Shoal: Query Optimization and Operator Placement for Access Controlled Stream Processing Systems

Cory Thoma, Alexandros Labrinidis, and Adam J. Lee

Department of Computer Science, University of Pittsburgh
{corythoma, labrinid, adamlee}@cs.pitt.edu

Abstract. Distributed Data Stream Processing Systems (DDSPS) execute on transient data flowing through long-running, continuous, streaming queries, grouped together in query networks. Often, these continuous queries are outsourced by the querier to third-party computing platforms to help control the cost and maintenance associated with owning and operating such systems. Such outsourcing, however, may be contradictory to a data provider's access controls as they may not permit their data to be viewed or accessed by an unintended third party. A data provider's access controls may, therefore, prevent a querier from fully outsourcing their query. Current research in this space has provided alternative access control techniques that involve computation-enabling encryption techniques, specialized hardware, or specialized query operators that allow for a data provider to enforce access controls while still allowing a querier to employ a third-party system. However, no system considers access controls and their enforcement as part of the *query optimization step*. In this paper, we present Shoal, an optimizer that considers access controls as first class citizens when optimizing and distributing a network of query operators. We show that Shoal can generate more efficient queries versus the state-of-the-art, as well as detail how changes in access controls can generate new query plans *at runtime*.

1 Introduction

The ever-increasing and ever-changing size, speed, and availability of accessible data has led to the rise of new outsourced data processing paradigms. One such paradigm is Data Stream Processing handled by *Distributed Data Stream Processing Systems* (DDSPSs). A DDSPS handles data on-the-fly by executing on transient data with long-running continuous computations (queries), such as streaming operations, map-reduce functions, or user-defined functions, etc. These computations are often outsourced to third-party systems that handle data processing and execution.

Outsourcing computation is desirable for the querier as it provides them with cost savings. For instance, the querier need not maintain expensive hardware and software platforms. Further, the cloud provider offers guarantees on uptime and service availability that a querier can rely on. Finally, the querier can take advantage of a cloud provider's ability to scale to meet demand, by allocating new resources or freeing up underused ones. When a querier contracts a third-party cloud service provider, they are often able to optimize their query to take advantage of different third-party offerings and pricing models. In doing so, they are able to improve the efficiency of their query for

some measurable metric (e.g., latency, throughput, monetary cost) by taking advantage of location, current pricing, current load, and other factors *at runtime* by changing the placement of certain components (i.e., operators) of their queries. This allows queriers to freely move queries around to improve some aspect of the query's performance.

When a data provider dictates access controls over their streaming data, however, a querier may lose some of these freedoms. For instance, if a data provider authors an access control policy that removes a third-party altogether, the querier would lose the ability to execute any part of their streaming query on that provider. Similarly, if access controls are enforced using some cryptographic method (e.g., Polystream [31] or Streamforce [3]), the querier may experience degraded performance as different permissions require different encryption schemes, each incurring different overheads. In both cases, a querier stands to lose some of the benefits of hosting their queries on a third-party system, and may even be required to host the query themselves.

Queriers must consider access controls when trying to generate and optimize their queries. For instance, when a query is broken down into different operators, some operators may be able to execute *directly* on plaintext, but may have to execute on a querier-maintained machine, whereas others may be more costly (monetarily, in terms of latency, or otherwise) but can be executed on a third-party system. A querier must now be able to reason about and decide which implementation of an operation to choose given the accesses they have been provided. This implies that a querier must be able to enumerate potential operators and consider them *at query optimization time* to ensure that the most efficient query plan is derived given each data providers' access controls. Further, when a data provider changes their access controls, query networks may need to be updated.

Currently, DDSPS optimizers and related work have explored DDSPSs optimization in a limited scope. Some have simply focused on better utilization of the underlying computation hardware alone [17], while others have focused on the underlying network alone [7, 13, 27, 29]. Several optimizers and systems have focused on the impact of data variability on the system in the presence of access controls [3, 31] and in the impact of data stream rates and selectivities [2, 6]. Currently, there is no system that focuses on optimizing queries based on the underlying access controls from different data-providers. Further, the closest related work focuses on the use of an *optimize-then-place* approach in which a user's query is first optimized for non-distributed execution and then post-processed for placement on distributed resources. Finally, related work with enforcing access controls in a DDSPS focuses on a single query, which may not suit a querier as they may query many data providers.

We present an optimizer that considers a querier's access privileges at optimization time to produce a high-quality *placement* and *ordering* of individual streaming operators. Our proposal, *Shoal*,[1] uses a dynamic programming algorithm to guarantee optimal placement and orderings for moderate-sized sets of streaming queries on a DDSPS, and includes a heuristic approach for larger query networks. Shoal combines the ordering and placement steps to take advantage of the underlying system by considering multiple

[1] A shoal is a heterogeneous group of fish that is organized to function towards a common purpose, typically of a safety or social nature.

orderings on various distributed computation infrastructures, and avoids the pitfalls of the optimize-then-place approach. To this end, we make the following contributions:

- We show the optimize-then-place approach to be a sub-optimal approach for computing operator placement in a DDSPS.
- We introduce the first cost model for distributed streaming queries that leverages parallelism inherent to the DDSPS and accounts for key sources of heterogeneity such as fluctuations and changes in the underlying data streams and the underlying system and network.
- We detail an optimization algorithm that can execute both at query initialization as well as when a change in the system is detected. This algorithm only optimizes the parts of any queries that are potentially affected by a change in access controls. By only considering parts that are affected by system changes, this online algorithm is able to quickly re-optimize and recover while maintaining an optimal solution.
- We run an extensive evaluation of our algorithms and compare to several baseline, state-of-the-art, and optimize-then-place algorithms. We show that our proposed framework can produce higher quality optimization and placement plans (up to 2.2x better) with reasonably low overheads, and further show these plans are of a higher quality when compared to related work.

