measure the scalability of our incremental technique for constant batch sizes and varying numbers of workers. Figure 10 shows the measured throughput for a subset of the TPC-H queries. We use batches with 50, 100, 200, and 400 million tuples to ensure enough parallelizable work inside update triggers. Figure 11 in Appendix C shows results for more TPC-H queries. We compare our approach against re-evaluation using Spark SQL for batches with 400 million tuples. Note that Spark SQL can handle only flat queries.

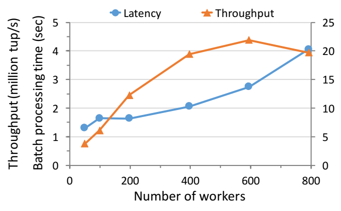
Figure 10a shows that the median latency of Q6 decreases with more workers until the cost of synchronization becomes comparable with the cost of batch processing. Processing 100 million tuples using 100 workers takes on average 14 ms per worker, leaving no opportunities for further parallelization. For the four batch sizes, the lowest median latencies are 98 ms, 130 ms, 153 ms, and 211 ms. Re-evaluating Q6 on each update using Spark SQL achieves the median latency of 32.8 seconds per batch on 100 nodes.

The incremental view maintenance of Q17 scales almost linearly, as shown in Figure 10b. The median latency of processing one batch of 400 million tuples declines by 10.7x (from 68.5s to 6.4s) when using 16x more resources (from 50 to 800 workers). Here, the amount of shuffled data per worker decreases from 77.4 MB to 4.2 MB, while the trigger processing time per worker drops from 27.3s to 2.3s. We observe similar effects when processing smaller batches. When using 800 workers, the median latency of processing different batch sizes varies from 3.6s to 6.4s.

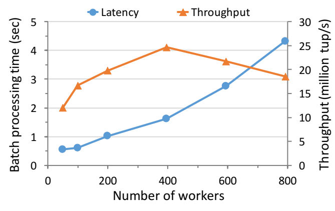
Figure 10c shows that the median latency of processing Q3 decreases with more nodes, from 30s with 25 workers to 4.2s with 400 workers for input batches with 400 million tuples. Adding more workers decreases the amount of work performed inside each trigger and the amount of shuffled data per worker while at the same time increases synchronization overheads. Using larger batches creates more parallelizable work and enables scalable execution across more nodes, as shown in Figure 10c. Re-evaluating Q3 using Spark SQL performs slower than our incremental program for the corresponding batch size, from 8.5x using 25 workers to 20.9x using 400 workers.



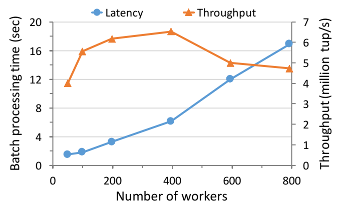
(a) Q6



(b) Q17

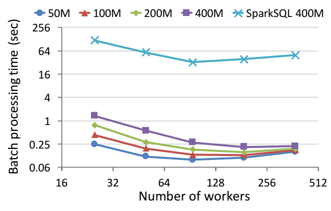


(c) Q3

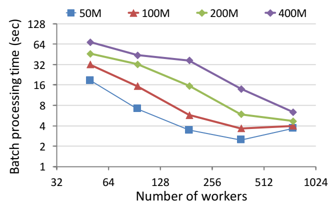


(d) Q7

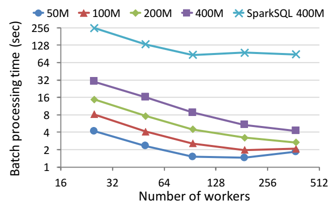
Figure 9: Weak scalability of the incremental view maintenance of TPC-H queries. Each worker processes batches of size 100, 000.



