Supporting Distributed Feed-Following Apps over Edge Devices

Badrish Chandramouli[1] Suman Nath[1] Wenchao Zhou[2*]
[1]*Microsoft Research Redmond* [2]*Georgetown University*
{badrishc, sumann}@microsoft.com, wzhou@cs.georgetown.edu

ABSTRACT

In *feed-following* applications such as Twitter and Facebook, users (*consumers*) follow a large number of other users (*producers*) to get personalized feeds, generated by blending producers' feeds. With the proliferation of Cloud-connected smart *edge devices* such as smartphones, producers and consumers of many feed-following applications reside on edge devices and the Cloud. An important design goal of such applications is to minimize communication (and energy) overhead of edge devices. In this paper, we abstract distributed feed-following applications as a view maintenance problem, with the goal of optimally placing the views on edge devices and in the Cloud to minimize communication overhead between edge devices and the Cloud. The view placement problem for general network topology is NP Hard; however, we show that for the special case of Cloud-edge topology, *locally optimal solutions yield a globally optimal view placement solution*. Based on this powerful result, we propose view placement algorithms that are highly efficient, yet provably minimize global network cost. Compared to existing works on feed-following applications, our algorithms are more general—they support views with selection, projection, correlation (join) and arbitrary black-box operators, and can even refer to other views. We have implemented our algorithms within a distributed feed-following architecture over real smartphones and the Cloud. Experiments over real datasets indicate that our algorithms are highly scalable and orders-of-magnitude more efficient than existing strategies for optimal placement. Further, our results show that optimal placements generated by our algorithms are often several factors better than simpler schemes.

1. INTRODUCTION

Feed-following applications [26, 27] such as Facebook, Twitter, iGoogle, and FourSquare have become extremely popular in recent years. In these applications, users "follow" a large number of other users (or entities) and get personalized *feeds* produced by blending streams of events generated by (or associated with the activities of) these users/entities. *Producers* generate time-ordered events of a particular followable interest, such as their locations/check-ins, tweets, facebook status, etc. Feeds from multiple producers are then blended into feeds for *consumers*. A consumer retrieves his feed on demand (e.g., when he visits iGoogle) or gets notified whenever his feed satisfies certain predicates (e.g., when a friend checks in a nearby place).

Figure 1: Architecture of Cloud-Edge applications.

In this paper, we focus on *feed-following Cloud-Edge applications*, that run on Cloud-connected *smart edge devices* (*edge devices* for short) such as smartphones, consumer electronics devices (e.g., Xbox and Google TV), and in-car dashboards (Figure 1). These devices are equipped with sensors that produce, in addition to user generated feeds, digitally born feeds such as GPS location, speed, user activity, and battery usage. Consider a *social Foursquare* application (e.g., *Loopt* app on iPhone, Android, and Windows Phone) that notifies a user whenever any of her friends checks in or visits a nearby location. Enabling this application requires blending fast-changing location feeds from users' smartphones as well as slowly changing reference data such as a social network (defining the friend relationship). The list of existing feed-following Cloud-Edge applications is quite long; examples include Twitter-like applications (e.g., $NanoTwitter$ on Android phones) that notify a smartphone user when any of his friends tweets on a similar topic he has tweeted on, context-aware notifiers (e.g., OnX on Android) that notify a user when, for example, his wife leaves office or his friends gather together for a movie, location-aware coupon services (e.g., $GeoQpons$ on Android and iPhone) that notify a user when she is close to a business offering coupons that she might like, online multiplayer games (e.g., $iMobsters$ on Android and $Halo : Reach$ on Xbox 360) that monitor players' mutual interactions and status, and in-car dashboard apps that provide online route and gas station recommendations.

As discussed in [26], the core functionality of a feed-following application can be formalized as a type of view materialization problem. For example, the final output of the social Foursquare application can be thought of a trigger on a *logical* `nearby_friends` view, consisting of many *physical* `nearby_friends_x_y` views, maintained one per every pair of friends (x, y). The view `nearby_friends_x_y` maintains current distance between two friends `x` and `y` if they are nearby, i.e., if their distance is within a threshold and is empty if the `x` and `y` are not nearby. The views are maintained by a continuous query correlating (or joining) location feeds from various users and their friend relationships. Even

*Work performed during internship at Microsoft Research

Proceedings of the VLDB Endowment, Vol. 6, No. 13

though the views are conceptually simple, typical feed-following applications have high skews in feed update rates and follow-relationship fanout. Moreover, popular streams have high fanout (e.g., many followers). As discussed in [26], these factors make view selection, scheduling, and placement extremely challenging.