The remainder of this paper is organized as follows. We present the system model in Section 2 and formalize our problem statement in Section 3. We describe our proposed approach in Section 4 and present results form our experimental evaluation in Section 5. Section 6 summarizes related work. We conclude in Section 7

2 Background and System Model

This section overview the features Shoal considers when optimizing. We further detail our system model and overview access control frameworks and their affect on a DDSPS.

2.1 Background on DDSPSs

Shoal uses a common Distributed Data Stream Processing System (DDSPS) model. DDSPSs separate the data provider from the data consumer, and often separate the data processing machines as well. DDSPSs rely on long running, continuous computations that execute on transient data. Once a DDSPS has processed the transient data, results can be stored or forgotten depending on the computation being done.

DDSPSs can implement many different stream-processing paradigms such as relational data stream processing [2, 4, 5], MapReduce [19], and user-defined tasks. Relational data stream processing systems use continuous SQL-like queries that are comprised of streaming operations that execute a single task. Similar to traditional database management systems, some operations require large amounts of information to produce their result (e.g., a join or aggregation). Since data is transient in a DDSPS, data is grouped by *windows* and *slides*: a window represents how much data to keep (e.g., 100 tuples, 10 minutes, etc.) and a slide represents how often to update the querier (e.g., every 10 seconds, every 500 tuples, etc.).

The system components of a typical DDSPS are listed below.

- **Data Providers** provide streaming data (subject to access controls) to the system.
- **Sites** are third-party computational and storage platforms (such as Amazon EC2 or Microsoft Azure). They are tasked with the execution of streaming operations as well as forwarding data.
- **Data Consumers** author and submit streaming queries to the system. These queries can be of a variety of paradigms such as relational streaming continuous queries, map-reduce computations, or user-defined functions.

2.2 Access Controls

In a DDSPS, access control enforcement becomes difficult since the data providers can not control the propagation of their data after it is transmitted. The current literature on enforcing access controls in a DSMS can be grouped into two categories: trusted third-party enforcement and untrusted enforcement. Trusted third-party enforcement techniques work by trusting that a computational site will enforce access to their data on their behalf. Such systems either work with special operators [9–11] or by re-writing queries [20, 23] so that access can be limited.

Systems that do not trust third-party enforcement will rely on cryptographically enforced access controls. Rather than forcing a querier to process data only after it has been decrypted, systems like PolyStream [31] and Streamforce [3] allow the data provider to use specialized computation-enabling encryption techniques to enable third-party computation for a querier *directly on encrypted data*. These systems, however, limit the expressiveness and accessibility of a queriers' potential query. In Streamforce, a querier may only access *integer* data via a *view-like* format, (i.e., only allowing filtering and aggregations on numeric data). PolyStream supports a richer set of query operations than Streamforce, but cannot support join or complex user-defined functions over streams from multiple providers. Furthermore, these systems also leak information about the underlying plaintext values, such as equality, relative partial ordering, or relationships between groups of tuples (e.g., the encrypted aggregate of some encrypted data). In either the fully-trusted third-party or the untrusted third-party scenario, access control enforcement comes with computational overheads that must be properly accounted for when optimizing and placing a query.

3 Problem Description

In this section, we detail the exact optimization problem addressed by our framework and define the different components of a Distributed Data Stream Processing System (DDSPS). We offer a description of our optimization approach and show the optimize-then-place approach to be suboptimal.

3.1 Problem Description

In order to properly define the problem being addressed here, we must first formalize the required components. There are three main components to optimizing and placing a data consumer's computation: *sites*, *operations*, *queries*, and *query networks*.

Definition 1. ***Site****: As introduced in Section 2, a site s executes operators. Interconnections between sites have* bandwidth *(in bits/s, where tuples can be of varying size) and* latency *(in ms) characteristics that we represent as $b(s_1, s_2)$ and $l(s_1, s_2)$ respectively. Sites are associated with the following properties:*

- *$s.cap$ is the site's processing capacity (in cycles, translated to $tuples/s$).*
- *$s.name$ is the site's name used for unique identification purposes.*
- *$s.per$ is a set of permissions $\{< o, f >|$ site s can execute a physical operator o on field $f\}$.*

Definition 2. ***Operation:*** *An operation op is a set of operators that execute the same task via different physical implementations.*

- *$op.type$ represents the action to be performed on a data stream (e.g., filter, projection, summation, top-k, etc.).*
- *$op.args$ includes metadata about the operations such as the join condition or selection criteria.*
- *$op.input$ represents the set of fields required for this operation to execute.*
- *$op.output$ represents the set of fields in the output of this operation.*
- *$op.id$ is a unique identifier for the operation.*

A typical operation can be a filter over someone's age, a join to match two streams, or an aggregation to find the maximum profit in a given window of time. Operations can be implemented using different techniques, represented as operators.

Definition 3. ***Operator:*** *The basic computational unit used in Shoal is an* operator *o, which has the following properties:*

- *$o.s$ represents the expected or actual* selectivity *of the operation. Selectivities can be derived either by estimation, measurements during a warm-up period, or historical selectivity data.*
- *$o.c$ represents the* cost *of the operation in terms of the latency for computing on one tuple. It can be calculated in a manner similar to the selectivities.*
- *$o.site$ represents the* site *an operator has been assigned to.*
- *$o.opId$ represents the ID of the operation (that is to say, the logical operator that this physical operator represents) that this operator implements.*
- *$o.window$ represents the window size for a stateful operator either in tuples or ms, with a default of 0.*

Operators allow flexibility when implementing an operation. Consider the potential difference between a hash-join implementation of a join operation versus a merge-join implementation. Given the input rate and selectivity of each stream, it is highly likely that one join would outperform the other in terms of overall latency. An operation would have the merge-join and hash-join as potential operators, and each operator would have a cost that can be used to better optimize the query network.

