Enabling Incremental Query Re-Optimization

Mengmeng Liu[*]
@WalmartLabs
liumengmeng@gmail.com

Zachary G. Ives
University of Pennsylvania
zives@cis.upenn.edu

Boon Thau Loo
University of Pennsylvania
boonloo@cis.upenn.edu

ABSTRACT

As declarative query processing techniques expand to the Web, data streams, network routers, and cloud platforms, there is an increasing need to re-plan execution in the presence of unanticipated performance changes. New runtime information may affect which query plan we prefer to run. Adaptive techniques require innovation both in terms of the *algorithms used to estimate costs*, and in terms of the *search algorithm* that finds the best plan. We investigate how to build a cost-based optimizer that recomputes the optimal plan *incrementally* given new cost information, much as a stream engine constantly updates its outputs given new data. Our implementation especially shows benefits for stream processing workloads. It lays the foundations upon which a variety of novel adaptive optimization algorithms can be built. We start by leveraging the recently proposed approach of formulating query plan enumeration as a set of *recursive datalog queries*; we develop a variety of novel optimization approaches to ensure effective pruning in both static and incremental cases. We further show that the lessons learned in the declarative implementation can be equally applied to more traditional optimizer implementations.

1. INTRODUCTION

Today, query optimization is being applied to many settings beyond traditional databases. Consider declarative cloud data processing systems [5, 22] and data stream processing [7, 23] platforms, where data properties and the status of cluster compute nodes may be constantly changing. Here it is very difficult to effectively choose a good plan for query execution: data statistics may be unavailable or highly variable; cost parameters may change due to resource contention or machine failures; and in fact a *combination* of query plans might perform better than any single plan. Given the complexities of cost estimation, work in the early 2000s was done on *self-tuning* so the performance of a query or set of queries can be improved [1, 6, 21]. More recently, the focus has been on generating "robust" rather than optimal plans [12] or selecting database designs that yield robust plans [24].

Nonetheless, there is need for *adaptive* techniques that can adjust to runtime conditions during query processing — *adaptive query processing* [11]. Adaptivity requires three orthogonal building blocks: (1) techniques for acquiring information about performance (ideally, performing accurate cost estimation) and trading off *exploration vs. exploitation* as we seek to determine the most efficient plan, (2) a planner capable of taking new information and rapidly re-estimating the most efficient plan, (3) an execution system whose computation can be adapted in mid-execution. Most recent work focused on the first and last problems: namely, new methods for better statistics collection and costs estimation during query execution [4, 6, 17, 18, 21, 30]; and techniques for adapting execution [10]. Our goal is to explore the second capability: how to support *incremental* techniques for re-optimization, whereby an optimizer, given new cost information, would only re-explore the minimal set of query plans whose costs may be affected. Given any existing set of cost estimation techniques, pruning heuristics, and transformation rules, matched with an execution engine, our approach will always incrementally recompute the *same optimal plan* (based on the cost model) as a full-blown optimization.

In this paper, we consider incremental re-optimization for two main settings: (1) *data stream management systems* where data may be bursty, and its distributions may vary over time — meaning that different query plans may be preferred over different segments. Here it is vital to optimize frequently based on recent data distribution and cost information; (2) traditional OLAP database settings when the same query (or highly similar queries) gets executed frequently, as in a prepared statement. Here we may wish to re-optimize the plan after each iteration, given increasingly accurate information about costs, and we would like this optimization to have minimal overhead. Our focus in the paper is not on novel optimizer strategies or cost estimation techniques per se, but on redesigning the planning components of the optimizer to incrementally recompute the best plan given *any preferred strategy that would be used in a standard optimizer*.

The main contribution of this paper is to show how an *incremental* re-optimizer can be developed, and how it can be useful in adaptive query processing scenarios matching the application domains cited above. Our incremental re-optimizer implements the basic capabilities of a modern database query optimizer, and could easily be extended to support other more advanced features and estimators; our main goal is to show that an incremental optimizer following our model can be competitive with a standard optimizer implementation for *initial* optimization, and significantly faster for *repeated* optimization — for any choice of cost model and estimation techniques. Moreover, in contrast to randomized or heuristics-based optimization methods, we **still guarantee the discovery of the best plan** according to the cost model. Since our work is oriented towards adaptive query processing, we evaluate the system in a variety of settings in conjunction with a basic pipelined query engine for stream and stored data.

We implement incremental re-optimization using a novel approa-

[*]This work was done while the author was at UPenn.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD '16, June 26–July 1, 2016, San Francisco, CA, USA.
© 2016 ACM. ISBN 978-1-4503-3531-7/16/06…$15.00
DOI: http://dx.doi.org/10.1145/2882903.2915212

ch, which is based on the observation that query optimization is essentially a recursive process involving the derivation and subsequent pruning of state (namely, alternative plans and their costs). If one is to build an *incremental* re-optimizer, this requires preservation of state (i.e., the optimizer memoization table) across optimization runs — but moreover, it must be possible to determine what plans have been *pruned* from this state, and to re-derive such alternatives and test whether they are now viable.

One way to achieve such "re-pruning" capabilities is to carefully define a semantics for how state needs to be tracked and recomputed in an optimizer. However, we observe that this task of "re-pruning" in response to updated information looks remarkably similar to the database problem of *view maintenance* through aggregation [15] and recursion as studied in the database literature [16]. In fact, recent work [9] has shown that query optimization can itself be captured in recursive datalog. Thus, we initially formulate our approach to developing an incremental re-optimizer using a *declarative specification*. **A key benefit of the declarative approach is that it enables us to identify state-pruning strategies agnostic to the order of control and data flow during plan enumeration.**

More precisely, we express the optimizer as a recursive datalog program consisting of a set of rules, and leverage the existing database query processor to actually execute the declarative program. In essence, this is optimizing a query optimizer using a query processor. Our implementation approaches the performance of conventional procedural optimizers for reasonably-sized queries. Our implementation recovers the initial overhead during subsequent re-optimizations by leveraging *incremental view maintenance* [16, 20] techniques. It only recomputes portions of the search space and cost estimates that might be affected by the cost updates. Frequently, this is only a small portion of the overall search space, and hence we often see order-of-magnitude performance benefits.

Given this setting, we tackle the problem of pruning while recursively computing the optimal plan. We develop a variety of novel *incremental* and *recursive* optimization techniques to capture the kinds of pruning used in a conventional optimizer, and more importantly, to generalize them to the incremental case. Our approach achieves pruning levels that rival or best bottom-up (as in System-R [27]) and top-down (as in Volcano [13, 14]) plan enumerations with branch-and-bound pruning. Our techniques are of broader interest to incremental evaluation of recursive queries as well.

Finally, we use insights gained from our declarative implementation to form a *specification for pruning and state management* that we use to retrofit standard optimizer architectures with incremental re-optimization capabilities. Here we still *guarantee optimality* of the pruning techniques based on their equivalence to the declarative implementation. We make the following contributions:

- A rule-based, declarative approach to query (re)optimization as an *incremental view maintenance problem*, which relaxes traditional restrictions on search order and pruning.
- Novel strategies to prune the state of an executing recursive query, such as a declarative optimizer: *aggregate selection* with *tuple source suppression*; *reference counting*; and *recursive bounding*.
- An implementation over a query engine developed for recursive stream processing [20], with a comprehensive evaluation of performance against alternative approaches, over a diverse workload.
- "Porting" techniques from incremental re-optimization to conventional cost-based adaptive query processing techniques [17, 30].

2. INCREMENTAL RE-OPTIMIZATION

We show our query processing architecture in Figure 1. We assume a set of **transformation rules** and **initial statistics**, using any existing strategy for statistics collection and cost estimation. Our specific implementation uses histograms on join attributes and

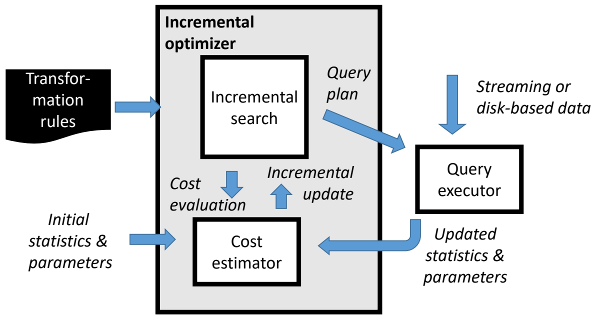


Figure 1: Query processing architecture

standard heuristics [8]. The implemented transformation space includes hash and nested loops join and group-by, as well as a sort *enforcer* and merge joins. Our focus is on the **incremental search** module: we develop strategies that — given updated runtime conditions from the **query executor** — can trigger an *incremental update* to the search, which still *guarantees the choice of the lowest-estimated-cost plan in the search space*, i.e., that exactly mirror what a non-incremental optimizer would have done. The **query executor** in our system [20] supports streaming, favors pipelined execution, and supports runtime monitoring of statistics.

In subsequent sections, we first consider a *declarative* implementation of the incremental re-optimizer, which enables us to decouple search and pruning from any particular order of search. Next we show how to take the same strategies and incorporate them into a traditional, procedural query optimizer implementation.

3. DECLARATIVE QUERY OPTIMIZATION

Our goal is an optimizer that incrementally maintains information about best (sub)plans and takes advantage of pruning opportunities. Restated, it performs *incremental state management* of (recursively computed) plan costs, in response to updates to query plan cost information. Incremental update propagation is a very well-studied problem for recursive queries, with a clean semantics and many efficient implementations. Thus we consider a model in which our optimizer is itself formulated as a recursive, incrementally maintainable query. (In Section 5.4 we show how to adapt ideas to a more traditional optimizer setting.)

Prior work has also demonstrated the feasibility of a datalog-based query optimizer [9]. Hence, rather than re-inventing incremental recomputation techniques we have built our optimizer as a series of recursive rules in datalog, executed in the query engine that already exists in the DBMS. In contrast to the prior work, our focus is not on formulating every aspect of query optimization in datalog, but rather on capturing the state management and pruning as datalog rules — so we can adapt incremental view maintenance (delta rules) and develop novel sideways information passing techniques, respectively. Other optimizer features that are not reliant on state that changes at runtime, such as cardinality estimation, breaking expressions into subexpressions, etc., are specified as built-in auxiliary functions. As a result, we specify an entire optimizer in only three stages and 10 rules (dataflow is illustrated in Figure 2).

Plan enumeration (SearchSpace). Searching the space of possible plans has two aspects. In the *logical phase*, the original query is recursively broken down into a full set of alternative relational algebra subexpressions[1]. The decomposition is naturally a "top-down" type of recursion: it starts from the original query expression, which then breaks down into subexpressions, and so on. The *physical phase* takes as input a query expression from the logical phase, and creates physical plans by enumerating the set of possible physical operators that satisfy any constraints on the output

[1]Alternatively, only *left-linear* expressions may be considered [27].

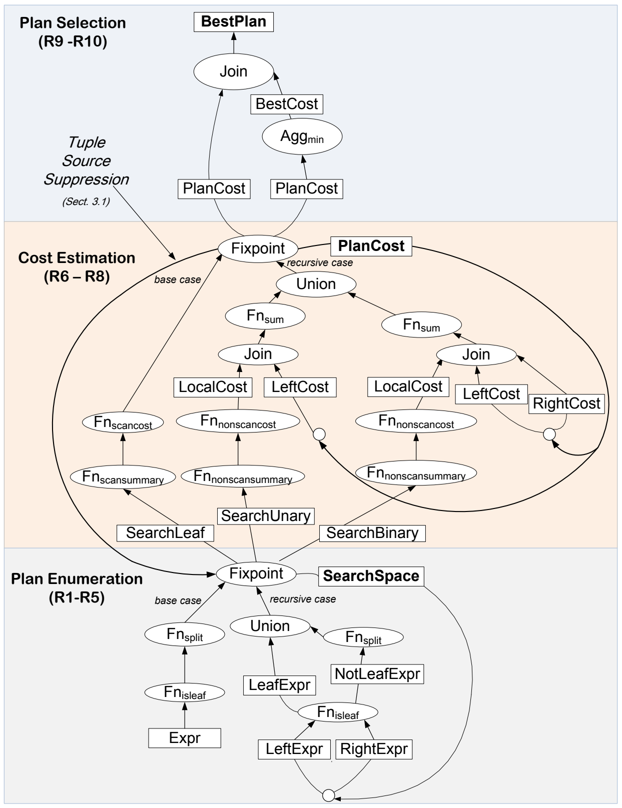


Figure 2: Query plan of our declarative query optimizer. Operators are in ellipses; views are in rectangles. Plan enumeration (SearchSpace) **consists of 5 rules, cost estimation** (PlanCost) **3 rules, and plan selection** (BestPlan) **2 rules. See Appendix.**

properties [14] or "interesting orders" [27] (e.g., the data must be sorted by a particular attribute). Without physical properties, the extension from logical plans to physical plans can be computed either top-down or bottom-up; however, the properties are more efficiently computed in goal-directed (top-down) manner.