Almost all existing feed-following Cloud-Edge applications blend data feeds from various edge devices at the Cloud—data updates from edge devices are periodically sent to the Cloud, which then aggregates data, updates its views, and pushes notifications (or takes other actions). This "views-at-the-Cloud" strategy, while it greatly simplifies the application logic, can be expensive. To illustrate this, suppose Alice is working in her office and her friend Bob is driving in a nearby neighborhood. Then, an existing social Foursquare application will require Bob to upload his location frequently (say, once every 10 seconds) so that the Cloud knows his up-to-date location to correlate with Alice's location in order to maintain the `nearby_friends_Alice_Bob` view. Alice, however, can upload her location infrequently (say, once an hour) since she is not moving much.[1] In this example, the total communication overhead of Alice and Bob will be 361 messages per hour (ignoring final notifications). This can be prohibitively expensive, especially when a user has many friends or runs many such applications. Note that we focus on efficiency only in terms of communication overhead (which is correlated to energy consumption of edge devices) between devices and the Cloud, because network bandwidth and energy are two of the most valuable resources in edge devices.

We observe that the inefficiency can be addressed easily in the above example. Instead of using views-at-the-Cloud, we could send Alice's location to Bob once an hour (via the Cloud), perform the correlation and compute the `nearby_friends_Alice_Bob` view *in Bob's device*. This way, Bob would not need to send his location anywhere and the total cost would become only 2 messages per hour (one from Alice to the Cloud, and one from the Cloud to Bob), compared to 361 messages per hour with the views-at-the-Cloud strategy. Note that depending on where the correlation computation is done, physical views involving Alice maybe be distributed among her device, her friends' devices, and the Cloud.

Challenges. Given the above observations, why do existing Cloud-Edge applications compute/maintain the views at the Cloud instead of distributing them among edge devices? One main reason is that the decision of how to distribute the views (i.e., what computation to push, and to which edge device) is non-trivial. The decision requires solving an optimization problem that depends on various factors such as the network topology, update rates of the data, data upload/download costs, what data sources need to be correlated, etc. Moreover, since some parameters such as feed rates can change over time, the decision needs to be dynamically updated. The complexity of implementing such optimizations often outweighs the cost of developing the core functionality of a Cloud-Edge app.

In this paper, we propose efficient algorithms to distribute views of feed-following applications in a way that provably optimizes communication between edge devices and the Cloud. Unlike previous works [26, 27], we allow feeds to be blended with expensive query operators (e.g., multi-way joins) and general black-box operators. We incorporate the algorithms in Race (for *Real-time feed-following Applications over Cloud-Edge*), a novel architecture and platform that leverages existing database engines to efficiently execute feed-following applications, as continuous subqueries on edge devices and in the Cloud. Race allows specifying feed-following applications as *logical view templates* written in a declarative language; it then automatically instantiates necessary physical views and distributes the views among edge devices and the Cloud to minimize communication costs.

To achieve an optimal partitioning and placement of feed-following views, our algorithms first transform the logical view template of an application into a *view graph*, a directed acyclic graph of computation vertices (views) connected by queues. Such a transformation can also leverage existing optimization strategies such as view/query rewriting and join reordering [10, 12, 13, 15]. Given a view graph, we next pose the decision of which view to run at which location as a view placement problem with a goal of minimizing the total communication overhead. The problem is NP-Hard for general communication graphs [19, 22]. Previous work has shown how to address this in polynomial time for tree-like communication networks (by using a min-cut algorithm) [22]. Our experimental results show that existing algorithms do not scale to graph sizes observed in feed-following applications. For example, for a social Foursquare application with 20,000 users, the min-cut based algorithm in [22] takes 4.75 hours (more details in Section 5). This is clearly impractical for applications with dynamic data rates and churn.