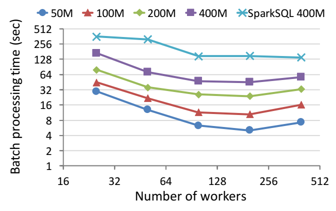
(a) Q6



(b) Q17



(c) Q3



(d) Q7

Figure 10: Strong scalability of the incremental view maintenance of TPC-H queries for different batch sizes (in million of tuples). Appendix C contains results for more TPC-H queries.

Q7 requires the most expensive maintenance work among the four TPC-H queries. The median latency of processing 100 million tuples drops from 44.8s with 25 workers to 10.4s with 200 workers. Beyond 200 workers, even though the size of shuffled data per worker decreases, managing large data creates stragglers that prolong execution time. Compared with Spark SQL re-evaluation, our approaches achieves 3.3x lower median latency with 200 workers.

Using fewer workers increases the variability of latency in almost all our queries due to larger amounts of shuffled data per worker. This observation confirms our previous conclusion that shuffling large data among many workers creates stragglers.

Optimization effects. We evaluate the effects of our optimization from Section 4 on TPC-H Q3. We present the numbers in Appendix C. We consider the naive implementation with all optimizations turned off; then, we include simplification rules for location transforms to minimize their number, followed by enabling the block fusion algorithm. Finally, we apply CSE and DCE optimizations to eliminate trigger statements doing redundant network communication during program execution. Our analysis indicates that merging together statements using the block fusion algorithm brings largest performance boosts and enables scalable execution.

7. RELATED WORK

Incremental view maintenance. Classical incremental view maintenance [17, 13, 14] is typically used for processing batch updates in data warehouse systems [6, 37, 33], where the focus is on achieving high throughput rather than low latency. Commercial database systems support incremental view maintenance but only for restricted classes of queries [3, 1]. Our approach is based on using recursive incremental view maintenance [20, 21]. The DBToaster system [22] implements recursive incremental processing for single-tuple updates in local mode. In contrast, we target batched execution in both local and distributed settings. To support efficient batched incremental maintenance of SQL queries, we develop a new technique, called domain extraction.

Scalable stream processing. Scalable stream processing platforms, such as MillWheel [7], Heron [25], and S4 [29], expose low-level primitives to the user for expressing complex query plans. S-Store [10] provides triggers for expressing data-driven processing with ACID guarantees in streaming scenarios. Naiad [27] and Trill [11] support flat LINQ-style continuous queries, while for complex queries with nested aggregates the user has to encode efficient execution plans. Spark Streaming [39] allows running simple SQL queries but only for windowed data. In contrast to these systems, our approach: 1) favors declaritivity as the user only needs to specify input SQL queries without execution plans; 2) can incrementally maintain queries with equality-correlated nested aggregates; 3) generates code tailored to the given workload; our compilation framework can target any scalable system with a synchronous execution model. Percolator [32] handles incremental updates to large datasets but targets latencies on the order of minutes. Scalable batch processing systems like Pig, Hive, and Spark SQL [9] aim for high throughput rather than low latency.

Distributed query processing. Our approach uses well-known techniques from distributed query processing [24, 16, 34], like the basic shuffling primitives and batch pre-processing for minimizing network communication. In contrast to classical distributed query optimization [5, 24, 12], the problem of optimizing recursive incremental programs is more complex since it has to keep track of data-flow dependencies among the program statements that maintain auxiliary views. These dependencies prevent arbitrary statement re-orderings inside trigger functions.

8. CONCLUSION

In this paper, we focus on the problem of low-latency incremental processing of SQL queries in local and distributed streaming environments. We present the domain extraction procedure for maintaining queries with equality-correlated nested aggregates. We study the effect of batch size on the latency of processing in local settings: we show that pre-processing input batches can boost the performance of incremental computation but also demonstrate that tuple-at-a-time processing can outperform batch processing using code specialization techniques (on roughly one half of the benchmarked queries). In local settings, our approach exhibit up to four orders of magnitude better performance than PostgreSQL in incremental batched processing. For distributed view maintenance, we show how to compile local maintenance programs into distributed code optimized for the execution on large-scale processing platforms. Our approach can process tens of million of tuples with few-second latency using hundreds of Spark workers.