Cost estimation (PlanCost)**.** This phase determines the cost for each physical plan in the search space, by recursively merging the statistics and cost estimates of a plan's subplans. It is naturally a bottom-up type of recursion, as the plan subexpressions must already have been cost-estimated before the plan itself. Here we can encode in a table the mapping from a plan to its cost.

Plan selection (BestPlan)**.** As costs are estimated, the program produces the plan that incurs the lowest estimated cost.

In our declarative approach to query optimization, we treat optimizer state as data, use rules to specify what a query optimizer is, and leverage a database query processor to actually perform the computation. Figure 2 shows a (simplified) query plan for the datalog rules. As we can see, the declarative program is by nature recursive, and is broken into the three stages mentioned before (with Fixpoint operators between stages). Starting from the bottom of the figure, **plan enumeration** recursively generates a SearchSpace table containing plan specifications, by decomposing the query and enumerating possible output properties; enumerated plans are then fed into the **plan estimation** component, PlanCost, which computes a cost for each plan, by building from leaf to complex expressions; **plan selection** computes a BestCost and BestPlan entry for each query expression and output property, by aggregating across the PlanCost entries.

This basic datalog program can be evaluated in a conventional datalog engine, augmented with a few user-defined functions. How-

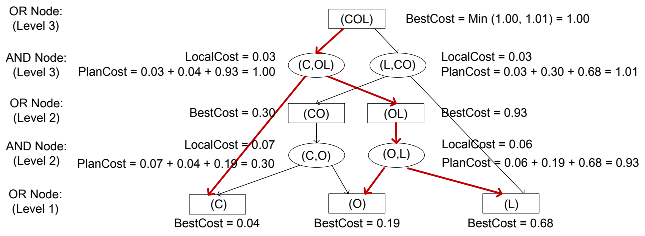


Figure 3: The and-or-graph for Q3S. Red edges denote the best plan. Rectangles and ovals denote "OR" and "AND" nodes respectively. Each "OR" node is labeled with its BestCost **and each "AND" node is labeled with its** LocalCost **and** PlanCost**.**

ever, it will likely perform poorly due to limited pruning of plan alternatives. A major contribution of this paper is to develop new strategies for pruning of recursive computations like planning. We use an example to help explain the overall evaluation process.

EXAMPLE 1. *As our driving example, consider a simplified TPC-H Query 3 with its aggregates and functions removed, called Q3S.*

```
SELECT L_orderkey, O_orderdate, O_shippriority
FROM Customer C, Orders O, Lineitem L
WHERE C_mktsegment = 'MACHINERY' and C_custkey =
    O_custkey and O_orderkey = L_orderkey and
    O_orderdate < '1995-03-15' and L_shipdate > '
    1995-03-15'
```

3.1 Plan Enumeration

Plan enumeration takes as input the original query expression as Expr, and then generates as output the set of alternative plans. As with many optimizers, it is divided into two levels:

Logical search space. The logical plan search space contains all the logical plans that correspond to subexpressions of the original query expression up to any logically equivalent transformations (e.g., commutativity and associativity of join operators). In traditional query optimizers such as Volcano [14], a data structure called an *and-or-graph* is maintained to represent the logical plan search space. The and-or-graph can capture both tree- and DAG-structured query plans [26]. Bottom-up dynamic programming optimizers do not physically store this graph, but it is still conceptually relevant.

EXAMPLE 2. *Figure 3 shows an example and-or-graph for Q3S, which describes a set of alternative subplans and subplan choices using interleaved levels. "AND" nodes represent alternative subplans (typically with join operator roots) and the "OR" nodes represent decision points where the cheapest AND-node was chosen.*

Our implementation captures each node in a table SearchSpace. As we discuss next, we supplement this information with further information about the output properties and *physical plan*. (We explain why we combine the results from both stages in Section 3.3.)

Physical search space. The physical search space extends the logical one in that it enumerates all the physical operators for each algebraic logical operator. For example, in our figure above, each "AND" node denotes a logical join operator, but it may have multiple physical implementations such as pipelined-hash, indexed nested-loops, or sort-merge join. If the physical implementation is not symmetric, exchanging the left and right child would become a different physical plan. A physical plan has not only a root physical operator, but also a set of physical properties over the data that it maintains or produces; if we desire a set of output properties, this may constrain the physical properties of the plan's inputs.

EXAMPLE 3. *Table 1 shows the* SearchSpace *content for a subset of Figure 3. The AND logical operators are either joins (with*

*Expr	*Prop	*Index	LogOp	*PhyOp	lExpr	lProp	rExpr	rProp
(COL)	–	1	join	sort-merge	(C)	C_custkey order	(OL)	O_custkey order
(COL)	–	2	join	indexed nested-loop	(L)	index on L_orderkey	(CO)	–
(OL)	O_custkey order	1	join	pipelined-hash	(O)	–	(L)	–
(CO)	–	1	join	pipelined-hash	(C)	–	(O)	–
(CO)	–	1	join	sort-merge	(C)	–	(O)	–
(O)	O_custkey order	–	scan	local scan	–	–	–	–
(L)	index on L_orderkey	–	scan	index scan	–	–	–	–
(C)	C_custkey order	–	scan	local scan	–	–	–	–

Table 1: A simplified SearchSpace relation encoding the and-or-graph for Q3S's search space. Primary keys are denoted by *.

2 child expressions), or tablescans (with selection predicates applied). Each expression Expr *may have multiple Indexed alternatives. Prop and PhyOp represent the physical properties of a plan and its root physical operator, respectively.*

For instance, expression SearchSpace(COL) *has encodes an "OR" node with two alternatives the first "AND" (join) child is* SearchSpace(C, OL) *and the second is* SearchSpace(L, CO)*. For the first* SearchSpace *tuple, the left expression is C and the right expression is OL. The tuple indicates a Sort-Merge join, whose left and right inputs' physical properties require a sort order based on $C\_custkey$ and $O\_custkey$, respectively. The second alternative uses an Indexed Nested-Loop Join as its physical operator. The left expression refers to the inner join relation indexed on $L\_orderkey$, while there are no ordering restrictions on the right expression.*

We enumerate plans using a single recursive query (bottom of Figure 2). Given an expression, the $\mathsf{Fn}_{\mathsf{split}}$ function enumerates all the algebraically equivalent rewritings for the given expression, as well as the possible physical operators and their interesting orders. Fixpoint is reached when Expr has been decomposed down to leaf-level scan operations over a base relation (checked using $\mathsf{Fn}_{\mathsf{isleaf}}$).

3.2 Cost Estimation and Plan Selection

The cost estimation component computes an estimated cost for every physical plan. Given the SearchSpace tuples generated in the plan enumeration phase, three datalog rules (R6 - R8) are used to compute PlanCost (corresponding to a more detailed version of the "AND" nodes in Figure 3, with physical operators considered and all costs enumerated), and two additional rules (R9 - R10) select the BestPlan (corresponding to an "OR" node). Cost estimates are recursively computed by summing up the children costs and operation costs. The computed sum for each physical plan is stored in PlanCost.

In addition to the search space, cost estimation requires a set of *summaries* (statistics) on the input relations and indexes, e.g., cardinality of a (indexed) relation, selectivity of operators, data distribution, etc. These summaries are computed using functions $\mathsf{Fn}_{\mathsf{scansummary}}$ and $\mathsf{Fn}_{\mathsf{nonscansummary}}$. The former computes the leaf level summaries over base tables, and the latter computes the output summaries of an operator based on input summaries. Given the statistics, the cost of a plan can be computed by combining factors such as CPU, I/O, bandwidth and energy into a single cost metric. We compute the cost of each physical operator using functions $\mathsf{Fn}_{\mathsf{scancost}}$ and $\mathsf{Fn}_{\mathsf{nonscancost}}$ respectively.

Given the above functions, cost estimation becomes a recursive computation that sums up the cost of children expressions and the root cost, to finally compute a cost for the entire query plan. At each step, $\mathsf{Fn}_{\mathsf{sum}}$ is used to sum up the PlanCost of its child plans with LocalCost. The particular form of the operation depends on whether the plan root is a leaf node, a unary or a binary operator.

EXAMPLE 4. *To illustrate the process of cost estimation, we revisit Figure 3, which shows a simplified logical search space (omitting physical operators and properties) for our simplified TPC-H Q3S. For every "AND" node, we compute the plan cost by summing up the cost of the join operator, with the best costs of computing its two inputs (e.g., the level 2 "AND" node (C, O) sums up its local cost 0.07, its left best cost 0.04, and its right best cost 0.19, and gets its plan cost 0.30). For every "OR" node, we determine the alternative with minimum cost among its "AND" node children (e.g., the level 3 "OR" node (COL) computes a minimum over its two input plan costs 1.00 and 1.01, and gets its best cost 1.00). After the best cost is computed for the root "OR" node in the graph, the optimization process is done, and an optimal plan tree is chosen.*

Once the PlanCost for every "AND" node are generated, the final two rules compute the BestCost for every "OR" node by computing a min aggregate over PlanCost of its alternative "AND" node derivations, and output the BestPlan for each "AND" node by computing a join between BestCost and PlanCost.

3.3 Execution Strategy

Given a query optimizer specified in datalog, the natural question is how to actually execute it. We seek to be general enough to incorporate typical features of existing query optimizers, to rival their performance and flexibility, and to only develop implementation techniques that generalize. We adopt two strategies and develop novel execution techniques to support them:

Merging logical and physical plan enumeration. The physical plan elaborates on the logical plan. Since both logical and physical enumeration are top-down types of recursion, and there is no pruning information from the physical stage that should be propagated back to the logical one, we can merge the logical and physical enumeration stages into a single recursive query.

As we enumerate each logical subexpression, we simultaneously join with the table representing the possible physical operators that can implement it. This generates the entire set of possible physical query plans. To make it more efficient to generate multiple physical plans from a single logical expression, we use *caching* to memoize the results of $\mathsf{Fn}_{\mathsf{nonscansummary}}$ and $\mathsf{Fn}_{\mathsf{split}}$.

Decoupling of cost estimation and plan enumeration. Cost estimation requires bottom-up evaluation: a cost estimate can *only* be obtained once cost estimates and statistics are obtained from child expressions. The enumeration stage naturally produces expressions in the order from parent to child, yet estimation must be done from child to parent. We *decouple* the execution order among plan enumeration and cost estimation, making the connections between these two components flexible. For example, some cost estimates may happen before all related plans have been enumerated. Cost estimates may even be used to *prune* portions of the plan enumeration space (and hence also further prune cost estimation itself) in an opportunistic way.

4. ACHIEVING PRUNING

We now consider how to incorporate *pruning* of the search space into pipelined execution of our query optimizer. Our approach is based on the idea of *sideways information passing*, in which the computation of one portion of the query plan may be made more efficient by filtering against information computed elsewhere, but not connected directly by pipelined dataflow. Specifically, we incorporate the technique of aggregate selection [28] from the deductive database literature, which we briefly review; we extend it to perform further pruning; and we develop two new techniques for recursive queries that enable tracking of dependencies and computation of bounds. Beyond query optimization, our techniques are broadly useful in the evaluation of recursive datalog queries. In the next section we make these strategies *incrementally maintainable*.

Section 4.1 reviews aggregate selection, which removes non-viable plans from the optimizer state if they are not cost-effective, and shows how we can use it to achieve the similar effects to dynamic programming in bottom-up style optimizers. There we also introduce a novel technique called *tuple source suppression*. In the remainder of the section we show how to introduce two familiar notions into datalog execution: namely, *reference counting* that enables us to remove plan subexpressions once all of their parent expressions have been pruned (Section 4.2), and *recursive bounding*, which lets the datalog engine incorporate branch-and-bound pruning as in a typical top-down query optimizer (Section 4.3). Our approaches relax the traditional restrictions on the search order and pruning techniques in either Volcano's [14] top-down traversal or System R's [27] bottom-up dynamic programming approaches. For example, a top-down search algorithm can have a depth-first order, breadth-first order or another order.