Definition 4. ***Query:*** *A query is represented as a set q of operations that describe the query or task that a data consumer wishes to execute over a set of data streams.*

- *leaves*(q) *returns the set of operations that operate on a raw data stream (i.e., do not require the output of another operation to execute).*

Definition 5. ***Query Network:*** *A query network is represented as a set qn of queries that will execute within the DSMS.*

- *sinks*(qn) *returns the set of operations that return a result to a querier (i.e., the last part of any one query).*

Using a query, permissions, and a set of available sites as input, Shoal produces a *plan* as output:

Definition 6. ***Plan:*** *A plan* $p = (V_o, E_o)$ *where* V_o *is a set of physical operators and* $E_o \subset V_o \times V_o$ *is the edge set linking the outputs of one operator to the inputs of adjacent operators.*

Definition 7. ***Satisfiability:*** *A plan p* satisfies *a query network qn if:*

- $\forall op \in qn, \exists!\ o \in V_o$ *s.t.* $o.opId = op.id$
- $\forall o \in V_o, \exists!\ op \in qn$ *s.t.* $op.id = o.opId \wedge < o, o.metadata > \in o.site.per$
- $\forall o \in p, o.input \subseteq \bigcup_{o' \mid < o', o > \in E_o} o'.output$

That is, each operation in each query that comprises the query network has a unique operator in the plan and that each operator in the plan is the implementation of one operation in the query, and that each operation in the query is represented in the resulting network. Additionally, each operator's input must be part of the output of its immediate predecessor, and each operator must be permitted to execute on its assigned site.

To determine the relative quality of a given plan p, we use the following cost model.

Definition 8. ***Cost:*** *For a plan p of a Query Network qn, the cost of p, starting at the leaf node(s), is determined by:*

$$\max_{path \in \mathsf{Paths}(p)} \mathsf{pathCost}(path) \tag{1}$$

where Paths is the set of paths from leaf nodes to sink nodes. The expected input rate for each operator (starting from the initial input rate of the leaf node from the source stream) is:

$$\mathsf{ir}(o_i) = IR_{qn} * \prod_{op_j \in \mathsf{pathUpTo}(op_i)} op_j.s \tag{2}$$

The function pathUpdTo(o) for an operator o is the ordered subset of operators that precede o in the plan (as part of the same query). IR_{qn} is the maximum input rate of any leaf operator on the current path. The cost of a path is:

$$\mathsf{pathCost}(path) = \sum_{op_i \in path} \max(op_i.c, op_i.window) + \mathsf{Latency}(op_i, path) * \mathsf{Penalty}(o_i) \tag{3}$$

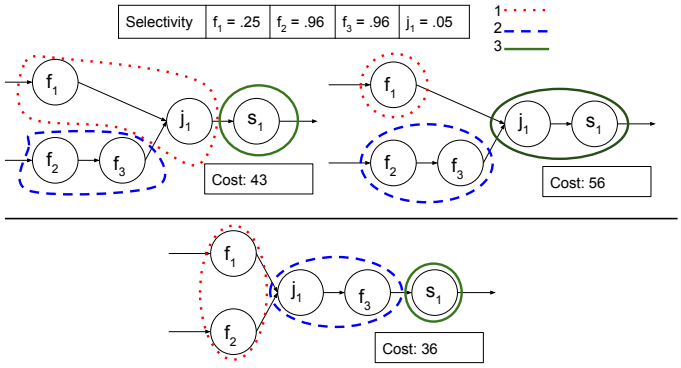


Fig. 1: Simple continuous query.

The penalty is defined as:

$$\text{Penalty}(o) = \begin{cases} 1, & \text{if } \text{pr}(o, o.site) > \text{ir}(o_i) \\ \frac{\text{ir}(o)}{\text{pr}(o,o.site)}, & \text{otherwise} \end{cases} \tag{4}$$

The function $\text{pr}(operator, site)$ determines what the processing rate of a site would be with the operator o assigned to it. If this processing rate is greater than the input rate, then there is no penalty. If the processing rate is insufficient to handle the input rate, the plan is penalized by the input rate over the processing rate. Latency is computed as:

$$Latency(o, path) = \begin{cases} 0, & \text{for } o \in leaves(qn) \\ \text{l}(op, op_{i-1}), & \text{for } op_{i>1} \end{cases} \tag{5}$$

Constrained by $s.cap$ and $bandwidth(op_i, op_{i-1})$.