9. REFERENCES

[1] Create Indexed Views. http://msdn.microsoft.com/en-us/library/ms191432.aspx.

[2] DBToaster Release 2.2 revision 3387, Nov. 27, 2015. http://www.dbtoaster.org/index.php?page=download.

[3] Materialized View Concepts and Architecture. http://docs.oracle.com/cd/B28359_01/server.111/b28326/repmview.htm.

[4] D. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *CIDR*, pages 277–289, 2005.

[5] S. Adali, K. S. Candan, Y. Papakonstantinou, and V. Subrahmanian. Query Caching and Optimization in Distributed Mediator Systems. In *ACM SIGMOD Record*, volume 25, pages 137–146. ACM, 1996.

[6] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. Efficient View Maintenance at Data Warehouses. In *SIGMOD*, volume 26, pages 417–427, 1997.

[7] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: Fault-tolerant Stream Processing at Internet Scale. *PVLDB*, 6(11):1033–1044, 2013.

[8] A. W. Appel and T. Jim. Continuation-passing, closure-passing style. In *POPL*, pages 293–302, 1989.

[9] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*, pages 1383–1394, 2015.

[10] U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, J. Meehan, A. Pavlo, M. Stonebraker, E. Sutherland, N. Tatbul, K. Tufte, H. Wang, and S. Zdonik. S-Store: A Streaming NewSQL System for Big Velocity Applications. *Proc. VLDB Endow.*, 7(13):1633–1636, 2014.

[11] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A High-performance Incremental Query Processor for Diverse Analytics. *PVLDB*, 8(4):401–412, 2014.

[12] S. Chaudhuri. An overview of query optimization in relational systems. In *PODS*, pages 34–43, 1998.

[13] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *ICDE*, pages 190–200, 1995.

[14] R. Chirkova and J. Yang. Materialized Views. *Databases*, 4(4):295–405, 2011.

[15] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: A Workload-driven Approach to Database Replication and Partitioning. *PVLDB*, 3(1-2):48–57, 2010.

[16] R. Epstein, M. Stonebraker, and E. Wong. Distributed Query Processing in a Relational Data Base System. In *SIGMOD*, pages 169–180, 1978.

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

[18] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. *SIGMOD Rec.*, 22(2):157–166, 1993.

[19] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building Efficient Query Engines in a High-level Language. *PVLDB*, 7(10):853–864, 2014.

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

[21] C. Koch. Incremental query evaluation in a ring of databases, 2013. Technical Report EPFL-REPORT-183766, https://infoscience.epfl.ch/record/183766.

[22] C. Koch, Y. Ahmad, O. Kennedy, M. Nikolic, A. Nötzli, D. Lupei, and A. Shaikhha. DBToaster: Higher-order delta processing for dynamic, frequently fresh views. *VLDBJ*, 23(2):253–278, 2014.

[23] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, et al. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *CIDR*, 2015.

[24] D. Kossmann. The state of the art in distributed query processing. *CSUR*, 32(4):422–469, 2000.

[25] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter Heron: Stream Processing at Scale. In *SIGMOD*, pages 239–250, 2015.

[26] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

[27] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A Timely Dataflow System. In *SOSP*, pages 439–455, 2013.

[28] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9):539–550, 2011.

[29] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed Stream Computing Platform. In *ICDMW*, pages 170–177, 2010.

[30] S. Padmanabhan, T. Malkemus, A. Jhingran, and R. Agarwal. Block Oriented Processing of Relational Database Operations in Modern Computer Architectures. In *ICDE*, pages 567–574, 2001.

[31] A. Pavlo, C. Curino, and S. Zdonik. Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. In *SIGMOD*, pages 61–72, 2012.

[32] D. Peng and F. Dabek. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *OSDI*, pages 1–15, 2010.