4.1 Pruning Suboptimal Plan Expressions

Dating back to System-R [27], every modern query optimizer uses dynamic programming techniques (although some via memoization [14]). Dynamic programming is based on the *principle of optimality*, i.e. an optimal plan can be decomposed into sub-plans that must themselves be optimal solutions. This property is vital to optimizer performance, because the same subexpression may appear repeatedly in many parent expressions. Formally:

PROPOSITION 5. *Given a query expression E and property p, consider a plan tree $T\langle E, p\rangle$ that evaluates E with output property p. For this and other propositions, we assume that plans have distinct costs. Here one such T will have the minimum cost: call that T_{OPT}. Suppose E^s is a subexpression of E, and consider a plan tree $T^s\langle E^s, p^s\rangle$ that evaluates E^s with output property p^s. Again one such T^s will have the minimum cost: call that T^s_{OPT}. If T^s is* not *T^s_{OPT}, then T^s must* not *be the subtree of T_{OPT}.*

This proposition ensures that we can safely discard suboptimal subplans without affecting the final optimal plan. Consider the and-or-graph of example query Q3S (Figure 3). The red (bolded) subtree is the optimal plan for the root expression (COL). The subplan of the level 3 "AND" node (L, CO) has suboptimal cost 1.00. If there exists a super-expression containing (COL), then the only viable subplan is the one marked in the figure. State for any alternative subplan for (COL) may be pruned from SearchSpace and PlanCost. We prune both relations via two novel techniques.

Pruning PlanCost **via aggregate selection.** Refer back to Figure 2: each BestCost tuple encodes the minimum cost for a given query expression-property pair, over all the plans associated with this pair in PlanCost. To avoid enumerating unnecessary PlanCost tuples, one can wait until the BestCost of subplans are obtained before computing a PlanCost for a root plan. This is how System R-style dynamic programming works. However, this approach constrains the order of evaluation.

We instead extend a logic programming optimization technique called *aggregate selection* [28], to achieve dynamic programming-like benefits for any arbitrary order of implementation. In aggregate selection, we "push down" a selection predicate into the input of an aggregate, such that we can prune results that exceed the current minimum value or are below the current maximum value. In our case (as shown in the middle box of Figure 2), the current best-known cost for any equivalent query expression-property pair is maintained within our Fixpoint operator (which also performs the non-blocked **min** aggregation). We only propagate a newly generated PlanCost tuple if its cost is smaller than the current minimum. This does not affect the computation of BestCost, which still outputs the minimum cost for each expression-property pair. Since pruning bounds are updated upon every newly generated tuple, there is no restriction on evaluation order. As with pruning strategies used in Volcano-style optimizers, the amount of state pruned varies depending on the order of exploration: the sooner a min-cost plan is encountered, the more effective the pruning is.

Pruning SearchSpace **via tuple source suppression.** Enumeration of the search space will generally happen in parallel with enumeration of plans. Thus, as we prune tuples from PlanCost, we may be able to remove related tuples (e.g., representing subexpressions) from SearchSpace, possibly preventing enumeration of their subexpressions and/or costs. We achieve such pruning through *tuple source suppression*, along the arcs indicated in Figure 2. Any PlanCost tuples pruned by aggregate selection should also trigger cascading deletions to the *source tuples* by which they were derived from the SearchSpace relation. To achieve this, since PlanCost contains a superset of the attributes in SearchSpace, we simply project out the cost field and propagate a deletion to the corresponding SearchSpace tuple.

4.2 Pruning Unused Plan Subexpressions

The techniques described in the previous section remove *suboptimal plans* for specific expression-property pairs. However, some *optimal plans* for certain expressions may be unused in the final query execution plan. Consider in Figure 3 the level 2 "AND" node (C, O): this node is not in the final plan because its "OR" node parent expression (CO) does not appear in the final result. In turn, this is because (CO)'s parent "AND" nodes (in this example, just a single plan (L, CO)) do not contribute to the final plan. Intuitively, we may prune an "AND" node if all of its parent "AND" nodes (through only one connecting "OR" node) have been pruned.

We would like to remove such plans once they are discovered, which requires a form of *reference counting* within the datalog engine executing the optimizer. Every tuple in SearchSpace is annotated with a *count*: this count represents the number of **parent plans** still present in the SearchSpace. For example, in Table 1, the plan entry of $O \bowtie L$ has reference count of 1, because it only has one parent plan, which is $C \bowtie OL$; on the other hand, the plan entry of (O) has reference count of 2, because it has two parent plans, which are $O \bowtie L$ and $C \bowtie O$.

PROPOSITION 6. *Given a query expression E with output property p: let T^s be a plan tree for E's subexpression E^s with property p^s. If T^s has reference count of zero, then T^s must not be a subtree of the optimal plan tree for the query E with property p.*

The proposition ensures that a plan with a reference count of zero can be safely deleted. Note that a deleted plan may make more reference counts to drop to zero, hence the deletion process may be recursive. Our reference counting scheme is more efficient than the *counting* algorithm of [16], which uses a count representing the *total number of derivations* of each tuple in bag semantics. Our count represents the number of *unique parent plans from which a subplan may be derived*, and can typically be incrementally updated in a single recursive step (whereas **counting** often requires multiple recursive steps to compute the whole derivation count).

Our reference counting mechanism complements the pruning techniques discussed in Section 4.1. Following an insertion (exploration) or deletion (pruning) of a SearchSpace tuple, we update the reference counts of relevant tuples accordingly; cascading insertions or deletions of SearchSpace (and further PlanCost) tuples may be triggered because their reference counts may be raised

r1: ParentBound($lExpr, lProp, bound - rCost - localCost$) :-
Bound($expr, prop, bound$), BestCost($rExpr, rProp, rCost$),
LocalCost($expr, prop, index, lExpr, lProp, rExpr,$
$rProp, -, localCost$);
r2: ParentBound($rExpr, rProp, bound - lCost - localCost$) :-
Bound($expr, prop, bound$), BestCost($Expr, lProp, lCost$),
LocalCost($expr, prop, index, lExpr, lProp, rExpr,$
$rProp, -, localCost$);
r3: MaxBound($expr, prop, max < bound >$) :-
ParentBound($expr, prop, bound$);
r4: Bound($expr, prop, min < minCost, maxBound >$) :-
BestCost($expr, prop, minCost$),
MaxBound($expr, prop, maxBound$);

Figure 4: Datalog rules to express bounds computation

above zero (or dropped to zero). Finally, the optimal plan computed by the query optimizer is unchanged, but more tuples in SearchSpace and PlanCost are pruned. Indeed, by the end of the process, the combination of aggregate selection and reference counts ensure SearchSpace and PlanCost *only* contain those plans that are on the final optimal plan tree. Such "garbage collection" greatly reduces the optimizer's state and the number of data items that must be updated incrementally, as described in Section 5.

4.3 Full Branch-and-Bound Pruning

Our third innovation generalizes *branch-and-bound pruning*, as in top-down optimizers like Volcano, during cost estimation of physical plans. Branch-and-bound pruning uses *prior exploration* of related plans to prune the exploration of new plans: a physical plan for a subexpression is pruned if its cost already exceeds the cost of a plan for the equivalent subexpression (or its parent, grandparent, or other ancestor expression). Typically, branch-and-bound pruning assumes a single-recursive descent execution thread during its enumeration. We develop a branch-and-bounding solution independent of the search order, which supports parallel enumeration.

Previous work [9] has shown that it is possible to do a limited form of branch-and-bound pruning in a declarative optimizer, by initializing a bound based on the cost of the parent expression, and then pruning subplan exploration whenever the cost has exceeded an *equivalent* expression. This can actually be achieved by our aggregate selection approach described in Section 4.1.

We seek to generalize this to prune against the *best* known bound for an expression-property pair — which may be from a plan for an equivalent expression, or from any ancestor plan that *contains* the subplan corresponding to this expression-property pair. (Recall that there may be several parent plans for a subplan: this introduces some complexity as each parent plan may have different cost bounds, and at certain point in time we may not know the costs for some of the parent plans.) The bound should be continuously *updated* as different parts of the search space are explored via pipelined execution. In this section, we assume that bounds are initialized to infinity and *monotonically decreasing*. In Section 5.3 we will relax this requirement.

Our solution, *recursive bounding*, creates and updates a single recursive relation Bound, whose values form the *best-known* bound on each expression-property pair (each "OR" node). This bound is the minimum of (1) known costs of any equivalent plans; (2) the highest bound of any parent plan's expression-property pair, which in turn is defined recursively in terms of parents of this parent plan. Figure 4 shows how we can express the bounds table using recursive datalog rules. ParentBound propagates cost bounds from a parent expression-property pair to child expression-property pairs, through LocalCost, while the child bound also takes into account the cost of the local operator, and the best cost from the sibling side. MaxBound finds the highest of bounds from parent plans, and Bound maintains the minimum bounding information derived from BestCost or MaxBound, allowing for more strict pruning.

Given the definition of Bound, we can reason about the viability of certain physical plans below:

PROPOSITION 7. *Given a query expression E with desired output property p: let T^s be a plan tree that produces E's subexpression E^s and yields property p^s. If T^s has a cumulative cost that is larger than Bound $\langle E^s, p^s \rangle$, then T^s cannot be a sub-tree of the optimal plan tree for the query E, for property p.*

Based on Proposition 7, recursive bounding may safely remove any plan that exceeds the bound for its expression-property pair. Indeed, with our definition of the bounds, this strategy is a generalization of the aggregate selection strategy discussed in Section 4.1. However, bounds are recursively defined here and a single plan cost update may result in a number of changes to bounds for others.

Overall the execution flow of pruning PlanCost and SearchSpace via recursive bounding is similar to that described in Section 4.1. Specifically, PlanCost is pruned inside the Fixpoint operator, where an additional comparison check PlanCost $<$ Bound is performed before propagating a newly generated PlanCost. Updates over other Bound tuples derived from a given PlanCost tuple are computed separately. SearchSpace is again pruned via sideways information passing where the pruned PlanCost tuples are directly mapped to deletions over SearchSpace.

5. INCREMENTAL RE-OPTIMIZATION

The previous section described how we achieve pruning at a level comparable to a conventional query optimizer, without being constrained to the standard data and control flow of a top-down or bottom-up procedural implementation. In this section, we discuss *incremental* maintenance during both query optimization and re-optimization. In particular, we seek to incrementally update not only the state of the optimizer, but also the state that affects pruning decisions, e.g., reference counts and bounds.

Initial query optimization takes a query expression and metadata like summaries, and produces a set of tables encoding the plan search space and cost estimates. During execution, pruning bounds will always be monotonically decreasing. Now consider *incremental* re-optimization, where the optimizer is given updated cost (or cardinality) estimates based on information collected at runtime after partial execution. This scenario commonly occurs in adaptive query processing, where we monitor execution and periodically re-optimize based on the updated status. For simplicity, our discussion of the approaches assumes that a single *cost parameter* (operator estimated cost, output cardinality) changes, though our architecture is able to handle multiple such changes simultaneously.

Given a change to a cost parameter, our goal is in principle to re-evaluate the costs for all affected query plans. Some of these plans might have previously been pruned from the search space, meaning they will need to be re-enumerated. Some of the pruning bounds might need to be adjusted, as some plans become more expensive and others become cheaper. As the bounds are changed, we may in turn need to re-introduce further plans that had been previously pruned, or to remove plans that had previously been viable. This is where our declarative query optimizer formulation is extremely helpful: we use *incremental view maintenance* techniques to only recompute the necessary results, while guaranteeing correctness.

Incremental maintenance enabled via datalog. From the declarative point of view, initial query optimization and query re-optimization can be considered the same task, if the datalog program is **extended** to handle updates (insertions, deletions and replacements). Indeed, incremental query re-optimization can be specified using a delta rules formulation like [16]. This requires extensions to the database query processor to support **direct processing of deltas**: instead of processing standard tuples, each operator in the query processor must be extended to process delta tuples encoding changes.

A delta tuple of a relation R may be an insertion (R[+x]), deletion (R[-x]), or update (R[x→x′]). For example, a new plan generated in SearchSpace is an insertion; a pruned plan in PlanCost is a deletion; an updated cost of BestCost is an update.

We extend the query processor following standard conventions from continuous query systems [19] and stream management systems [23]. The extended query operators consume and emit deltas largely as if they were standard tuples. For stateful operators, we maintain for each encountered tuple value a *count*, representing the cumulative total of how many times the tuple has been inserted and deleted. Insertions increment the count and deletions decrement it; counts may temporarily become negative if a deletion is processed out of order with its corresponding insertion, though ultimately the counts converge to nonnegative values, since every deletion is linked to an insertion. A tuple only affects the output of a stateful operator if its count is positive.

Upon receiving a series of delta tuples, every query operator (1) updates its corresponding state, if necessary; (2) performs any internal computations such as predicate evaluation over the tuple or against state; (3) constructs a set of output delta tuples. Joins follow the rules established in [16]. For aggregation operators that compute minimum (or maximum) values, we must further extend the internal state management to keep track of *all* values encountered — such that, e.g., we can recover the "second-from-minimum" value. If the minimum is deleted, the operator should propagate an update delta, replacing its previous output with the next-best-minimum for the associated group (and conversely for maximum).