Definition 9. ***Problem:*** *Given a query network qn of queries, a set of Sites s, and a set of access control permissions per, produce a plan p that satisfies qn such that Equation 1 is minimized.*

3.2 Optimize-then-place Approach

A reasonable first step solution to this optimization problem would be to separate optimization from operator placement. This would allow a placement algorithm to simply use an existing off-the-shelf optimizer and post-process the result for placement. This approach, however, can lead to a sub-optimal plan even if the query itself is fully optimized. Consider the following scenario for a continuous query optimizer, which we will use throughout the remainder of the paper to illustrate Shoal:

A simple query contains three filters (f_1,f_2,f_3), a join (j_1), and a projection (s_1) as depicted in the top of Figure 1 as the result of an optimization step. Next consider the three sites available for placement, and assume the network cost is uniform. Each site has a capacity of 10 (unit-less for simplicity). The cost of each filter is 4, of the join 6, and of the project 2; each have selectivities shown in Figure 1. Given these costs, either the join must be co-located with f_1 (the top of Figure 1) or separated from all

Algorithm 1 DynamicProgramming

```
1: DynamicProgramming(sc, perms, sites)
2: optPlace= new Array(ArrayList(plan))
3: for leaf ∈ leaves(sc) do
4:   optPlace[0] = ∅                                   ▷ Initialize Empty list at level 0
5:   for s ∈ sites do
6:     for l ∈ operators(leaf) do
7:       if s.cap ≥ l.c && (< l,l.metadata.field >∈ s.perms) then
8:         l.site = s
9:         optPlace[0].add(new plan(l))                ▷ Capacity kept per plan
10:  prunePlans(optPlace[0])
11: for lv = 1...|sc| do
12:   optPlace[lv] = ∅                                 ▷ Initialize Empty list at level lv
13:   for operation ∈ sc | operation.type = join do
14:     for plan1, plan2 ∈ optPlace[lv − 1] | (operation.input ⊆ (plan1.output ∩ plan2.output)) ∧ operation.opId ∉
    plan1.opIds ∧ operation.opId ∉ plan2.opIds do
15:       for s ∈ sites do
16:         for join ∈ allowableOps(operation) do
17:           if (updateCapacity(plan1, plan2, s) ≥ join.c) ∧ (< join, join.metadata.field >∈ s.perms) then
18:             join.site = s
19:             optPlace[lv].add(joinPlans(plan1, plan2, join))
20:   for plan ∈ optPlace[lv − 1] do
21:     for operation ∈ sc | (operation.type ≠ join)∧(operation.opId ∉ plan.opIds) ∧ (operation.input ⊆ plan.output)
    do
22:       for s ∈ sites do
23:         for o ∈ allowableOps(operation) do
24:           if plan.s.cap ≥ o.c∧ < o, o.metadata.field >∈ s.perms then
25:             o.site = s
26:             optPlace[lv].add(combine(plan, o))
27:   prunePlans(optPlace[lv])
```

of f_1, f_2 and f_3, (the middle of Figure 1). However, notice that the selectivity of f_1 combined with f_2 is .92, meaning that a tremendous amount of data is being sent over the network to the join. The selectivity of the join, however, is far lower at .05, meaning that a smaller amount of data is being produced. If the query was instead optimized so that f_3 were to follow the join, the overall network cost would be substantially reduced, resulting in a higher quality plan (78.8ms vs. 67.8ms with Equation 1). This illustrates the need for the optimization and placement steps to be considered simultaneously.

4 The Shoal Optimizer

In this section, we introduce our optimization and placement algorithms.

4.1 Online Optimization Approach

Given the long-running nature of continuous queries, there is a high chance (essentially a certainty) that a data provider's access controls will change over time, requiring re-optimization of the network of streaming queries currently deployed. When access controls change for any *one* data streaming operator, it could possibly have a ripple effect for other downstream operators as they may need to be moved to reallocate resources. To accommodate these changes, there are two possible approaches: *stop-the-world* and *on-the-fly*. The stop-the-world approach simply halts query execution and uses Algorithm 1 to re-optimize the query from the root nodes. This approach, however, can lead

Algorithm 2 Access Control first-impacted identifier.

```
1: ACUpdate(Update u, Plan p)
2: cld = sc.leaf                                   ▷ Operations not processed
3: for o ∈ cld do
4:   if o.input | u.protectedFields then
5:     return p.levelOf(o)
6:   else
7:     cld.add(p.childrenOf(o)
8:   cld.remove(o)
```

to large re-optimization times for larger query networks, and can end up doing repetitive work when a relatively small set of operations are affected by the change.

We introduce an *on-the-fly* approach to mitigate these overheads. The principle behind our on-the-fly approach is to execute Algorithm 1 from the operator that is first affected by the access control update relative to the data providers of the overall query network, which we will call the *first-impacted*. This requires the ability to determine the first-impacted, which depends upon the type of access control update that occurred.

Access Control Algorithm 2 determines which operators are first-impacted by a change in access controls. It starts by adding operations that directly access raw data streams on Line 2 to the current query network, *cld*. These operators are then looped through on Line 3, and Line 4 determines if that operator accesses the data being protected by the new access control update. If so, this operation is the first-impacted and the algorithm determines its level by asking the plan for the level. If the operation does not access the protected data, its children are added to the *cld* set, and it removes itself from this set. This continues until the first-impacted is found. At this point, Algorithm 1 will execute on the *descendant* children of the first-impacted, as well as all operations at the same level and their descendants. Note that on Lines 16 and 23, we check for all allowable operations. Recall that one physical operation can be implemented by many physical operations (e.g., the querier may have sufficient access to query in plaintext local to their machine and further have access to use a computation-enabling encryption scheme such as an order preserving scheme on encrypted data in the cloud. This function enumerates the possible operators based on the current permissions of the querier. This leaves the already optimized operations and their ancestor operations intact from the previous plan, and re-optimizes the operations at and after the first-impacted's level, leading to less optimization time. The only alteration required for Algorithm 1 is the inclusion of the current plan from which to start, which is simply placed in the *optimalPlans* set and the Algorithm starts from Line 11 where the level is determined by traversing back to the leaf nodes.