[33] D. Quass and J. Widom. On-line Warehouse View Maintenance. *SIGMOD Rec.*, 26(2):393–404, 1997.

[34] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2003.

[35] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *GPCE*, pages 127–136, 2010.

[36] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. *Commun. ACM*, 55(6):121–130, 2012.

[37] J. Wiener, H. Gupta, W. Labio, Y. Zhuge, H. Garcia-Molina, and J. Widom. A System Prototype for Warehouse View Maintenance. 1995.

[38] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *HotCloud*, 2010.

[39] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *SOSP*, pages 423–438, 2013.

	Re-evaluation (PostgreSQL)						IVM (PostgreSQL)						Recursive IVM (DBToaster, C++)						
Batch size	1	10	100	1000	10000	100000	1	10	100	1000	10000	100000	Single	1	10	100	1000	10000	100000
TPC-H 1	10	32	103	187	1003	2889	57	332	2478	9686	12250	12194	1267132	307715	1186926	2812549	4179921	4163385	4372480
TPC-H 2	11	146	427	493	5685	13420	28	125	193	4726	1995	12044	756611	456559	911141	1005175	971154	986810	691260
TPC-H 3	34	111	301	522	2120	5722	17	78	222	61	9100	5676	3736860	1251162	3004637	4401315	5323732	5182684	4580003
TPC-H 4	23	122	400	400	3713	7333	55	93	5857	10223	12550	12971	10076062	1603637	4652895	9012747	10687864	10790913	9752380
TPC-H 5	30	88	253	460	2304	5424	11	174	9	11	153	8665	584261	387568	625805	690475	658595	618632	509490
TPC-H 6	27	88	282	319	2500	6528	58	579	4065	11091	12742	12556	138216710	17017320	44270149	78310773	98176845	116931875	101327791
TPC-H 7	34	117	311	317	2985	7083	7	30	2	16	6908	5944	650650	397643	689015	753051	775288	770259	646018
TPC-H 8	35	114	338	335	2629	5886	28	8	21	19	222	5571	91221	73786	219613	279041	273387	277808	221020
TPC-H 9	30	151	335	513	1952	5999	14	25	22	21	197	1545	104370	78460	90949	109515	102940	86251	76296
TPC-H 10	32	88	322	207	2282	5701	33	165	372	123	7836	12306	2889537	1391237	3605282	5771226	6354245	7050705	5964290
TPC-H 11	22	61	191	366	1930	8734	23	71	220	367	2062	9414	768	776	1923	13500	109603	407547	591716
TPC-H 12	33	109	356	646	3279	4823	49	1050	6248	10837	630	5306	8675929	1678780	3905117	7131341	8236605	7706686	7469474
TPC-H 13	21	74	170	489	345	2578	33	94	283	267	2108	7334	779515	444588	685871	758725	701083	679684	474765