Our solution. Race addresses the above challenges by exploiting two observations. First, even though the general network topology given by edge devices and computing nodes on the cloud can be very complex, edge devices, which significantly outnumber cloud nodes, are connected in a much simpler way. More specifically, edge devices are typically connected to the Cloud only (though WiFi, cellular network, etc.) and they communicate with one another only through the Cloud. This reflects the dominant form of communication between mobile-phones today. Second, since we focus on efficiency only in terms of the communication between edge-devices and the Cloud, we can treat the "Cloud" as one single box (providing a centralized service running on a Cloud platform or at provider's servers).[2] These two observations result in the *star topology* underlying Cloud-Edge applications (see Figure 1).

A key contribution of this paper is to show that for such a star topology, *locally optimal view placement solutions obtained by each device independently using an efficient greedy algorithm yield a globally optimal placement solution*. This powerful result enables Race to find the provably optimal placement in a very efficient, scalable, and possibly distributed way, without using expensive centralized algorithms such as graph partitioning (as is done in [22]) or linear programming (as is done in [19]). Race can handle multi-level composition of views, with cross-view sharing, asymmetric network link costs, and more general view graphs with black-box operators.

Note that even though we treat the Cloud as a black-box in this paper, our techniques can be composed with existing techniques for an arbitrary network topology inside the Cloud. For example, one could first use our efficient technique to optimally place views among edge devices and the Cloud, and then use existing expensive techniques in [22] or [19] to optimally distribute the operators assigned to the Cloud among its compute nodes. Since our technique can efficiently handle all edge devices, which significantly outnumber Cloud nodes that need to be handled by expensive techniques, this two step process is much more scalable and faster than applying expensive techniques on the entire network topology (consisting of all edge devices and Cloud nodes). Interestingly, if the

[1] The rates may depend on other factors such as users' explicit settings, connectivity status, distance between friends, etc.

[2] Optimizing the Cloud to address challenges such as partitioning and balancing computation among multiple servers are orthogonal to this work and part of our future work.

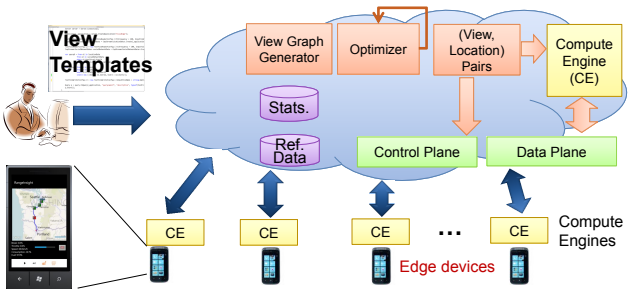


Figure 2: The Race architecture.

placement algorithm for the Cloud is optimal, this two step process yields a globally optimal placement, as we show in Section 4.2.

Several recent works proposed various optimizations for feed-following applications [18, 26, 27]. In [26], authors give an overview of various challenges, including the view placement problem we address in this paper, related to feed-following applications and explain why prior works on pub/sub systems, caching, and materialized views are inadequate to address these challenges. Feeding Frenzy [27] proposes a view selection algorithm to optimally choose between two types of views (consumer-pivoted views and producer-pivoted views); in contrast we consider the view placement problem. All these works assume that final feeds to consumers are produced by applying simple selection and union operations; in contrast, our algorithms are more general—they support views with selection, projection, correlation (join) and arbitrary black-box operators, and can even refer to other views. Finally, unlike prior works, we consider feed-following applications distributed over a large number of edge devices and the Cloud.

We have implemented a full-fledged prototype of Race, running over smartphones and the Cloud. Experiments with real datasets show that Race is scalable and can find optimal solutions orders-of-magnitude faster than current strategies. Further, our placements are several factors better than simpler solutions such as views-at-the-Cloud.

Contributions In summary, we make the following contributions:

- We propose (§ 2) a general view placement paradigm for feed-following apps, and design Race, a platform for Cloud-Edge apps that leverages compute engines on devices and the Cloud to share the computation of logical views for the application.
- We develop (§ 3, 4) efficient algorithms that generate provably optimal view placements for Cloud-Edge apps on Race, while handling network links with asymmetric costs and multi-level view graphs with correlations and black-box operators.
- We implement and evaluate (§ 5) a full-fledged prototype of Race. We report results using the social Foursquare application over realistic social network and location data, demonstrating scalability, efficiency, and utility of Race and its optimizer.