4.2 Greedy & Hybrid Approaches

As with traditional dynamic programming optimizers, our algorithm could suffer from prohibitively large execution times for large or complicated query networks (explored further in Section 5). When query networks become too large or complex, we defer to a greedy approach. This approach simply considers one operator at a time and optimally

places it. In the base case where each operator needs to be placed, the user defines a time threshold $t_{offline}$ for their optimization step. If the dynamic programming approach is expected to exceed $t_{offline}$, then the greedy approach is used. The online approach poses a different problem because there may be uncertainty in how costly an update may be to the system (i.e., the number of operators that need to be re-optimized).

The larger the number of operators that need to be considered, the greater the number of operators requiring re-optimization, and therefore the greater the cost of the update. In a system operating at or near capacity, online updates may end up hindering the quality of the result as some information may be lost during optimization, especially for costly updates on large overall query networks. To combat this problem, we use the greedy approach when updates are too costly relative to the system load. The greedy approach simply re-optimizes, placing each operator in the most optimal location, in a quick but likely non-optimal fashion.

The greedy approach lends itself nicely to distributed systems with heavy load where re-optimization needs to be quick to avoid losing data, but it will not produce plans of the same quality as the dynamic programming approach. To help a data consumer determine which to use, we propose a *hybrid* solution which automatically determines which approach to use given the current system state. The determination is based on three factors relative to the overall streaming query network submitted by the data provider: (a) buffer capacity, (b) processing time of a single streaming tuple (end-to-end), and (c) the input rate. When an update is deemed necessary, its cost c in seconds is determined by multiplying the number of operations needing to be re-optimized by the average amount of time to optimize one operator (based on the execution time of optimizing the entire query, or a running average). Then, the following equation is used to determine which algorithm to use:

$$uc = (\sum_{i}^{o \in p} (b_i * t_i)) + ir_o * c \tag{6}$$

where o is the operator in the plan p, b_i is the utilized buffer size of o, t_i is the processing time of o, ir is the input rate. If $c < uc$ then the dynamic programming approach is used, otherwise the greedy approach is used to minimize data loss.

4.3 Example

To help illustrate how Shoal optimizes a set of streaming queries, consider the following continuous query on a data stream that contains tuples with a timestamp, companyName, companyId, and the company profit, as illustrated in Figure 2:

```
SELECT max(avg_profit), companyName
FROM (SELECT companyName, AVG(profit) as avg_profit
    FROM profitStream GROUP BY companyName EVERY 1m UPDATE 15s;)
GROUP BY companyName EVERY 1m UPDATE 15s;
```

This query requires five operations; a max (m), a projection (p), an average (a), and two group-by operations (g_1 and g_2) represented by circles in Figure 2. Assume that

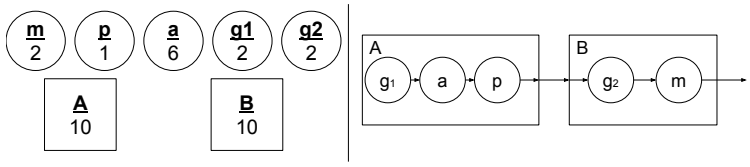


Fig. 2: Given a the set of operations and the sites *A* and *B*, Shoal optimizes and places the operations so that the first aggregation is placed on *A* with the projection reducing network load to the second aggregation operation placed on *B*.

the profit field is protected by a homomorphic encryption and the others are plaintext. Further assume (for simplicity) that there are two sites *A* and *B* with capacities 10 and 10 respectfully, and a latency of 10ms between them (squares in Figure 2).

Shoal starts with the operation *a* as it is the operation that accesses raw data. Since a homomorphic option exists for the aggregation, an operator executing a homomorphic scheme is put onto each site and the next round of dynamic programming is initiated. Further, plans are also added for random encryption and trusted machine processing. This aggregation requires a group-by operation, which can execute on the plaintext column for "company name". This operation is placed with the aggregation on each site, making each site's best plan having a cost of 8 which, along with other plans with varying physical operators, are kept for each site. Shoal then tests the remaining operations and determines that the projection *p* can be added to each site's best plan for a cost of 9. Plans are now kept for each site and for each physical operator, but the minimum plan score is 9. Note that this choice reduces the overall network load by eliminating all columns except the company name and the average. Shoal continues and determines that the maximum operation along with its group-by can not fit on either site and chooses them to operate on site *B* with site *A* keeping its previous plan. With all operators placed, the new plan resembles the one in the right half of Figure 2.

5 Evaluation

To evaluate our optimizers, we decided to use relational continuous queries for the bulk of our experimentation.

Setup: For our evaluation, we limit Shoal to be used in a simple streaming system with data providers, data consumers, and data processing components. For our simulation, data is streamed from a laptop into Amazon AWS EC2 instances. Once data is processed, it is passed back to the laptop to act as the data consumer. We implement the streaming layer on the Apache Storm [30] framework. To keep the streaming layer simple, we use the most basic functionality of Storm where our data provider implements a *spout* and our data processing nodes implement *bolts* with no multi-threating or replication (i.e., a bolt just mimics a machine for our purposes). We use Storm *only* for the transport layer as it guarantees delivery and provides acking and nacking functionality. To simulate real-world streams, each stream is imposed with an artificial latency of 0-30ms to emulate them being geographically separated.

Datasets: We use queries from the TPC-H [25] workload and modify them for use in a streaming system (e.g., aggregations use windows). We will explicitly call out any changes to the query we made, or if we use more than one query as part of the query network. We further segment data based upon a timestamp so that it is streamed into the system (in a pre-processing step) so that days are equivalent to minutes. All queries are referenced using the query number (e.g., q1 for TPC-H query 1) and the number of operators it translates to (e.g., q1(4) is TPC-H query 1, which has four operations).