TPC-H 14	26	74	230	313	2322	6556	35	40	56	148	11686	11611	33041606	2769955	13146565	27984674	43468480	51827803	53436252
TPC-H 15	18	74	203	154	1613	4674	N/A	34	38	86	923	3944	17	17	27	52	109	285	964
TPC-H 16	19	58	203	330	1161	2822	18	35	302	1317	5304	10095	123936	115749	131902	121464	108208	75015	58721
TPC-H 17	27	57	88	194	481	666	36	111	564	948	589	1288	379303	210671	256882	208937	279930	155138	131964
TPC-H 18	20	66	207	223	1190	2114	20	18	17	77	676	5881	1133647	572132	1040612	1278945	1272541	1274853	971313
TPC-H 19	31	72	234	329	2218	6139	42	422	502	144	291	11944	1946309	2461229	5829592	8753856	9988084	9737049	8776165
TPC-H 20	6	9	15	18	36	56	11	7	5	21	25	43	977	950	1504	129092	874469	2191422	1871407
TPC-H 21	32	110	347	417	794	7333	31	54	24	10	10583	6797	836800	282532	449838	508128	501657	478923	407540
TPC-H 22	14	45	141	247	1551	5462	12	36	97	298	1252	741	189	183	336	5918	54459	434245	815903
TPC-DS 3	35	92	328	2045	3136	6694	164	1785	4457	5444	6866	8416	12309621	2072493	5945894	8116538	8718851	7740490	6442932
TPC-DS 7	1	14	109	219	1635	5000	98	1104	4262	4438	6721	6694	858649	361726	1024456	1227564	1076421	963048	392581
TPC-DS 19	1	106	132	264	2480	5056	0	78	232	82	7231	6139	30	29	29	29	29	25	7
TPC-DS 27	107	837	3706	5760	7307	8037	76	867	4326	6331	6846	8212	588394	196141	560333	669487	529235	497035	202849
TPC-DS 34	0	4	42	208	1821	4730	1	4	41	173	7510	6350	2803540	1019381	4742812	9219979	10612493	11445739	10689928
TPC-DS 42	9	79	405	196	4322	5594	153	1234	4607	5124	6766	7694	21127917	2589874	7953492	10242024	11325623	9460881	8530989
TPC-DS 43	0	4	42	201	1643	4411	91	710	3232	4702	7246	6972	3103901	589052	3488082	9893883	12851893	14494762	14605017
TPC-DS 46	0	4	43	204	2050	5040	0	1	14	56	6599	6611	185	149	149	149	149	147	90
TPC-DS 52	8	79	406	191	4289	6583	136	869	4266	5066	6808	8121	21005081	2456902	8102885	10552388	11401574	10311449	9074722
TPC-DS 55	31	97	294	875	2444	5444	52	1128	4639	5726	6528	6825	34822881	2343359	7680837	10282103	10993911	10028796	8951066
TPC-DS 68	0	4	42	202	2226	5238	0	1	14	57	6628	6361	182	149	149	149	149	147	88
TPC-DS 73	107	810	3866	5892	7308	8111	139	896	4352	6396	6919	8184	2104283	1028758	4739743	8906107	10620186	10460940	9176666
TPC-DS 79	0	4	39	197	1819	4066	1	4	41	170	5036	6111	838490	427822	1903781	3289301	3869664	3254954	2834858