2. Race MODEL AND ARCHITECTURE

Race exposes a set of input data sources to a feed-following application. Inputs can be continuous feeds such as location, checkin, or tweet streams generated by individual edge devices, as well as static (or slow-changing) *reference data* such as social network relationship tables. The application specifies a set of physical views over these inputs. Race supports more expressive views than discussed in [26]; specifically, a view can correlate (join) multiple data sources, and support selects, projects, and black-box operators. Further, views can reference other views as input, allowing multi-level queries over the data sources. For example, a social Foursquare app may wish to compute a logical view `nearby_friends` that consists of all pairs of nearby friends. To compute this logical view, it may generate a set of physical views `nearby_friends_x_y` for each pair of friends `x` and `y`, each of which is placed at some edge device or in the Cloud[3].

Figure 2 shows the overall system architecture of Race. An application may need millions of physical views. Therefore, Race uses logical *view templates* to allow applications to succinctly specify a large number of physical views. A *View Generator* module compiles these logical view templates into a single large *physical view graph*, with physical views as nodes in the graph. The view graph is passed to an *Optimizer* module that computes the optimal placement of views across the Cloud and edge devices. Finally, physical views are shipped to their destination, where a *Compute Engine* performs the actual view computation and maintenance. Race handles the data communication necessary to provide each physical view with all the inputs that it needs (either feeds or other physical views). End subscribers (e.g., Foursquare users) can construct their required logical views by accessing the relevant physical views, as needed by the application. These components are discussed below.

In this paper, we assume that all edge devices are capable of computing views[4]. For simplicity, we limit our discussion to cases where devices are always connected (i.e., no failures) and subscribers require all view updates (no on-demand delivery). These assumptions are relaxed in Sections 4.4 and 4.5 respectively.

2.1 View Graph Generator

Since there can be millions of views needed by an application, Race offers a succinct way to automatically generate them using logical view templates that define views in terms of logical inputs. For example, one may specify a view template V_1 that joins a logical location stream with a logical social network stream. Race detects that the social network input is reference data, and materializes the relationship to generate one physical view for each user-friend pair. Logical views can be composed: one can correlate V_1 with the location stream to get one physical `nearby_friends_x_y` view for every relationship in the social network. Reference data can be slow-changing; Race handles this by periodically updating the set of active physical views in the system.

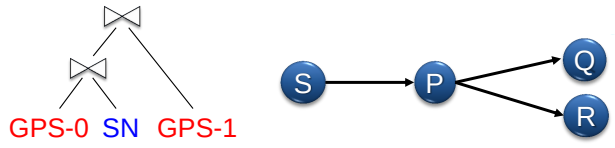
The logical view template represents the template or skeleton for generating the expanded view graph. Figure 3(a) illustrates the view template for our social Foursquare query. Assuming that the social network is as shown in Figure 3(b), Figure 4 shows the corresponding instantiated physical views[5]. Note that in order to allow information sharing and avoid duplicated edges in the graphs, the instantiated sources and physical views are named carefully, as shown in the figure.

The final step is to stitch the instantiated physical views into a complete physical view graph. Figure 5 shows the final view graph derived from the instantiated views from Figure 4. Note that when combining the instantiated graphs, the vertices (in the instantiated patterns) with the same name are mapped to the same vertex in the

[3]The number of physical views is linear in the number of edges in the social network graph

[4]This restriction can be relaxed by adding constraints for specific edge devices; we can still produce feasible placements in this case.

[5]Note that materialization of the social network would actually convert the view template into a set of single-level joins; we leave them as multi-level joins with a partitioned social network input to illustrate how we can leverage opportunities for sharing intermediate view results.



(a) Logical view template (b) Social network of four users

Figure 3: Logical view template and social network.

view graph. For instance, the `Join<GPS-P, SN`$_P$`>` vertex is shared by the instantiated patterns for edges (P, R) and (P, Q).