Baseline Algorithms: In addition to our original and hybrid dynamic programing algorithms, we chose three additional baselines for comparison: (1) *all-on-client*, where all of the operations run on one machine, (2) *first site*, where each operation is placed on the first site available, and switched to the next when either the site is at capacity or there is a conflict with access controls, and (3) *greedy*, where a plan is generated by greedily assigning each operation based on the best score.

5.1 Online Optimizer

This section evaluates our dynamic programing optimizer as compared to other baseline approaches. The cost of an update is based on how many operations in the query network are affected by the update, so we omit cases where the entire network was updated since it would degenerate to the basic case where each operator must be optimized.

Optimization Time Given the cost of an update, this experiment determines the average optimizer execution time for the our dynamic programming approach as well as the baseline approaches.

Configuration: We combine queries in increasing size order (i.e., 1 query, 2 queries, 3 queries, up to 4 queries, or 8, 14, 28, and 45 operators). This provides four data points with an increasing size and number of sinks. All aggregation and join operations are given windows of 5 minutes (to directly use the date field in each relevant tuple). We trigger updates so that only a certain number of operators in each query are affected by the update. Each optimizer is then used to order and place the subsequent operations.

Results (Figure 3): Here we can see the dynamic programming approach is the slowest. This optimizer execution time included the time to determine the first-impacted for each approach, for each query.

Takeaway: Although Shoal has the highest optimization time, it is still relatively low, especially when executing on a long-running continuous query in a network where the resulting plan quality is much more important.

Plan Quality This experiment evaluates the overall plan quality of each approach in terms of latency (ms) for each updated plans. Again, we present both the expected latency, as well as the actual latency. Here, we include the hybrid approach to show when it may switch optimizers to reduce the overall impact of an update.

Configuration: Queries are executed for 10 minutes in total. There is a two-minute window for the initial query, after which an access control update is presented. The query is then updated and the remainder of the time is spent monitoring the updated query. The results presented below are the quality (latency in ms) of the updated query network, as presented by the number of operators updated in the largest network.

Results (Figures 5a and 5b): Our dynamic programming optimizer produces the best overall latencies for both expected and actual evaluations for the query network. The

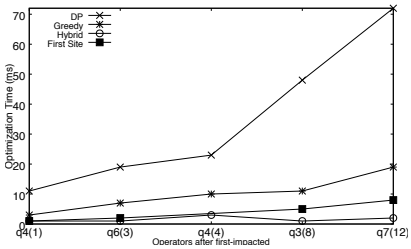


Fig. 3: Optimizer execution time for the online algorithm approaches.

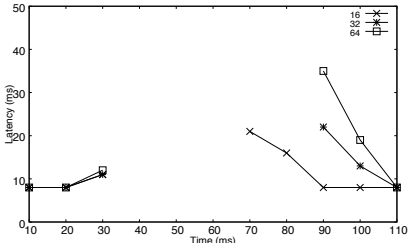
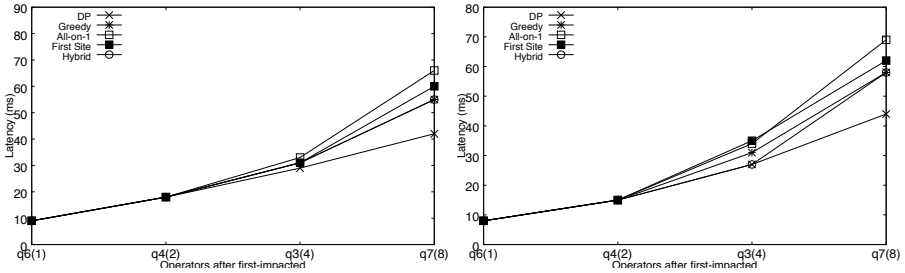


Fig. 4: Recovery time caused by an access control update for different costs.

difference between the expected and the actual is roughly 10.2%, which indicates that Shoal can produce results that are close to the actual values. Notice that the hybrid approach chose to switch to the Greedy optimizer in the last update to the query network. This is due to the system being near capacity when the update occurred (roughly 2,500 tuples/s with a processing rate of roughly 2,615 tuples/s), and in the time to process a new query, the system could have lost data, so the hybrid algorithm chose to use the greedy optimizer.

Takeaway: Shoal produces higher quality plans when compared to the baseline.



(a) Expected latency.

(b) Actual latency.

Fig. 5: Expected and Actual latency for Shoal on random data.

Recovery Time When an update occurs, the system must determine how to re-optimize from first-impacted operation. This process takes time, and while it is processing, the query will still need to be executing. The time between the start of an update optimization and the normal execution of the resulting plan is the time it takes the system to recover from an update. In this experiment, we evaluate this *recovery time* for access control updates.

Configuration: For this experiment, we generated a 128-operator query network. Operations were selected from a random distribution of operations which included two-way joins, filters, summations, averages, projections, and decrypt-process-encrypt operations. Access control updates occur by specifying a specific change in access controls

that target a specific operator such that the update cost remains consistent across the evaluation, and each update causes an increase in latency and a decrease in throughput (i.e., switch from plaintext to encrypted). A query is considered *recovered* once the latency has normalized back to a steady value.

Results (Figure 4): When an update occurs to an access control policy (Figure 4), the data consumer may lose access as indicated by the unreported latency values. Once the query has been resumed, the larger updates cause a large spike in latency that takes more time to recover from, as expected. Note the processing time of each update also increases, but the recovery time is more-or-less the same (20-30ms). This shows that the new queries can handle the increased workload to make-up for the lost work and then maintain a new latency rather quickly.

5.2 Comparison to the State-of-the-Art

We now evaluate the quality of the plans produced by Shoal versus other operator placement approaches, namely Pietzuch et al. [24] and Srivastava et al. [29].