Challenge: recomputation of pruned state. While datalog allows us to propagate updates through rules, a major challenge is that the pruning strategies of Section 4 are achieved *indirectly*. In this section we detail how we incrementally re-create pruned state as necessary. Section 5.1 shows how we incrementally maintain the output of aggregate selection and "undo" tuple source suppression. Section 5.2 describes how to incrementally adjust the reference counts and maintain the pruned plans. Section 5.3 shows how we can incrementally modify the pruning bounds and the affected plans. Finally, Section 5.4 discusses how we leverage the ideas gained from the declarative perspective solving the problem into more traditional procedural-based query optimization frameworks and several optimization techniques.

5.1 Incremental Aggregate Selection

Aggregate selection [28] prunes state against bounds and does not consider how incremental maintenance might change the bound itself. Our incremental aggregate selection algorithm is a generalization of the non-incremental case we describe in Section 4.1. Recall that we push down a selection predicate, PlanCost < BestCost, within the Fixpoint operator that generates PlanCost. To illustrate how this works, consider how we may revise BestCost and BestPlan after encountering an insertion, deletion or update to PlanCost. There are four possible cases:

1. Upon an *insertion* PlanCost[+c], set BestCost to **min** (c, current BestCost).
2. Upon a *deletion* PlanCost[-c], set BestCost to the next-best PlanCost iff the current BestCost is equal to c.
3. Upon a *cost update* PlanCost[c→c′], if $c < c'$, set BestCost to **min** (c', next-best PlanCost) iff the current BestCost is equal to c.
4. Upon a *cost update* PlanCost[c→c′], if $c > c'$, if the current BestCost is equal to c, then set BestCost to c'; else set BestCost to **min** (c', current BestCost).

Each PlanCost tuple denotes a newly computed cost associated with a physical plan, and a BestCost tuple denotes the best cost that has been computed so far for this physical plan's expression-property pair. We update BestCost based on the current state of PlanCost. In Cases 1 and 4, we can directly compute updates to BestCost. To support Cases 2 and 3, we modify the aggregate operator to buffer *multiple* alternative PlanCost tuples, including those that are non-minimal; we use view-maintenance techniques to determine when we should replace the current minimum-value with the next-best value or a new value [20]. In our implementation we use a priority queue to order the candidate "minimum" tuples.

We may also need to re-introduce tuples in SearchSpace that were suppressed when they led to PlanCost tuples that were pruned, we achieve this by propagating an insertion (rather than deletion as in Section 4.1) to the previous stage.

5.2 Incremental Reference Checking

Once we have updated the set of viable plans for given expressions in the search space, we must consider how this impacts the viability of their subplans: we must incrementally update the reference counts on the child expressions to determine if they should be left the same, re-introduced, or pruned. As before, we simplify this process and make it order-independent through the use of incremental maintenance techniques.

We incrementally and recursively maintain the reference counts for each expression-property pair whenever an associated plan in the PlanCost relation is inserted, deleted or updated. When a new entry is inserted into PlanCost, we increment the count of each of its child expression-property pairs; similarly, whenever an existing entry is deleted from PlanCost, we decrement each child reference count. Replacement values for PlanCost entries do not change the reference counts, but may recursively affect the PlanCost entries for super-expressions. Whenever a count goes from 0 to 1 (or drops from 1 to 0) we recompute (prune, respectively) all of the physical plans associated with this expression-property pair.

If we combine this strategy with aggregate selection, only the best-cost plan needs to be pruned or re-introduced (all others are pruned via aggregate selection). Similar to Section 5.1, the aggregate operators internally maintain a record of all PlanCost tuples they have received as input, so "next-best" plans can be retrieved if the best-cost entry gets deleted or updated to a higher cost value. During incremental updates, we only propagate changes affecting the old and new best-cost plan and all recursively dependent plans.

5.3 Incremental Branch-and-bounding

We next consider how to incrementally maintain the branch-and-bound pruning structure of Section 4.3: as new costs for any operation are discovered, we must recursively recompute the bounds for all super-expressions. As necessary we then update PlanCost and SearchSpace tuples based on the updated bounds. Recall from Figure 4 that the Bound relation's contents are computed recursively based on the **max** bounds derived from parent plans; and also based on the **min** values for equivalent plan costs. Hence, an update to LocalCost or BestCost may affect the entries in Bound. Here we again rely in part on the fact that Bound is a recursive query and we can incrementally maintain it, then use its new content to adjust the pruning. We illustrate the handling of cost updates by looking at what happens when a cost *increases*.

Suppose a plan's LocalCost increases. As a consequence of the rules in Figure 4, the ParentBound of this plan's children may increase due to rules r1 and r2. MaxBound is then updated by r3 to be the maximum of the ParentBound entries: hence it may also increase. As in the previous cases, the internal aggregate operator for ParentBound maintains all input values; thus, it can recompute the new minimum bound and output a corresponding update from old to new value. Finally, as a result of the updated ParentBound, Bound in r4 may also increase. The process may continue recursively to this plan's descendant expression-property pairs, until Bound has converged to the correct bounds for all expression-property pairs.

Alternatively, suppose an expression-property pair's BestCost estimate increases (e.g., due to discovering the machine is heavily loaded). This may trigger an update to the corresponding entry in Bound (via rule r4). Moreover, via rules r1 and r2, an update to this bound may affect the bounds on the parent expression, i.e., ParentBound, and thus affecting any expression whose costs were pruned via ParentBound.

The cases for handling cost *decreases* are similar. Sometimes we may get simultaneous changes in both directions. Consider, for instance, that an expression's cost bound may increase, as in the previous paragraph. At the same time, perhaps the expression-property pair's ParentBound may decrease. Any equivalent plan (sibling expression) for our original expression-property pair is bounded *both* by the bounds of sibling expressions and parents. As ParentBound decreases, MaxBound and Bound may also potentially decrease through r3 and r4. The results are guaranteed to converge to the best of the sibling and parent bounds.

So far we focused only on how to update bounds given updated cost information; of course, there is the added issue of updating the pruning results. Recall in Section 4.3 that we evaluate the following predicate ϕ before propagating a newly generated PlanCost value: if PlanCost < Bound then set Bound to PlanCost. When PlanCost or Bound is updated, we can end up in any of 3 cases:

1. Upon a plan cost update PlanCost`[+c]`,PlanCost`[-c]` or PlanCost`[c→c']`: if predicate ϕ's result changes from false to true, then emit an insertion of the PlanCost tuple; otherwise if ϕ's result changes from true to false, then emit a deletion. Incrementally update the corresponding Bound entry, including its aggregated cost value, as a result.
2. Upon an update on Bound`[b→b']` where $b < b'$: for those tuples t in PlanCost where $b < t.cost < b'$, re-insert t into PlanCost and re-insert t's counterpart in SearchSpace to *undo* tuple source suppression.
3. Upon an update on Bound`[b→b']` where $b > b'$: for those tuples t in PlanCost where $b > t.cost > b'$, prune tuple t from PlanCost and delete t's counterpart from SearchSpace via tuple source suppression.

The first step resembles incremental aggregate selection (Section 5.1), except that here the condition check is not on BestCost but rather on Bound. Essentially we want to incrementally update Bound based on the current bounding status, hence a sorted list of PlanCost tuples needs to be maintained.

An interesting observation of Cases 2 and 3 is that an update on Bound may affect the pruned or propagated plans as well. If a bound is raised, it may re-introduce previously pruned plans; if a bound is lowered, it may incrementally prune previously viable plans. If incremental aggregate selection is used, then only the optimal plan among the pruned plans needs to be revisited. SearchSpace is again updated via sideways information passing.

5.4 Adapting Traditional Optimizers

Our declarative query re-optimizer enabled us to define pruning strategies independently from execution flow, and guaranteed correctness. This has given us a specification for pruning that we can "retrofit" into traditional bottom-up or top-down query optimizers. We can modify the procedural query optimizer to incrementally recompute portions of a query plan (and the contents of the memoization table), and propagate control flow upwards or downwards as appropriate to propagate the effects of a change. Then a variety of our techniques from the prior sections can similarly be adapted:

- *Incremental aggregate selection* can be used to memoize suboptimal plans as well as optimal ones, and only propagate changes that affect the choice of a "best" alternative plan for a given expression.
- *Incremental branch-and-bound* can maintain a table of bounds for subplans, which can be used for pruning.
- Separating the bounds computation enables it to be updated during enumeration both of alternative plans and superplans, as well as smaller subplans.

For both declarative and procedural-based query optimizers, we can also apply two main optimizations to minimize recomputation and bookkeeping. We term the first idea *lazy propagation*: only propagate cost changes if they might affect the choice of a super-plan; keep any remaining changes local to a node. For example, if we know in advance that an optimal subplan has not changed or it has even decreased in cost, we only need to consider alternative (previously suboptimal) plans whose costs may also have decreased.

The second optimization is to trade-off memoization versus recomputation, to reduce the cost of bookkeeping for suboptimal plans. On one extreme, if we memoize nothing during pre-optimization, then re-optimization will always need to do a full recomputation. On the other extreme, if we memoize everything, we might use up too much memory for bookkeeping. Hence in some cases, we can choose to only memoize the suboptimal plans most likely to become viable if small changes are made; and choose to discard the other alternatives, later recomputing them as necessary. This is a time and space complexity trade-off, but often a small amount of memoization can limit the recomputation to a few nodes in the and-or graph. We can also cache the output when a recomputation occurs, to avoid multiple revisits of the same node during re-optimization.

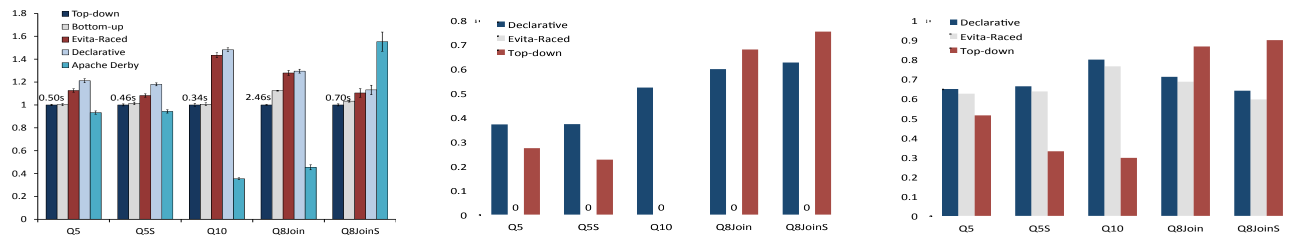
6. EVALUATION

In this section, we discuss the implementation and evaluate the performance of our incremental re-optimizer both versus other strategies, and as a primitive for adaptive query processing. (Note that we reuse existing adaptive query processing technologies from [17, 30]; our focus is to show that incremental re-optimization *improves* these systems.) We also demonstrate that our ideas work in both declarative and procedural query optimizer implementations.

We implemented the declarative optimizer in 10 datalog rules (see Appendix) with 8 auxiliary functions (including histogram generation, cost estimation, expression decomposition and so forth). Our goal was to implement as a proof of concept the **common core** of optimizer techniques — not an exhaustive set. We executed the optimizer in a modified version of the ASPEN system's query engine [20]. To support the pruning and incremental update propagation features in this paper, we added approximately 10K lines of code to the query engine. In addition, we developed a plan generator to translate the declarative optimizer into a dataflow graph as in Figure 2. Our experiments were performed on a single local node.

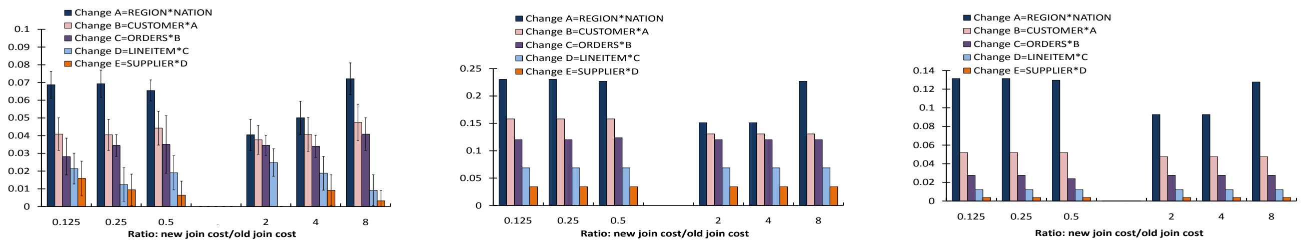
For comparison, we implemented in Java both a Volcano-style top-down query optimizer with branch-and-bound pruning, and a System-R-style bottom-up dynamic programming optimizer, which both reuse the histogram generation, cost estimation, and other core components as auxiliary functions in our declarative optimizer. We also built a variant of our declarative optimizer that only uses the pruning strategies of the Evita Raced declarative optimizer [9]. Wherever possible we used common code across the implementations to make fair comparisons. Finally, we extended this optimizer to support incremental computation using the declarative implementation as a reference.