Table 1: Throughput comparison of re-evaluation and incremental maintenance in PostgreSQL and recursive incremental maintenance in generated C++ code for different batch sizes (in tuples per second).

APPENDIX

A. QUERY LANGUAGE

Here, we revisit the query language from our previous work [22]. The language uses algebraic formulas to define queries (views) over generalized multiset relations. Valid queries result in relations with finite support, and because the tuples in a relation are unique, we can interpret query results as maps (dictionaries) with tuples being the keys and multiplicities being the values.

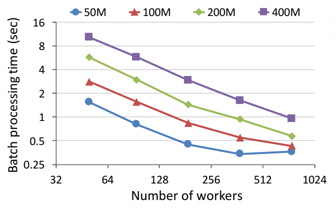
We first present a fragment of the language sufficient for expressing flat queries with aggregates.

- Relation $R(A_1, A_2, ...)$ represents the contents of a base table. It defines a mapping from every unique tuple of the relation to its multiplicity. The SQL equivalent is `SELECT A1, A2, ..., COUNT(*) FROM R GROUP BY A1, A2, ...`, modulo the aggregate value, which is now part of the multiplicity.
- Bag union $Q_1 + Q_2$ merges tuples of Q_1 and Q_2, summing up their multiplicities.
- Natural join $Q_1 \bowtie Q_2$ matches tuples of Q_1 with tuples of Q_2 on their common columns, multiplying the multiplicities.
- $\mathrm{Sum}_{[A_1,A_2,...]}Q$ serves as multiplicity-preserving projection. The SQL equivalent is a group-by `SUM(1)` aggregate, except that the aggregate value is stored inside the multiplicity of the group-by tuples.
- Constant c can be interpreted as a singleton relation mapping the empty tuple to the multiplicity of c. The empty tuple joins with any other tuple.
- Value $f(var1, var2, \dots)$ is an interpreted relation defining a mapping from tuple $\langle var1, var2, \dots \rangle$ to its multiplicity determined by $f(var1, var2, \dots)$. Constant c is a special case of a value term. Value terms are valid only if all variables are bound at evaluation time. For example, expression A has a free (unsafe) variable, thus it is invalid, while expression $R(A,B) \bowtie A$ has finite support. Note that the latter expression corresponds to the SQL query `SELECT A, B, SUM(A) FROM R GROUP BY A, B`.
- Variable assignment $(var := value)$ defines a singleton relation mapping a single-column tuple with the indicated value to a multiplicity of 1. Multiple assignments can be joined together to construct an arbitrary wide tuple.
- Comparison $(value_1 \, \theta \, value_2)$ is an interpreted relation where each tuple has a multiplicity of either 0 or 1 depending on the truthfulness of the boolean predicate. Joining an expression with a comparison filters out tuples that do not satisfy the predicate by setting their multiplicity to 0; the multiplicities of the matching tuples remain unchanged.

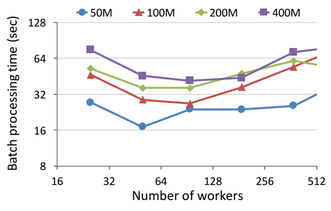
To support queries with nested aggregates, the query language generalizes the assignment operator to take on arbitrary expressions instead of just values. Then variable assignment $(var := Q)$ defines a finite-size relation containing tuples of expression Q with non-zero multiplicities extended by column var holding that multiplicity. Each output tuple has the multiplicity of 1. Expression Q may be correlated with the outside as usual in SQL.

B. SINGLE-NODE EXPERIMENTS

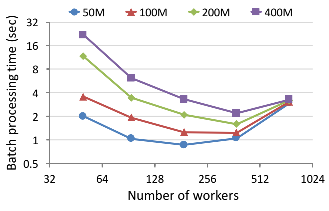
In this section, we provide more single-node experimental results, including a comparison of recursive incremental view maintenance in DBToaster and re-evaluation and classical incremental view maintenance in PostgreSQL.