Algorithms: Pietzuch et al. [24] propose a solution that focuses on placing operators in a large-scale distributed network using a latency metric. Their optimizer takes a query plan and places it using a two-step algorithm: first a Virtual Operator Placement step and then a Physical Operator Placement step. The virtual operator placement step considers all operators in a query and places them based on a cost space. This cost space consists of a decentralized view of the network from a single node's perspective and focuses on the latencies between potential sites. There is also a load dimension that can ensure that a single site does not become overwhelmed. Their approach allows for access control updates by allowing operators to migrate between sites. To compare to our work, we fix the cost space by artificially creating latencies and data rates between potential sites (i.e., the assumed information gathered by the DDSMS in their work) and then allow it to adjust over time. The main optimization function used in their work is to minimize the following formula:

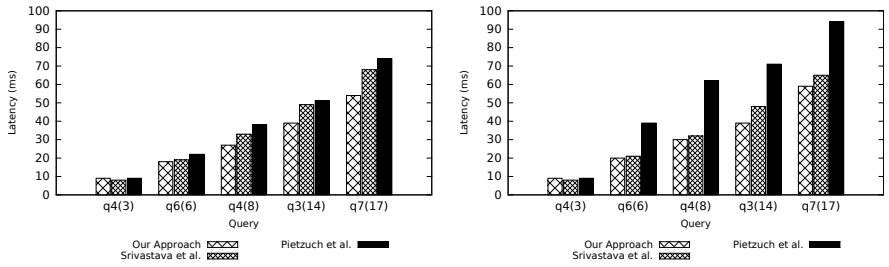
$$\sum_{l \in L} DR(l) * LAT(l) \tag{7}$$

Where l is the link between two nodes, $DR(l)$ is the data rate of that link, and $LAT(l)$ is the latency of that link.

Srivastava et al. [29] also reduce data transmission, but do so for localized networks. Their work focuses on using parts of the query itself, as well as the machines available for placement, to make a placement decision. Specifically, they focus on the selectivity of filtering operations, the cost associated with each operation, and the cost associated with sending a tuple through the network. In addition to the above costs, a join's cost is calculated using its selectivity and the cost per unit time for processing one tuple. The cost of a placement plan is therefore the sum of all of the nodes where the selectivities of upstream filters are multiplied by the cost of the current filter. Some filters are correlated and some are not, so the ordering decision comes from the commutative aspect and the overall cost comes from minimizing the cost of the filter and join orderings. To compare with our work, we again assume an artificially created latency and use the same operators' costs and latencies across all approaches.

Configuration: For our comparison, we use multiple queries over a fixed number of sites. We use 5 sites, each connected to each other with an initial latency randomly selected from a range of 5-500 ms. Each query is comprised of between 4 and 128 operations selected as either filters (selection operations) or joins, with plaintext data. Since [24] requires an initial query plan, we use Shoal with a single site and sufficient capacity to generate a non-distributed query plan. Finally, each filter is given a selectivity randomly selected from the set $\{.1,.2,...,.9\}$. To gather information on actual latencies, each query was executed for a total of five minutes for each approach.

Results (Figures 6a & 6b): As depicted in Figures 6a and 6b, Shoal produces plans with better expected and actual latency. As before, the expected and actual are within an average of 8%, however the Pietzuch et al. approach is more predictable since its expected is on average only 4% different from the actual value. Shoal is able to outperform the other approaches because it attempts to find an optimal solution that takes into account the parallelism inherent to a distributed system by preferring plans that allow work to be done on multiple devices simultaneously. The Pietzuch et al. approach relies on an optimize-then-place approach and missed better filter orderings, which becomes more apparent as queries grow larger. The Srivastava et al. approach does consider ordering, but does not consider the parallelism inherent to a distributed system and would often serialize sets of operations that could have otherwise been done in parallel.



(a) Expected latency.

(b) Actual latency.

Fig. 6: Expected and Actual latency for Shoal on random data.

Takeaway: By considering ordering and placement at optimization time, as well as taking advantage of parallelization inherent to the distributed system, Shoal can outpreform other state-of-the-art optimizers in terms of end-to-end latency.

6 Related Work

Stream processing has been rigorously studied in the literature to include novel systems such as Aurora [1], Borealis [2], and Twitter Herron [18]. For traditional database applications, the focus for operator placement in distributed database systems usually focuses on replication, sharding, or scalability [12,14,15,28]. The PAQO [16] optimizer focuses on placing operators in a distributed database system so that one entity does not

learn the underlying intension of the query. For data steaming systems, operator placement is of a larger concern since queries are long-running and operators are expected to consume resources for long periods of time while possibly fluctuating in their required resource utilization. The contributions in [8] explore the general problem of operator placement on heterogeneous computational platforms for DDSMs, and propose a linear programming model to place operators. Their approach processes placement in a separate step from optimization, which can lead to suboptimal results (cf. Section 3).

Huang et al. [17] fit operators onto sites by calculating the execution time of an operation and place it based on the capacity of each site, using end-to-end delay and throughput as the metrics. Thoma et al. [32] place operators in a DDSMS where queriers have the ability to control where operators are placed via a set of constraints. These constraints generally cover all aspects of the placement, but do not consider the access control policies of a data consumer. Operators placement using heuristics to optimize for end-to-end latency and network traffic have also been explored [7, 13, 27].