Experimental Workload. For **repeated optimization** scenarios we use TPC-H queries, with data from the TPC-H and skewed TPC-D data generators [25] (Scale Factor 1, with Zipfian skew factor 0 for the latter). We focused on the single-block SQL queries: Q1, 3, 5, 6 and 10. (Q1 and 6 are aggregation-only queries; Q3



(a) Execution time (normalized to top-down style) (b) Pruning ratio: plan table entries (c) Pruning ratio: plan alternatives

Figure 5: Performance comparison for initial query optimization, across different optimizer architectures



(a) Execution time (normalized to top-down style) (b) Update ratio: plan table entries (c) Update ratio: plan alternatives

Figure 6: Performance during incremental re-optimization of TPC-H Q5 — change to join selectivity estimate

joins 3 relations; Q10 joins 4; and Q5 joins 6 relations). Our experiments showed that Q1, 3, and 6 are all simple enough to optimize that (1) there is not a compelling need to adapt, since there are few plan alternatives; (2) they completed in under 80msec on all implementations. (The declarative approach tended to add 10-50msec to these settings, as it has higher initialization costs.) Thus we focus our presentation on join queries with more than 3-way joins. To create greater query diversity, we modified the 4-way and larger join queries by removing aggregation — we constructed a simplified query Q5S. Finally, to test scale-up to larger queries, we manually constructed an eight-way join query, Q8Join, and its simplified version (removing aggregates), Q8JoinS. For **adaptive stream processing** we used the Linear Road benchmark [2]: We modified the largest query, called SegToll, into SegTollS. We show the TPC-H and LinearRoad benchmark queries used in our experiments in Table 5 in Appendix B.

Experimental Methodology. We aim to answer five questions:

- Can a declarative query optimizer perform at a rate competitive with procedural optimizers, for 4-way-join queries and larger?
- Does incremental query re-optimization show running time and search space benefits versus non-incremental re-optimization, for repeated query execution-over-static-data scenarios?
- How does each of our three pruning strategies (aggregate selection, reference counting, and recursive bounding) contribute to the performance?
- Can the ideas of incremental re-optimization studied from the declarative perspective be applied to traditional procedural-based query optimization frameworks, and what is their performance versus their non-incremental counterparts?
- Does incremental re-optimization improve the performance of cost-based adaptive query processing techniques for streaming?

The TPC-H benchmark experiments are conducted on a single local desktop machine: a dual-core Intel Core 2 2.40GHz with 2GB memory running 32-bit Windows XP Professional, and Java JDK 1.6. The Linear Road benchmark experiments are conducted on a single server machine: a dual-core Intel Xeon 2.83GHz with 8 GB memory running 64-bit Windows Server Standard. Performance results are averaged across 10 runs, and 95% confidence intervals are shown. We mark as 0 any results that are exactly zero.

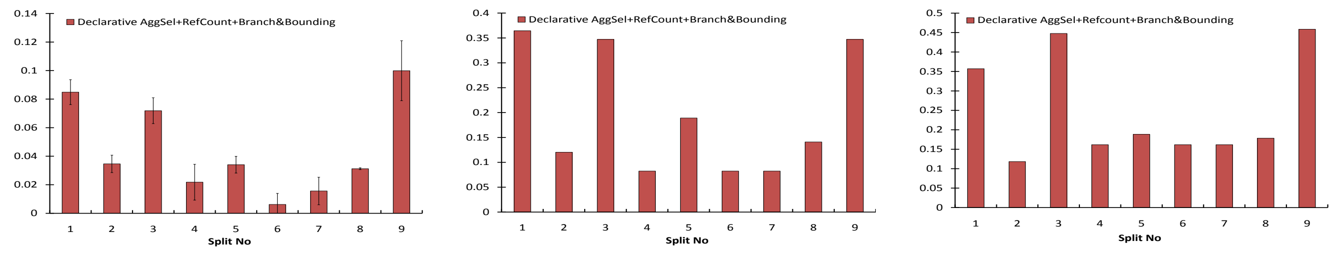
6.1 Declarative Optimization Performance

Our initial experiments focus on the question of **whether our declarative query optimizer can be competitive with procedural optimizers** based on bottom-up style enumeration through dynamic programming and top-down style enumeration with memoization and branch-and-bound pruning models. To show the value of the pruning techniques developed in this paper, we also measure the performance when our engine is limited to the pruning techniques developed in Evita Raced [9] (where pruning is only done against logically equivalent plans for the same output properties). Recall that all of our implementations share the same procedural logic (e.g., histogram derivation); their differences are in search strategy, dataflow, and pruning techniques.

We begin with a running time comparison among top-down style, bottom-up style, and declarative implementations (one using our sideways information passing strategies, and one based on the Evita Raced pruning heuristics) — shown in Figure 5 (a). This graph is normalized against the running times for our top-down style implementation (which is also included as a bar for visual comparison of the running times). Actual top-down running times are shown directly above the bar. Observe from the graph that the top-down strategy is always the fastest, though bottom-up style enumeration often approaches its performance due to simpler (thus, slightly faster) exploration logic. Our declarative implementation is not quite as fast as the dedicated procedural optimizers, with an overhead of 10-50%, but this is natural given the extra overhead of using a general-purpose engine and supporting incremental update processing. The Evita Raced-style declarative implementation is marginally faster in this setting, as it does less pruning. We also measure the query optimization performance of Apache Derby (chosen because it is a state of the art open-source DBMS implemented in Java, like our engine) to show how a well-known DBMS performs the same query, as a reference.[2] We shall see in later experiments that there are significant benefits to our more aggressive strategies during re-optimization — which is our focus in this work.

To understand the contributions of the different options, we next study their effectiveness in *pruning* the search space. We divide this into two parts: (1) pruning of expression-property entries in

[2]While some commercial DBMSs offer richer statistical or machine learning techniques, such capabilities are reliant on static data or predictable workloads.



(a) Execution time (normalized to top-down style) (b) Update ratio: plan table entries (c) Update ratio: plan alternatives

Figure 7: Performance during incremental re-optimization of TPC-H Q5 — updates to costs based on real execution over skewed data

the plan table, such that we do not need to compute and maintain *any* plans for a particular expression yielding a particular property; (2) pruning of *plan alternatives* for a particular expression-property pair. In terms of the and-or graph formulation of Figure 3, the first case prunes or-nodes and the second prunes and-nodes. We show these two cases in Figure 5 parts (b) and (c). We omit bottom-up style optimizers from this discussion, as they use a dynamic programming-based pruning model is not directly comparable.

Part (b) shows that our declarative implementation achieves pruning of approximately 35-80% of the plan table entries, resulting in large reductions in state (and, in many cases, reduced computation). We compare with the strategies used by Evita Raced, which never prunes plan table entries, and with our top-down style implementation. Our pruning strategies, which are flexible with respect to order of evaluation, are often more effective than the top-down strategy, which is limited to top-down enumeration with branch-and-bound pruning. (All pruning strategies' effectiveness depends on the specific order in which nodes are explored, and may be non-deterministic: better pruning is achieved when inexpensive options happen to be considered early. However, in the common case, high levels of pruning are observed, as we see next.)

Part (c) shows that our implementation prunes approximately 55-75% of the plans (95% confidence intervals are plotted but are extremely tight). It exceeds the pruning ratios obtained by the Evita Raced strategies by 4-8%, and often results in significantly greater pruning than the top-down style.

We conclude from these results that a declarative optimizer can be competitive in terms of running time and pruning.

6.2 Incremental Re-optimization

We next study the trade-offs in processing incremental changes to costs. A typical setting in a non-streaming context would be the repeated execution of a prepared statement query, where a bound variable affects costs. We measure, for a given update, how expensive it is to re-optimize the query and produce the new, predicted-optimal plan. Note that there exist no comparable techniques for incremental cost-based re-optimization, so we compare the gains versus those of re-running a complete optimization (as is done in [17, 30]). In these experiments, we consider running time — versus the running times for the best-performing initial optimization strategy, namely that of our top-down style implementation — as well as how much of the total search space gets re-enumerated. We consider re-optimization under "microbenchmark"-style simulated changes to costs, for synthetic updates as well as observed execution conditions over skewed data. We measured performance across the **full suite of queries** in our workload. However, since the results are representative, we focus our presentation on query Q5. We show more experimental results in the appendix.

6.2.1 Synthetic Changes to Subplan Costs

We first simulate what happens if we discover that an operator's output is not in accordance with our original selectivity estimates. Figures 6 (a)-(c) show the impact of synthetically injecting changes for each join expression's selectivity, and therefore the PlanCost of the related plans and their super-plans. For conciseness in the graph captions, we assign a symbol with each expression, e.g., the first join $Region \bowtie Nation$ is expression A, and the second join expression combines the output of A with data from the $Customer$ table, yielding $B = Customer \bowtie A$. We expect that changes to smaller subplans will take longer to re-optimize, and changes to larger subplans will take less time (due to the number of recursive propagation steps involved). We separately plot the results of changing each expression's selectivity value, as we change it along a range from 1/8 the predicted size through 8 times larger than the predicted size. Running times in part (a) are plotted relative to the top-down implementation's performance: we see that the speedups are at least a factor of 12, when the lowest-level join cost is updated; going up to over 300, when the topmost join operator's selectivity is changed. In general the speedups confirm that larger expressions are cheaper to update. We can observe from these last two figures that we recompute only a small portion of the search space.

6.2.2 Changes based on Real Execution

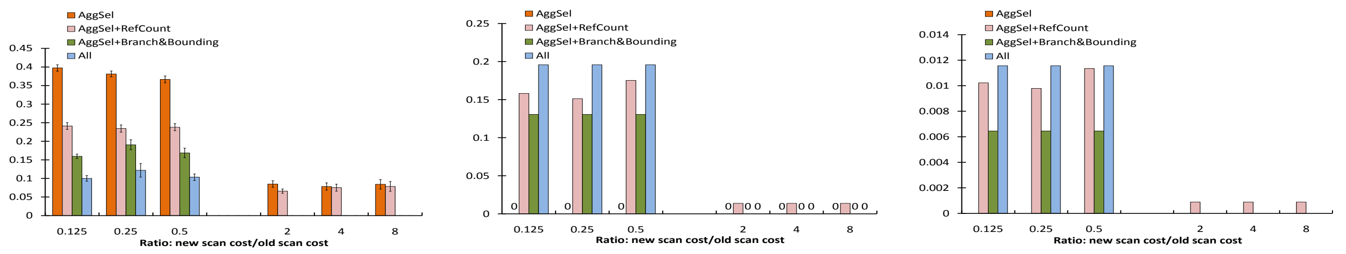
We now look at what happens when costs are updated according to an actual query execution. We took TPC-H Q5 and to gain better generality, we divided its input into 10 partitions (each having uniform distribution and independent variables) that would result in equal-sized output. We optimized the query over one such partition, using histograms from the TPC-H dataset. Then we ran the resulting query over different partitions of **skewed data** (Zipf skew factor 0.5, from the Microsoft Research skewed TPC-D generator [25]); each of which exhibits different properties. At the end we re-optimized the given the cumulatively observed statistics from the partition. We performed re-optimization on each of such interval, given the current plan and the revised statistics.

Figure 7 (a) shows the execution times for each round of incremental re-optimization, normalized against the running time of top-down style. We see that, as with the join re-estimation experiments of Figure 6, there are speedups of a factor of 10 or greater. In terms of throughput, the top-down model takes 500msec to perform one optimization, meaning it can perform 2 re-optimizations per second; whereas our declarative incremental re-optimizer can achieve 20-60 optimizations per second, and it can respond to changing conditions in 10-100msec. Again, Figure 7 (b) and (c) show that the speedup is due to significant reductions in the amount of state that must be recomputed.

Table 2 shows overall performance for this setting, over TPC-D scale factor 10 data. We omit the cost of initial optimization, and show (top row) the *overall cost* of incremental re-optimization plus execution for each split, vs. (bottom row) the cost of simply executing the originally optimized plan over the same data split. This corresponds to running a prepared statement, with selection predicates over different parts of the data, using incremental re-optimization given costs for the selected data – vs. a single query plan over all of the data.