2.2 Optimizer

The Optimizer module accepts the view graph as input, and decides where to execute each physical view so that the total communication cost of the application is minimized. With thousands or even millions of users participating the Cloud-Edge system, the view graph could be huge – containing millions of physical views. For such a large view graph, the optimal view placement is non-trivial. In Section 3, we describe novel and efficient solutions used within the Race Optimizer module. Race performs periodic re-optimization to adjust the placement to changes in the view graph and/or statistics.

2.3 Compute Engines

Race hosts a *compute engine* on every edge, as well as at the Cloud. The compute engine is responsible for executing the view and keeping it up to date. Given an optimizer decision, Race sends each view to the corresponding compute engine via a *control plane*. The compute engines merge similar views into a single query plan when possible, so that a fewer number of queries are executed at a node. A complication is that edge devices (e.g., phones) are usually not directly reachable or addressable from Race. Instead, Race maintains a server to which the edge devices create and maintain persistent connections in order to receive management commands that are forwarded to the DSMS instances on the edges.

During view maintenance at compute engines, data needs to flow between edge devices and the Cloud. Race uses a *data plane* that is exposed as a server at the Cloud, and to which the edge devices connect. The data plane routes data from the Cloud to devices and vice versa. Note that in the Cloud-Edge topology, the data flow between two devices is indirect (via the Cloud), i.e., devices do not communicate directly with one another.

3. OPTIMAL VIEW PLACEMENT

The view graph for a typical feed-following Cloud-Edge application can be huge, with millions of views[6] in the view graph. Since feed sources are distributed (e.g., user location data originates from edge-devices), the placement of every intermediate view has an impact on view maintenance overhead. As there are exponentially many view placement choices, a naïve approach that searches the whole design space is not feasible. In addition, we consider the sharing of intermediate results, which makes the problem even harder.

3.1 Summary of Theoretical Results

As mentioned earlier, we leverage the "star" topology of Cloud-Edge systems to find the provably optimal placement efficiently. Table 1 summarizes the theoretical results and time complexities

[6]For brevity, when it is clear from the context, we use term "view" as the abbreviation for physical view.

	Condition	**Local Complexity**
Select Conditions	*None*	$O(N)$, N = *# of friends*
	Sampling	$O(N \log N)$, N = *# of friends*
	Condition	**Global Complexity**
Graph Complexity	*Single-level*	*Parallelizable local algorithm*
	Multi-level	*Local algorithm in top-down fashion*
Asymmetric Costs	$C^u \leq C^d$	*Parallelizable local algorithm*
	$C^u > C^d$	*DP with acyclic residual graph*

Table 1: Summary of the view placement algorithm. Global optimality is achieved in all cases.

for our placement algorithm, for various combinations of the complexities of view graphs, select conditions, and upload/download cost ratios. Note that we focus on correlations/joins (with sampling filters) in this section; the extension to more general view graphs with black-box operators is covered in Section 4.

In Section 3.4, we first establish a formal relationship between view placement and the *assignment problem*, which decides whether a vertex in the *demand graph* (defined in Section 3.3) should push subsets of its data to the Cloud.

We first focus on symmetric network links, where the per-unit cost of uploading (C^u) and downloading (C^d) data are equal. In Section 3.5, we cover view graphs with single-level joins, with and without *sampling filters*. Here, an important result is that *locally optimal choices are globally consistent and optimal*. This gives us a parallelizable placement algorithm for optimal placement. We further propose efficient greedy algorithms for computing locally optimal choices in this case. In Section 3.6, we extend our solutions to more complex view graphs with multi-level joins, where we can still derive the provably optimal placement by treating intermediate views in a view graph as virtual vertices, and computing the local optimality for each individual vertex in a top-down fashion.

We cover asymmetric links in Section 3.7, where we observe that our earlier results directly hold when $C^u \leq C^d$. When $C^u > C^d$, we show that in the common case where the *residual graph* (see Section 3.7) is acyclic, there is an efficient dynamic programming (DP) algorithm to find the optimal view placement.