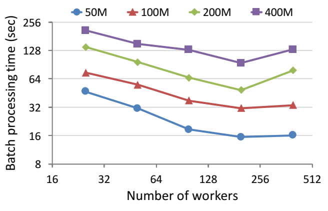
(a) Q1



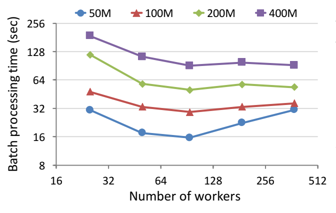
(b) Q2



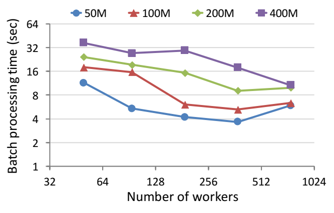
(c) Q4



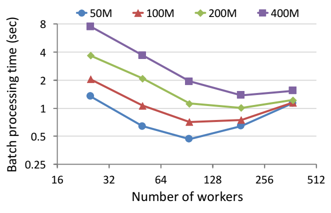
(d) Q8



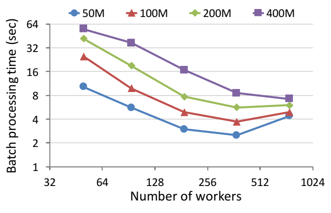
(e) Q10



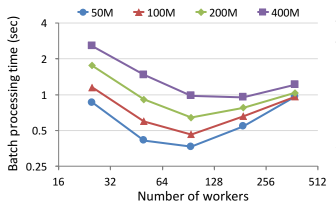
(f) Q11



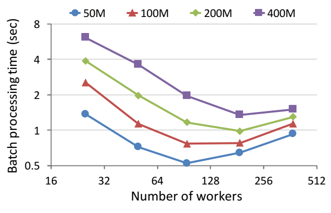
(g) Q12



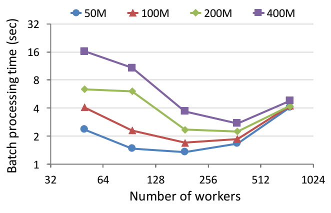
(h) Q13



(i) Q14



(j) Q19



(k) Q22

Figure 11: Strong scalability of the incremental view maintenance of TPC-H queries for different batch sizes (in million of tuples).

Batch size	Single	1	10	100	1000	10000	100000
instructs	19,633	145,670	33,407	17,199	15,750	15,425	15,868
I1 misses	2.0	6.8	3.5	2.0	1.8	1.8	1.4
LLC refs	485	683	533	424	402	578	668
LLC misses	369	562	416	302	258	302	316

Table 2: Cache locality of TPC-H Q3. All numbers are in millions.

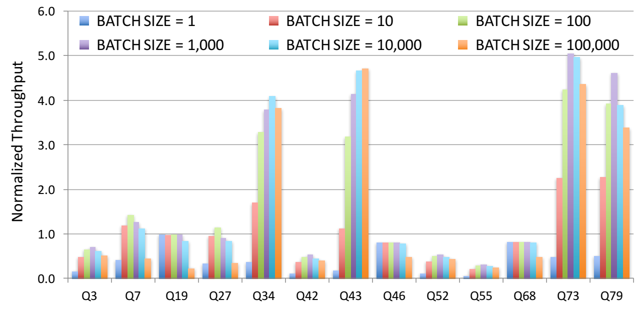


Figure 12: Normalized throughput of TPC-DS queries for different batch sizes with single-tuple execution as the baseline.

B.1 Comparison with PostgreSQL

Table 1 compares the throughput of recursive incremental view maintenance using our generated C++ programs and re-evaluation and classical incremental view maintenance using PostgreSQL for TPC-H and TPC-DS queries. These numbers demonstrate that our view maintenance and code specialization techniques outperform, in all but few cases, the database system by orders of magnitude, even when processing large update batches.

B.2 Cache Locality

In this experiment, we measure the cache locality of generated programs for single-tuple and batched incremental processing on a 10GB input stream. We use perf 3.13.11 to monitor CPU performance counters during the view maintenance of TPC-H Q3. We start profiling after loading the streams and forming input batches. Table 2 presents the obtained results.

The numbers of retired instructions of the single-tuple and batched programs roughly correspond to the normalized throughput numbers from Figure 7. The batch processing with size 1 executes almost 10x more instructions than with size 1,000, which translates into a 4.3x slowdown in running time. The generated programs exhibit low numbers of L1 instruction cache misses compared to the total number of instructions. These results are indicators of good code locality of our view maintenance code.

The second block shows the number of caches references and misses that reached the last level cache. Extremely large and small batch sizes have negative effects on cache locality. Relatively high numbers of cache misses are expected given that in this streaming scenario most input data passes through the query engine clearing out the cache and without being referenced again. Batched processing with size 1,000 exhibits the lowest number of cache references and cache misses, which corresponds to the result from Figure 7.

B.3 TPC-DS Benchmark

Figure 12 shows the normalized throughput of batched incremental processing of a subset of the TPC-DS queries for different batch sizes using the tuple-at-a-time performance as the baseline. These results show that single-tuple processing of TPC-DS queries often outperforms batched processing due to simpler maintenance code. Preprocessing input batches to filter out irrelevant tuples and remove unused columns can bring up to 5x better performance for four TPC-DS queries from our workload.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
Jobs	1	1	1	1	2	1	3	2	3	1	2
Stages	1	3	3	2	5	1	6	6	7	3	4
	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
Jobs	1	2	1	1	3	1	1	1	1	2	2
Stages	2	4	2	3	5	2	3	2	3	4	3

Table 3: View maintenance complexity of TPC-H queries in Spark.