6.3 Contributions of Pruning Strategies

Here we investigate how each of our pruning and incremental



(a) Execution time (vs top-down style) (b) Pruning ratio: plan table entries (c) Pruning ratio: plan alternatives

Figure 8: Contributions of pruning techniques for re-optimization of Q5 when $Orders$ has updated scan cost

Split number								
1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
2.07	**2.91**	**2.05**	2.02	2.91	2.02	**2.04**	**2.04**	**2.06**
2.77	3.49	2.66	2.04	2.95	2.01	2.23	2.86	3.01

Table 2: Query processing time (s) of incrementally re-optimized vs. single-stage TPCH-Q5, for splits of Fig. 7

strategies from Sections 4 and 5 contribute to the overall performance of our declarative optimizer. We systematically considered all techniques individually and in combination, unless they did not make sense (e.g., reference counting must be combined with one of the other techniques, and branch-and-bound requires aggregate selection to perform pruning of the search space). See Figure 8, where *AggSel* refers to aggregate selection with source tuple suppression; *RefCount* refers to reference counting; and *Branch&-Bounding* refers to recursive bounding. We consider aggregate selection in isolation and with the other techniques.

The incremental setting — shown for Q5 and changes to the orders table, in Figures 8 (a)-(c) — shows benefits in running time and pruned search space. In contrast to our other graphs for incremental re-optimization, plots (b) and (c) isolate the amount of pruning performed, rather than showing the total state updated. Our different techniques work best in combination, and each increases the amount of pruning.

Query	Num of Joins	Pruning Strategy	Space (KB)
Q10	4	No pruning	2211
		AggSel	253
		All pruning	412
Q5	6	No pruning	3046
		AggSel	266
		All pruning	445
Q8Join	8	No pruning	DNC
		AggSel	1046
		All pruning	1669

Table 3: Overhead of memoized state in declarative optimizer

Table 3 shows the total state size in our declarative optimizer. We measure the size of the objects instead of the JVM memory to alleviate the effects of garbage collection. We measure the space overhead of TPC-H Q10, Q5 and Q8Join, with 4, 6, 8 joins respectively. Since we employ the same memoization scheme for initial query pre-optimization and query re-optimization in the declarative approach, these numbers both relate to pre-optimization and re-optimization. The table shows the size of the unpruned state. (Here since the optimization time for Q8Join without pruning did not complete within over 2 minutes, due to a complete lack of pruning.) If we only prune by using AggSel, we have less state to maintain than the full pruning scenario, but the optimization time is worse, as shown in Figure 8.

6.4 Incrementalizing a Conventional Optimizer

To this point, we have focused on the performance of our datalog-based query optimizer implementation. As described in Section 5.4, some of the ideas from this implementation can also be retrofitted into more conventional optimizers (both top-down and bottom-up). Thus we measure, in Figure 9, how a traditional optimizer, retrofitted with incremental re-optimization, performs versus an equivalent non-incremental implementation.

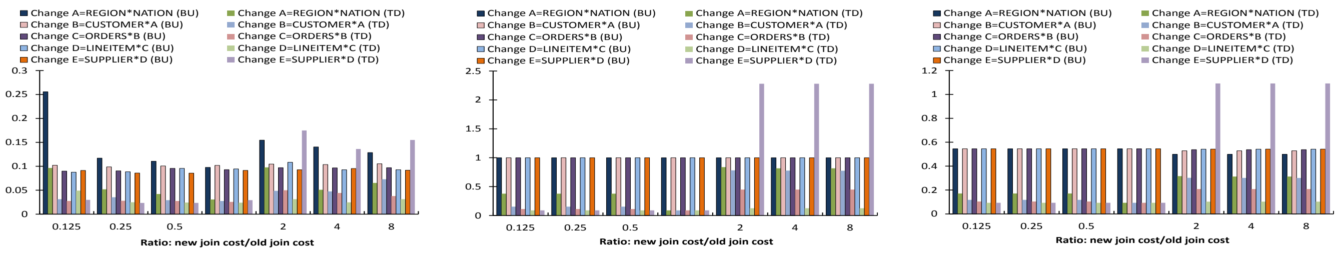
In this experiment, we see the cost of re-optimization if a join's actual cost is discovered to vary (by a factor ranging from 1/8th to 8x of its estimated cost) for TPC-H query Q5. In our procedural implementation we memoize not only the best plans, but alternative subplans. Figure 9 (a) shows the running time for incremental query re-optimization (normalized to the time for non-incremental counterparts), for several different queries. The "(BU)" entries are bottom-up implementations, whereas "(TD)" references a comparable top-down implementation. Observe that incremental re-optimization is 4-20 times faster than a complete optimization. As expected, the results validate that top-down strategies are generally superior to bottom-up ones, because they enable greater amounts of pruning. The only exception to this is the last data point, where we change the join cost of $E = Supplier * D$: here, increasing this join cost causes the top-down re-optimizer to recompute a larger number of nodes than the bottom-up implementation.

We can see this in more detail in Figure 9 (b), which shows the proportion of *updated* plan table entries (OR nodes), and in (c), which shows the proportion of *recomputed* plan alternatives (AND nodes). In general, the larger the subexpression for which a join cost changes, the less work is required to re-compute and update, and the more benefit provided by incremental re-optimization. Bottom-up re-optimizers generally incur the same amount of updated plan table entries as the non-incremental approach due to lack of pruning, and constantly recompute less than 60% of the plan alternatives; top-down re-optimizers usually incur far less updates and recomputations when a join cost decreases and more updates when a join cost increases.

6.5 Incremental Reoptimization for AQP

A major motivation for our work was to facilitate better *cost-based adaptive query processing*, especially for continuous optimization of stream queries. Our goal is to show the benefits of incremental reoptimization; we leave as future work a broader comparison of adaptive query processing techniques. Our final set of experiments shows how our techniques can be used within a standard cost-based adaptive framework, one based on the *data-partitioned* model of [17] where the optimizer periodically pauses plan execution, forming a "split" point from which it may choose a new plan and continue execution. In general, if we change plans at a split point, there is a challenge of determining how to combine state across the split. In contrast to [17] we chose *not* to defer the cross-split join execution until the end: rather, we used CAPE's *state migration* strategy [30] to transfer all existing state from the prior plan into the current one. As necessary, new intermediate state is computed. Note that our data-partitioned model could be combined with other cost-based adaptive schemes such as [18, 21].

To evaluate this setting, we combine unfold the various views comprising the SegToll query from the Linear Road benchmark [2]



(a) Optimization time ratio to non-incremental counterparts

(b) Update ratio: plan table entries

(c) Recompute ratio: plan alternatives

Figure 9: Performance of top-down vs bottom-up style procedural incremental re-optimization of TPC-H Q5 — upon change to join cost value

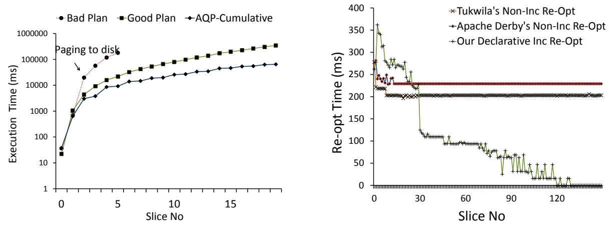


Figure 10: AQP execution (left) and re-optimization times

— resulting in a 5-way join plan SegTollS with multiple windowed aggregates. We use the standard Linear Road data generator to synthesize data whose characteristics frequently change.

In the adaptive setting, our goal is to have the optimizer start with zero statistical information on the data, and find a sequence of plans whose running time equals or betters the *single best static plan* that it would pick given complete information (e.g., histograms).

On the left of Figure 10, we see that our incremental AQP scheme provides **superior performance to the single best plan** ("good single plan"), if we re-optimize every 1 second. This is because the adaptive scheme has a chance to "fit" its plan to the local characteristics of whatever data is currently in the window, rather than having to choose one plan for all data.

Per Slice	Re-Opt Time	N.C. Exec Time	Total Time
1s	5.75s	2.20s	7.95s
5s	1.23s	6.82s	8.05s
10s	0.63s	13.35s	13.98s

Table 4: Frequency of Adaptation (20 sec stream)

A natural question is where the "sweet spot" is between query execution versus optimizer overhead. We measured, for several slice sizes, the total query processing time *over each new slice of data* (not considering the additional overhead of state migration, which depends on how similar the plans are). Table 4 shows significant gains in shrinking the interval from 10sec to 5sec, but little more gain to be had in going down to 1sec. Figure 10 shows that as we re-optimize and execute, the overhead of a non-incremental re-optimizer remains constant (about 200 msec each time), whereas the incremental re-optimization time drops off rapidly, going to nearly zero as the plan stabilizes. This means that the system has essentially *converged on a plan* and that new executions do not affect the final plan. We measure the performance against Tukwila-style query optimizer implementation [17], as well as the Apache Derby query optimizer. We see that the majority of incremental re-optimizations are much faster than the non-incremental baselines.

6.6 Experimental Conclusions

We summarize the answers to questions posed at the start of this section. First, our declarative query optimizer performs respectably when compared to a procedural query optimizer, for initial optimization: despite the overhead of a full query processor, it gets within 10-50% of the running times of a dedicated optimizer. It more than recovers this overhead during incremental re-optimization, where it typically shows an order-of-magnitude speedup or better. Such gains are largely due to having to re-enumerate a much smaller space of plans. In addition, our pruning techniques of Section 4 and Section 5 each contribute in a meaningful way to the overall performance of incremental re-optimization. We also successfully adapted our novel pruning techniques to a procedural implementation, providing speedup factors of more than 4. Finally, our incremental re-optimization techniques enable *finer-grained* adaptivity and hence better overall performance. Overhead decreases as the system *converges on a single plan*.

7. RELATED WORK

Our work takes a first step towards supporting continuous adaptivity in a distributed (e.g., cloud) setting where correlations and runtime costs may be unpredictable at each node. Fine-grained adaptivity has previously only been addressed in the query processing literature via heuristics, such as flow rates [3, 29], that continuously "explore" alternative plans rather than using long-term cost estimates. Exploration adds overhead even when a good plan has been found; moreover, for joins and other stateful operators, the flow heuristics has been shown to result in state that later incurs significant costs [10]. Other strategies based on filter reordering [4] are provably optimal, but only work for selection-like predicates. Full-blown cost-based re-optimization can avoid these future costs but was previously only possible on a coarse-grained (high multiple seconds) interval [17, 30].

Our use of declarative techniques to specify the optimizer was inspired in part by the Evita Raced [9] system. However, their work aims to construct an entire optimizer using reprogrammable datalog rules, whereas our goal is to effectively perform incremental maintenance of the output query plan. We seek to fully match the pruning techniques of conventional optimizers following the System R [27] and Volcano [14] models. Our results show for the first time that a declarative optimizer *can* be competitive with a procedural one, even for one-time "static" optimizations, and produce large benefits for future optimizations.

8. CONCLUSIONS AND FUTURE WORK

To build large-scale, pipelined query processors that are reactive to conditions across a cluster, we must develop new adaptive query processing techniques. This paper represents the first step towards that goal: namely, a fully cost-based architecture for incrementally re-optimizing queries. As future work, we plan to study how our declarative execution model parallelizes across multi-core hardware and clusters, and how it can be extended to consider the cost of *changing* a plan given existing query execution state.

9. ACKNOWLEDGMENTS

This work was funded in part by grants NSF IIS-0713267, IIS-1050448, CNS-0845552, CNS-1218066, CNS-1513679, and NIH 1U01EB020954-01.

References

[1] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *VLDB*, 2000.

[2] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear road: a stream data management benchmark. In *VLDB*, pages 480–491, 2004.

[3] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD*, 2000.

[4] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *SIGMOD*, 2004.

[5] A. Behm, V. R. Borkar, M. J. Carey, R. Grover, C. Li, N. Onose, R. Vernica, A. Deutsch, Y. Papakonstantinou, and V. J. Tsotras. Asterix: towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases*, 29(3):185–216, 2011.

[6] N. Bruno and S. Chaudhuri. Exploiting statistics on query expressions for optimization. In *SIGMOD*, 2002.

[7] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

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

[9] T. Condie, D. Chu, J. M. Hellerstein, and P. Maniatis. Evita raced: metacompilation for declarative networks. *PVLDB*, 1(1):1153–1165, 2008.

[10] A. Deshpande and J. M. Hellerstein. Lifting the burden of history from adaptive query processing. In *VLDB*, pages 948–959, 2004.

[11] A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 2007.

[12] A. Dutt and J. R. Haritsa. Plan bouquets: query processing without selectivity estimation. In *SIGMOD*, pages 1039–1050, 2014.

[13] G. Graefe. The Cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[14] G. Graefe and W. J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *ICDE*, pages 209–218, 1993.

[15] A. Gupta, H. Jagadish, and I. S. Mumick. In P. Apers, M. Bouzeghoub, and G. Gardarin, editors, *EDBT*. Berlin / Heidelberg, 1996.

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

[17] Z. G. Ives, A. Y. Halevy, and D. S. Weld. Adapting to source properties in processing data integration queries. In *SIGMOD*, pages 395–406, 2004.

[18] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD*, pages 106–117, 1998.