Finally, some related work has focused on the impact of enforcing access controls in a DDSPS. Enforcement systems such as FENCE [21, 22] include the enforcement overheads in the optimization step by adding streaming operations that can be handled like any other operation, but do so without considering operator placement. Other systems will rewrite queries or alter streaming operators [9–11, 23], while others focus on protecting a single system, such as Borealis [20]. These systems simply explore the overheads associated with access control enforcement and do not consider them at optimization time or during operator placement. Furthermore, these systems do not explore the tradeoff between different types of access control enforcement during optimization time, which is provided in Shoal. Systems like PolyStream [31], and Streamforce [3], CryptDB [26] consider such tradeoffs, but do either do not operate in a distributed fashion (CryptDB), or do not consider them at optimization time.

Thus far current optimizers and systems have focused on a limited scope of characteristics within a DDSPS, mostly excluding access controls. Either they do not consider optimization and placement simultaneously, or they limit their approach to optimize solely for something like network, hardware, or other traditional metrics. Shoal provides a general cost model and dynamic programming algorithm that accounts for data provider's access control enforcement at query optimization time.

7 Conclusion

We present Shoal, which considers access controls as first-class-citizens during query optimization. By simultaneously ordering and placing streaming query networks on a per-operator level, Shoal can guarantee optimal results through a dynamic programming algorithm. Further, Shoal reduces optimization time for updates based on changes in access controls by identifying the precise operators that need to be re-optimized and only optimizing from those points forward in an online fashion. Finally, we show that Shoal produces higher quality plans (up to 2.2x) versus the state-of-the-art optimizers, and does so while considering data provider's access controls.

Acknowledgements This work was supported in part by the National Science Foundation under awards CNS–1253204 and CNS–1704139.

References

1. D. Abadi et al. Aurora: a new model and architecture for data stream management. *VLDB*, 12(2):120–139, 2003.
2. D. Abadi et al. The design of the borealis stream processing engine. In *CIDR*, 2005.
3. D. T. T. Anh and A. Datta. Streamforce: outsourcing access control enforcement for stream data to the clouds. In *ACM CODASPY*, pages 13–24, 2014.
4. A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. Stream: The stanford data stream management system. *Book chapter*, 2004.
5. A. Arasu et al. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.
6. A. Arasu et al. Stream: The stanford data stream management system. In *Data Stream Management*, pages 317–336. Springer, 2016.
7. N. Backman, R. Fonseca, and U. Çetintemel. Managing parallelism for stream processing in the cloud. In *HOTCDP Workshop*, pages 1–5. ACM, 2012.
8. V. Cardellini et al. Optimal operator placement for distributed stream processing applications. In *DEBS*, pages 69–80. ACM, 2016.
9. B. Carminati et al. Enforcing access control over data streams. In *ACM SACMAT*, pages 21–30, 2007.
10. B. Carminati et al. Specifying access control policies on data streams. In *Advances in Databases: Concepts, Systems and Applications*, pages 410–421. Springer, 2007.
11. B. Carminati et al. A framework to enforce access control over data streams. *ACM TISSEC*, 13(3):28, 2010.
12. R. Cattell. Scalable sql and nosql data stores. *Acm Sigmod Record*, 39(4):12–27, 2011.
13. A. Chatzistergiou and S. D. Viglas. Fast heuristics for near-optimal task allocation in data stream processing over clusters. In *CIKM*, pages 1579–1588. ACM, 2014.
14. J. C. Corbett et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.
15. C. Curino et al. Relational cloud: A database-as-a-service for the cloud. *CIDR*, 2011.
16. N. Farnan et al. Paqo: Preference-aware query optimization for decentralized database systems. In *ICDE*, 2014.
17. Y. Huang et al. Operator placement with qos constraints for distributed stream processing. In *CNSM*, pages 1–7. IEEE, 2011.
18. S. Kulkarni et al. Twitter heron: Stream processing at scale. In *SIGMOD*, pages 239–250. ACM, 2015.
19. K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon. Parallel data processing with mapreduce: a survey. *AcM sIGMoD Record*, 40(4):11–20, 2012.
20. W. Lindner and J. Meier. Securing the borealis data stream engine. In *IEEE IDEAS*, pages 137–147, 2006.
21. R. Nehme et al. A security punctuation framework for enforcing access control on streaming data. In *ICDE*, pages 406–415, 2008.
22. R. V. Nehme et al. Fence: Continuous access control enforcement in dynamic data stream environments. In *ACM CODASPY*, pages 243–254, 2013.
23. W. S. Ng et al. Privacy preservation in streaming data collection. In *ICPADS*, pages 810–815, 2012.
24. P. Pietzuch et al. Network-aware operator placement for stream-processing systems. In *ICDE*, pages 49–49. IEEE, 2006.
25. M. Poess and C. Floyd. New tpc benchmarks for decision support and web commerce. *ACM Sigmod Record*, 29(4):64–71, 2000.

26. R. Popa et al. Cryptdb: protecting confidentiality with encrypted query processing. In *ACM SOSP*, pages 85–100, 2011.
27. S. Rizou et al. Solving the multi-operator placement problem in large-scale operator networks. In *ICCCN*, pages 1–6. IEEE, 2010.
28. J. Shute et al. F1: A distributed sql database that scales. *VLDB*, 6(11):1068–1079, 2013.
29. U. Srivastava, K. Munagala, and J. Widom. Operator placement for in-network stream query processing. In *SIGMOD*, pages 250–258. ACM, 2005.
30. StormProject. Storm: Distributed and fault-tolerant realtime computation. `http://storm.incubator.apache.org/documentation/Home.html`, 2014.
31. C. Thoma et al. Polystream: Cryptographically enforced access controls for outsourced data stream processing. In *SACMAT*, volume 21, page 12, 2016.
32. C. Thoma, A. Labrinidis, and A. J. Lee. Automated operator placement in distributed data stream management systems subject to user constraints. In *ICDEW*, pages 310–316. IEEE, 2014.