C. DISTRIBUTED EXPERIMENTS

In this section, we present more scalability results of the TPC-H queries, shown in Figure 11, and provide additional information about our distributed incremental view maintenance approach.

C.1 Query Complexity in Spark

Generated Spark code runs a sequence of jobs in order to perform incremental view maintenance. Each job consists of multiple stages (e.g., map-reduce phases), and each stage corresponds to one block of distributed statements. In Table 3, we show the complexity of the TPC-H queries in Spark expressed as the number of jobs and stages necessary to process one batch of updates to base relations, assuming the partitioning strategy described in Section 6.2. The structure of each query determines the number of jobs and stages.

C.2 Optimization Effects

Figure 13 shows the effects of our optimizations from Section 4 on the distributed incremental view maintenance of TPC-H Q3 for input batches with 200 million tuples. We consider the naive implementation with all optimizations turned off; then, we include simplification rules for location transforms to minimize their number, followed by enabling the block fusion algorithm. Finally, we apply CSE and DCE optimizations to eliminate trigger statements doing redundant network communication during program execution.

Our results show that applying simplification rules can reduce the median latency of incremental processing of Q3 by 35% when using 400 workers. Grouping together trigger statements using the block fusion algorithm reduces the number of stages necessary to process one input batch, which enables scalable execution. Eliminating redundant network communication statements further decreases the latency by 11% for 400 workers. The final optimized program relies on the Spark framework to pipeline processing stages, bringing up to 22% performance improvements.

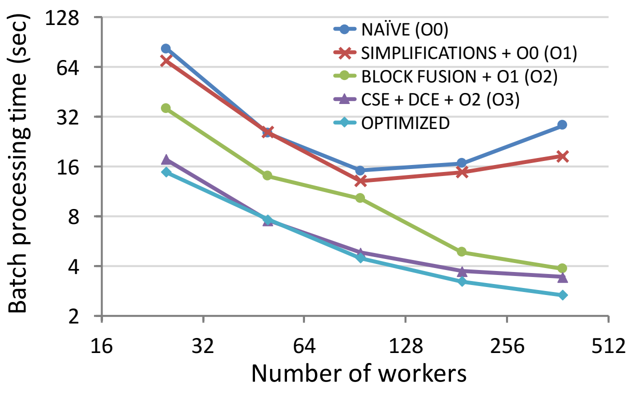


Figure 13: Optimization effects on the distributed incremental view maintenance of TPC-H Q3 for batches with 200 million tuples.

C.3 Block Fusion Algorithm

Here, we present the block fusion algorithm described in Section 4.3.2 for reordering and merging together consecutive statement blocks with the goal of minimizing their number.

```
def commute(s1: Stmt, s2: Stmt): bool =
  !s2.rhsMaps.contains(s1.lhsMap) &&
  !s1.rhsMaps.contains(s2.lhsMap)

def commute(b1: Block, b2: Block): bool =
  b1.stmts.forall(lhs =>
    b2.stmts.forall(rhs => commute(lhs, rhs)))

def mergeIntoHead(hd: Block, tl: List[Block]) =
  tl.foldLeft (hd,Nil) { case ((b1,rhs),b2) =>
    if (b1.mode == b2.mode &&
        rhs.forall(b => commute(b,b2)))
      (Block(b1.mode, b1.stmts++b2.stmts), rhs)
    else (b1, rhs :+ b2) }

def merge(blocks: List[Block]) = blocks match {
    case Nil => Nil
    case hd::tl =>
      val (hd2,tl2) = mergeIntoHead(hd,tl)
      if (hd == hd2) hd::merge(tl)
      else merge(hd2::tl2) }
```