[19] L. Liu, C. Pu, R. Barga, and T. Zhou. Differential evaluation of continual queries. Technical Report TR95-17, University of Alberta, June 1995.

[20] M. Liu, N. E. Taylor, W. Zhou, Z. G. Ives, and B. T. Loo. Recursive computation of regions and connectivity in networks. In *ICDE*, 2009.

[21] V. Markl, V. Raman, G. Lohman, H. Pirahesh, D. Simmen, and M. Cilimdzic. Robust query processing through progressive optimization. In *SIGMOD*, 2004.

[22] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1):330–339, 2010.

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

[24] B. Mozafari, E. Z. Y. Goh, and D. Y. Yoon. Cliffguard: A principled framework for finding robust database designs. In *SIGMOD*, pages 1167–1182, New York, NY, USA, 2015.

[25] V. Narasayya. TPC-D skewed data generator. 1999.

[26] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*, volume 29(2), pages 249–260, 2000.

[27] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34, 1979.

[28] S. Sudarshan and R. Ramakrishnan. Aggregation and relevance in deductive databases. In *VLDB*, 1991.

[29] S. D. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *SIGMOD*, pages 37–48, 2002.

[30] Y. Zhu, E. A. Rundensteiner, and G. T. Heineman. Dynamic plan migration for continuous queries over data streams. In *SIGMOD*, 2004.

APPENDIX

A. DATALOG RULES FOR OPTIMIZER

R1: SearchSpace$(expr, prop, index, logOp, phyOp, lExpr,$
$lProp, rExpr, rProp)$:-
Expr$(expr, prop)$, Fn_isleaf$(expr, false)$,
Fn_split$(expr, prop, \underline{index}, \underline{logOp}, \underline{phyOp}, \underline{lExpr},$
$\underline{lProp}, \underline{rExpr}, \underline{rProp})$;

R2: SearchSpace$(expr, prop, index, logOp, phyOp,$
$lExpr, lProp, rExpr, rProp)$:-
SearchSpace$(-, -, -, -, -, expr, prop, -, -)$,
Fn_isleaf$(expr, false)$,
Fn_split$(expr, prop, \underline{index}, \underline{logOp}, \underline{phyOp},$
$\underline{lExpr}, \underline{lProp}, \underline{rExpr}, \underline{rProp})$;

R3: SearchSpace$(expr, prop, index, logOp, phyOp,$
$lExpr, lProp, rExpr, rProp)$:-
SearchSpace$(-, -, -, -, -, -, -, expr, prop)$,
Fn_isleaf$(expr, false))$,
Fn_split$(expr, prop, \underline{index}, \underline{logOp}, \underline{phyOp},$
$\underline{lExpr}, \underline{lProp}, \underline{rExpr}, \underline{rProp})$;

R4: SearchSpace$(expr, prop, -,' scan', phyOp, -, -, -, -))$:-
SearchSpace$(-, -, -, -, -, expr, prop, -, -)$,
Fn_isleaf$(expr, true)$, Fn_phyOp$(prop, \underline{phyOp})$;

R5: SearchSpace$(expr, prop, -,' scan', phyOp, -, -, -, -)$:-
SearchSpace$(-, -, -, -, -, -, -, expr, prop)$,
Fn_isleaf$(expr, true)$, Fn_phyOp$(prop, \underline{phyOp})$;

R6: PlanCost$(expr, prop, index, logOp, phyOp, -, -, -, -,$
$md, cost)$:-
SearchSpace$(expr, prop, index, logOp, phyOp, -, -, -, -)$,
Fn_scansummary$(expr, prop, \underline{md})$,
Fn_scancost$(expr, prop, md, \underline{cost})$;

R7: PlanCost$(expr, prop, index, logOp, phyOp,$
$lExpr, lProp, -, -, md, cost)$:-
SearchSpace$(expr, prop, index, logOp, phyOp,$
$lExpr, lProp, -, -,$ Fn_isleaf$(lExpr, false)$,
PlanCost$(lExpr, lProp, -, -, -, -, -, -, -, lMd, lCost)$,
Fn_nonscansummary$(expr, prop, index, logOp, lMd, -, \underline{md})$,
Fn_nonscancost$(expr, prop, index, logOp, phyOp,$
$lExpr, lProp, -, -, md, \underline{localCost})$,
Fn_sum$(lCost, null, localCost, \underline{cost})$;

Q1: SELECT l_returnflag, l_linestatus, sum(l_quantity) as sum_qty, sum(l_extendedprice) as sum_base_price, sum(l_extendedprice*(1-l_discount)) as sum_disc_price, sum(l_extendedprice*(1-l_discount)*(1+l_tax)) as sum_charge, avg(l_quantity) as avg_qty, avg(l_extendedprice) as avg_price, avg(l_discount) as avg_disc, count(*) as count_order FROM lineitem WHERE l_shipdate ≤ '1998-09-01' GROUPBY l_returnflag, l_linestatus;
Q3: SELECT l_orderkey, sum(l_extendedprice*(1-l_discount)) as revenue, o_orderdate, o_shippriority FROM customer, orders, lineitem WHERE c_mktsegment = 'MACHINERY' and c_custkey = o_custkey and l_orderkey = o_orderkey and o_orderdate < '1995-03-15' and l_shipdate > '1995-03-15' GROUPBY l_orderkey, o_orderdate, o_shippriority;
Q5: SELECT n_name, sum(l_extendedprice * (1 - l_discount)) as revenue FROM customer, orders, lineitem, supplier, nation, region WHERE c_custkey = o_custkey and l_orderkey = o_orderkey and l_suppkey = s_suppkey and c_nationkey = s_nationkey and s_nationkey = n_nationkey and n_regionkey = r_regionkey and r_name = 'AMERICA' and o_orderdate ≥ CAST('1993-01-01' and o_orderdate < '1994-01-01' GROUPBY n_name;
Q5S: SELECT n_name, l_extendedprice * (1 - l_discount) FROM customer, orders, lineitem, supplier, nation, region WHERE c_custkey = o_custkey and l_orderkey = o_orderkey and l_suppkey = s_suppkey and c_nationkey = s_nationkey and s_nationkey = n_nationkey and n_regionkey = r_regionkey and r_name = 'AMERICA' and o_orderdate ≥ '1993-01-01' and o_orderdate < '1994-01-01';
Q6: SELECT sum(l_extendedprice*l_discount) as revenue FROM lineitem WHERE l_shipdate ≥ '1994-01-01' and l_shipdate < '1995-01-01' and l_discount ≥ 0.06 - 0.01 and l_discount ≤ 0.06 + 0.01 and l_quantity < 24.2;
Q10: SELECT c_custkey, c_name, sum(l_extendedprice * (1 - l_discount)) as revenue, c_acctbal, n_name, c_address, c_phone, c_comment FROM customer, orders, lineitem, nation WHERE c_custkey = o_custkey and l_orderkey = o_orderkey and o_orderdate ≥ '1993-06-01' and o_orderdate < '1993-09-01' and l_returnflag = 'R' and c_nationkey = n_nationkey GROUPBY c_custkey, c_name, c_acctbal, c_phone, n_name, c_address, c_comment;
Q8Join: SELECT c_name, p_name, ps_availqty, s_name, o_custkey, r_name, n_name, sum(l_extendedprice * (1 -l_discount)) FROM orders, lineitem, customer, part, partsupp, supplier, nation, region WHERE o_orderkey = l_orderkey and c_custkey = o_custkey and p_partkey = l_partkey and ps_partkey = p_partkey and s_suppkey = ps_suppkey and r_regionkey = n_regionkey and s_nationkey = n_nationkey GROUPBY c_name, p_name, ps_availqty, s_name, o_custkey, r_name, n_name;
Q8JoinS: SELECT c_name, p_name, ps_availqty, s_name, o_custkey, r_name, n_name, l_extendedprice * (1 -l_discount) FROM orders, lineitem, customer, part, partsupp, supplier, nation, region WHERE o_orderkey = l_orderkey and c_custkey = o_custkey and p_partkey = l_partkey and ps_partkey = p_partkey and s_suppkey = ps_suppkey and r_regionkey = n_regionkey and s_nationkey = n_nationkey;
SegTollS: SELECT r1_expway, r1_dir, r1_seg, COUNT(distinct r5_xpos) FROM CarLocStr [size 300 time] as r1, CarLocStr [size 1 tuple partition by expway, dir, seg] as r2, CarLocStr [size 1 tuple partition by caid] as r3, CarLocStr [size 30 time] as r4, CarLocStr [size 4 tuple partition by carid] as r5 WHERE r2_expway = r3_expway and r2_dir = 0 and r3_dir = 0 and r2_seg < r3_seg and r2_seg > r3_seg - 10 and r3_carid = r4_carid and r3_carid = r5_carid and r1_expway = r2_expway and r1_dir = r2_dir and r1_seg = r2_seg GROUP BY r5_carid, r2_expway, r2_dir, r2_seg;

Table 5: Queries modified based on TPC-H and LinearRoad benchmark queries used in our experiments

R8: $\mathsf{PlanCost}(expr, prop, index, logOp, phyOp,$
$lExpr, lProp, rExpr, rProp, md, cost)$:-
$\mathsf{SearchSpace}(expr, prop, index, logOp, phyOp,$
$lExpr, lProp, rExpr, rProp),$
$\mathsf{Fn\_isleaf}(lExpr, false), Fn\_isleaf(rExpr, false),$
$\mathsf{PlanCost}(lExpr, lProp, -, -, -, -, -, -, -, lMd, lCost),$
$\mathsf{PlanCost}(rExpr, rProp, -, -, -, -, -, -, -, rMd, rCost),$
$\mathsf{Fn\_nonscansummary}(expr, prop, index, logOp, lMd, rMd, \underline{md}),$
$\mathsf{Fn\_nonscancost}(expr, prop, index, logOp, phyOp,$
$lExpr, lProp, rExpr, rProp, md, \underline{localCost}),$
$\mathsf{Fn\_sum}(lCost, rCost, localCost, \underline{cost});$
R9: $\mathsf{BestCost}(expr, prop, min < cost >)$:-
$\mathsf{PlanCost}(expr, prop, index, logOp, phyOp,$
$lExpr, lProp, rExpr, rProp, md, cost);$
R10:$\mathsf{BestPlan}(expr, prop, index, logOp, phyOp,$
$lExpr, lProp, rExpr, rProp, md, cost)$:-
$\mathsf{BestCost}(expr, prop, cost),$
$\mathsf{PlanCost}(expr, prop, index, logOp, phyOp,$
$lExpr, lProp, rExpr, rProp, md, cost);$

B. EXPERIMENTAL QUERIES

Table 5 shows our experimental query workload, which consists of the main select-project-join-aggregate queries from TPC-H (queries 1, 3, 5, 6, 8, and 10) with some modifications; as well as a streaming query from Linear Road [2].

C. PROOFS OF PROPOSITIONS IN SECTION 3

PROOF OF PROPOSITION 5. $\Rightarrow$: Prove by contradiction. If T^s is T^s_{OPT}, then suppose T^s is *not* the subtree of T_{OPT}. Let T'^s be the subtree of T_{OPT}. Since $T^s = T^s_{OPT}$, then we must have $\mathsf{PlanCost}(T^s\langle E^s, p^s\rangle) < \mathsf{PlanCost}\ (T'^s\langle E^s, p^s\rangle)$. If we substitute T'^s with T^s in the tree T_{OPT}, we get a new tree T'', with $\mathsf{PlanCost}(T''\langle E, p\rangle) < \mathsf{PlanCost}\ (\ T_{OPT}\ \langle E, p\rangle)$. Contradiction to the definition of T_{OPT}.

$\Leftarrow$: Prove by contradiction. If T^s is the subtree of T_{OPT}, then suppose T^s is *not* T^s_{OPT}, since plans have distinct costs, we have $\mathsf{PlanCost}(T^s\langle E^s, p^s\rangle) > \mathsf{PlanCost}(T^s_{OPT}\langle E^s, p^s\rangle)$. If we substitute subtree T^s with subtree T^s_{OPT} in the tree T_{OPT}, we get a new plan tree T', which has $\mathsf{PlanCost}\ (T'\langle E, p\rangle) < \mathsf{PlanCost}\ (T_{OPT}\langle E, p\rangle)$. Contradiction to the definition of T_{OPT}. $\square$

PROOF OF PROPOSITION 6. $\Rightarrow$: Prove by contradiction. If $\mathsf{RefCount}\ \langle E^s, p^s\rangle = 0$, then any plan tree whose root is the AND node representing the parent of $\langle E^s, p^s\rangle$ is pruned. Suppose T^s is a subtree of $T_{OPT}\langle E, p\rangle$, then consider the immediate parent node of T^s in T_{OPT}, let it be $T^{s'}$. According to Proposition 5, $T^{s'} = T^{s'}_{OPT}$. This means it has the minimal cost to subexpression $\langle E^{s'}, p^{s'}\rangle$, is on the optimal tree of the final optimal plan, but has been pruned. Contradiction.

$\Leftarrow$: Prove by contradiction. If T^s is *not* a subtree of T_{OPT}, according to Proposition 5, T^s is not T^s_{OPT}. Hence, $\mathsf{PlanCost}(T^s_{OPT}) < \mathsf{PlanCost}(T^s)$. Suppose T^s has the reference count other than zero, then it must have at least a parent plan that has not been pruned. Suppose that parent plan is $T'^{s'}$. If we substitute T^s with T^s_{OPT} in $T'^{s'}$, we get a plan tree $T''^{s'}$ that has a smaller cost than $T'^{s'}$. Hence $T'^{s'}$ can be pruned in the final result because it is suboptimal to $T''^{s'}$. Contradiction. $\square$

PROOF OF PROPOSITION 7. $\Rightarrow$: Prove by contradiction. Suppose $\mathsf{PlanCost}(T^s) > \mathsf{Bound}\langle E, p\rangle$. And suppose T^s is a subtree of the optimal plan tree T_{OPT}, according to Proposition 5, T^s is T^s_{OPT}. Hence, $\mathsf{PlanCost}(T^s_{OPT}) > \mathsf{Bound}\langle E, p\rangle$. Now we prove by induction that if T^s is a subtree of T' of subexpression $E^{s'}$ and property $p^{s'}$, then $\mathsf{PlanCost}(T'_{OPT}) > \mathsf{Bound}\langle E^{s'}, p^{s'}\rangle$.

First, the base case $\mathsf{PlanCost}(T^s_{OPT}) > \mathsf{Bound}\langle E^s, p^s\rangle$ holds. Now, if $\mathsf{PlanCost}(T^s_{OPT}) > \mathsf{Bound}\langle E^s, p^s\rangle$, according to the definition of Bound where $\mathsf{Bound}\langle E^s, p^s\rangle$ = min($\mathsf{BestCost}(T^s)$,

$\mathsf{maxBound}\langle E^s, p^s\rangle$), we have $\mathsf{Bound}\langle E^s, p^s\rangle = \mathsf{MaxBound}\langle E^s, p^s\rangle < \mathsf{PlanCost}(T^s_{OPT})$. According to the definition of MaxBound, $\mathsf{MaxBound}\langle E^s, p^s\rangle = \max(\mathsf{ParentBound}\langle E^s, p^s\rangle)$, therefore, for any arbitrary sibling OR node of T^s, $Sib(T^s)$, and corresponding parent AND node of T^s, $Par(T^s)$ of expression $E^{s'}$ and property $P^{s'}$, we have $\mathsf{Bound}\langle E^{s'}, p^{s'}\rangle$ - $\mathsf{BestCost}(Sib(T^s))$ - $\mathsf{LocalCost}(Par(T^s)) \leq \mathsf{MaxBound}\langle E^s, p^s\rangle < \mathsf{PlanCost}(T^s_{OPT})$. Hence, $\mathsf{Bound}\langle E^{s'}, p^{s'}\rangle < \mathsf{BestCost}(Sib(T^s))$ + $\mathsf{BestCost}(T^s)$ + $\mathsf{LocalCost}$ $(Par(T^s))$. Because this holds for any sibling and parent, we have $\mathsf{Bound}\langle E^{s'}, p^{s'}\rangle <$ min($\mathsf{BestCost}(Sib(T^s))$ + $\mathsf{BestCost}(T^s)$ + $\mathsf{LocalCost}(Par(T^s))$) = $\mathsf{BestCost}(Par(T^s))$ = $\mathsf{PlanCost}(Par(T^s)_{OPT})$.

By applying this induction we prove that if T^s is a subtree of T' of subexpression $E^{s'}$ and property $p^{s'}$, then $\mathsf{PlanCost}(T'_{OPT}) > \mathsf{Bound}\langle E^{s'}, p^{s'}\rangle$. Since T^s is a subtree of the optimal plan tree T_{OPT}, hence $\mathsf{PlanCost}(T_{OPT}) > \mathsf{Bound}\langle E, p\rangle$. However, as T_{OPT} is the root, it has no parent, according to the definition of Bound, it should have $\mathsf{Bound}(T_{OPT})$ = $\mathsf{PlanCost}(T_{OPT})$. Contradiction.

$\Leftarrow$: Prove by contradiction. Suppose T^s is *not* a subtree of T_{OPT}, according to Proposition 5, T^s is not T^s_{OPT}, hence, $\mathsf{PlanCost}(T^s) > \mathsf{PlanCost}(T^s_{OPT})$. Suppose $\mathsf{PlanCost}(T^s) \leq \mathsf{Bound}\langle E^s, p^s\rangle$, then we have $\mathsf{PlanCost}(T^s_{OPT}) < \mathsf{PlanCost}(T^s) \leq \mathsf{Bound}\langle E^s, p^s\rangle$. According to the definition of Bound, $\mathsf{Bound}\langle E^s, p^s\rangle$ = min($\mathsf{BestCost}(T^s)$, $\mathsf{maxBound}\langle E^s, p^s\rangle$), we have $\mathsf{PlanCost}(T^s_{OPT}) < \mathsf{BestCost}(T^s)$, contradiction to the definition of T^s_{OPT}. $\square$

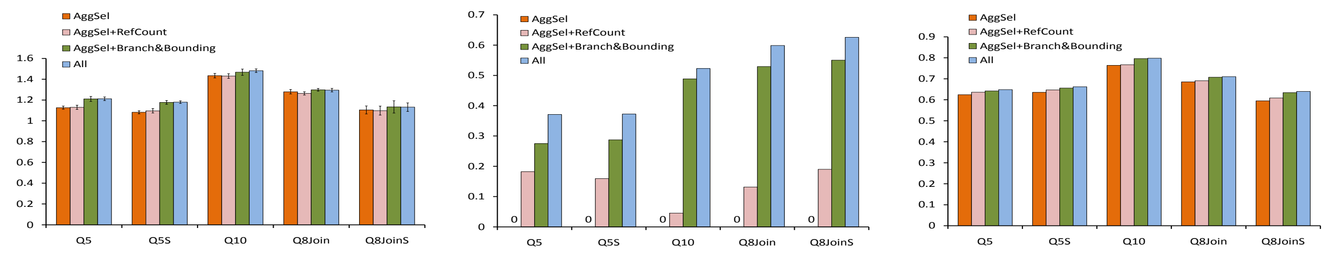
D. ADDITIONAL EXPERIMENTAL RESULTS

As a supplement to the figures in Section 6.3, Figures 11 (a)-(c) compare the three pruning strategies when performing initial query *pre-optimization* on various TPC-H queries. It can be observed that each of the pruning techniques adds a small bit of runtime overhead (never more than 10%) in this setting, as each requires greater computation and data propagation. Parts (b) and (c) show that each technique adds greater pruning capability, however.

For greater completeness, we show additional experimental results to gain more insights over a full range of TPC-H queries under different scenarios.

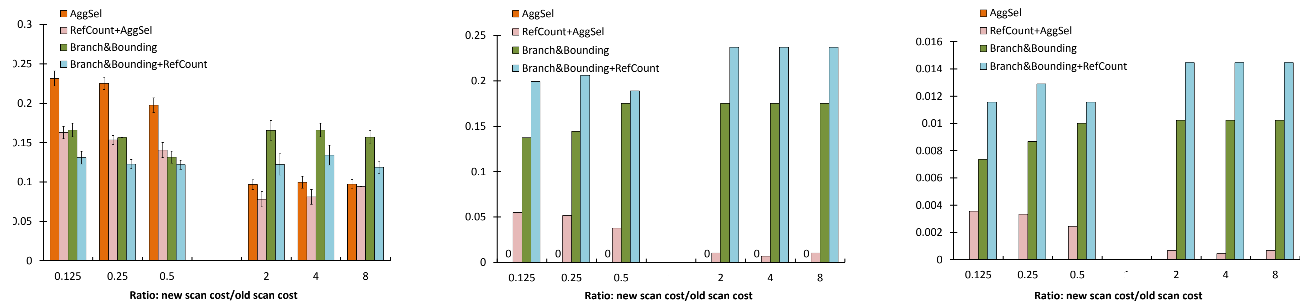
Figure 12 is a supplement to Figure 8 where it measures the performance breakdown of different pruning strategies during incremental re-optimization of TPC-H query Q5 when a different table, $Suppliers$, other than $Orders$, has updated its scan cost from 1/8 to 8x of the original. We can observe in the figure that each pruning strategy contributes in a meaningful way to the effectiveness of incremental re-optimization, as well.

Figure 13 and Figure 14 show a full range of incremental re-optimization performance upon a scan cost change of each base relation, for TPC-H query Q10 (a 4-way join query) and Q8Join (a 8-way join query as shown in Table 5) respectively. We can see from Figure 13 (a) and Figure 14 (a) that incremental re-optimization has at least 3x of speedup compared to a top-down style non-incremental optimizer. (b) and (c) show the numbers of plan table entries and plan alternatives updated during incremental re-optimization, to gain more insights on where the speedup comes from.



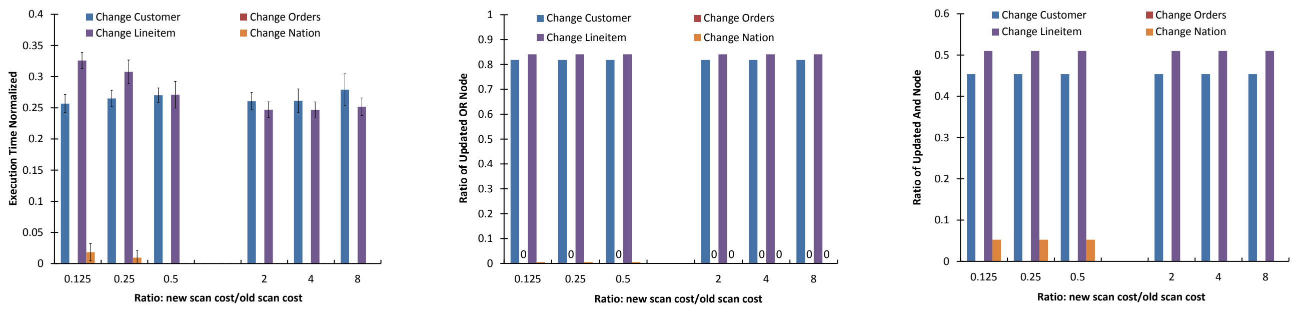
(a) Execution time (normalized to top-down style) (b) Pruning ratio: plan table entries (c) Pruning ratio: plan alternatives

Figure 11: **Performance breakdown of pruning techniques for initial optimization, across full query workload**



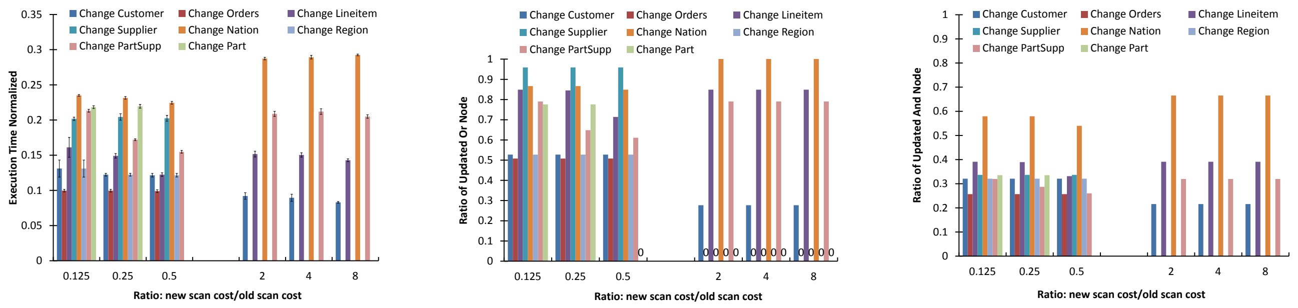
(a) Execution time (normalized to top-down style) (b) Pruning ratio: plan table entries (c) Pruning ratio: plan alternatives

Figure 12: **Performance breakdown of pruning techniques during incremental re-optimization of Q5 when** $Suppliers$ **has updated scan cost**



(a) Execution time (normalized to top-down style) (b) Update ratio: plan table entries (c) Update ratio: plan alternatives

Figure 13: **Performance during incremental re-optimization of TPC-H Q10 — changes to a base scan relation cardinality estimate**



(a) Execution time (normalized to top-down style) (b) Update ratio: plan table entries (c) Update ratio: plan alternatives

Figure 14: **Performance during incremental re-optimization of TPC-H Q8Join — changes to a base scan relation cardinality estimate**