3.2 Assumptions

As is common in content-delivery and publish/subscribe systems [7], we ignore the cost of the "final hop" notification sent to end subscribers, as this is usually negligible compared to the cost of in-network view maintenance. We further make two assumptions:

Assumption 1 *For any correlation/join $A \bowtie B$ (where A and B are the input feeds of the correlation), its resulting view is located either at the Cloud or on the nodes where A or B originated.*

This assumption of restricting possible placement locations of a view is practical due to privacy and efficiency reasons. As the data may include sensitive private information (e.g., location and context), an edge device may want to ship its data only to the devices (e.g., a friend) that register its feed data. Moreover, since computing a view consumes energy and other resources on an edge device, a device may want to maintain only views that involve its data. The assumption does not simplify the placement problem; there are still an exponential number of possible view placements.

Assumption 2 *A feed from a given edge device appears in no more than one subtree of any view in the view graph.*

This is a reasonable assumption, since one can simply combine feeds from the same edge device into a single feed, or locally perform the necessary computation that these feeds are involved in. Note that this assumption does not preclude sharing of source or

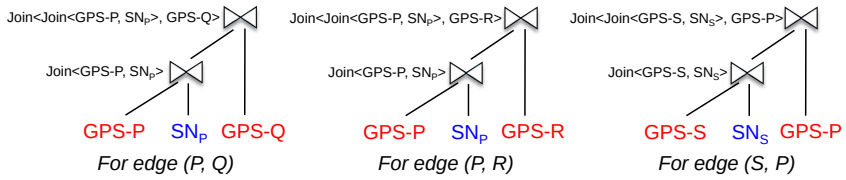


Figure 4: Instantiated physical views.

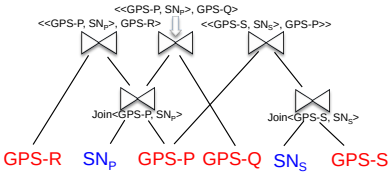


Figure 5: Final physical view graph.

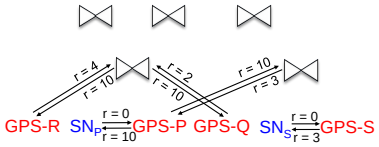


Figure 6: An example demand graph.

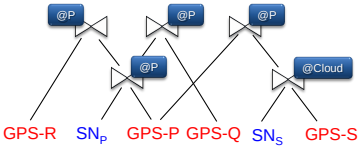


Figure 7: Optimal view placement.

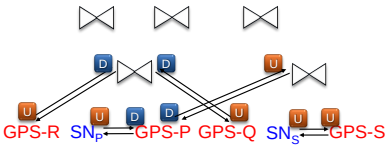


Figure 8: The corresponding assignment.

Algorithm 1 Generate Demand Graph from View Graph

1: **func** $DemandGraph(G^Q = (V^Q, E^Q))$
2: $V^D \leftarrow \phi$; $E^D \leftarrow \phi$
3: **for** $\forall\, v_1 \in V^Q$ **do**
4: $\quad$ *Suppose* $e_1 = (v_2, v_1) \in E^Q$, $e_2 = (v_2', v_1) \in E^Q$
5: $\quad V^D \leftarrow V^D + \{v_1\}$
6: $\quad E^D \leftarrow E^D + \{e_1' = (v_2, v_2'), e_2' = (v_2', v_2)\}$
7: **return** $G^D = (V^D, E^D)$

intermediate results, and in particular, it always holds in case the view template (that defines the local view) is a left-deep tree over different feed sources.

3.3 Demand and Demand Graph

Before presenting our proposed algorithm, we introduce the following graph-based denotations to facilitate the discussion.

Definition (Demand) *We denote, as a pair (v_1, v_2), that a feed data source v_2 "demands" (i.e., needs to correlate with) the data generated by another source v_1.*

Definition (Demand Graph) *Given a feed-following Cloud-Edge application, the demand graph $G = (V, E)$ is defined as follows: the vertex set $V = \{v | v$ is a feed data source $\}$, and $E = \{(v_1, v_2) | (v_1, v_2)$ is a demand pair$\}$. Each edge $e = (i, j) \in E$ is associated with a rate $r_{i,j}$, indicating the rate of v_i's feed that is demanded by v_j.*

Figure 6 shows the corresponding demand graph for the `nearby_friends` logical view, given the social network shown in Figure 3(b). The edges in the graph illustrate the demand relationships, for instance, the edge (GPS-P, SN_P) indicates that the GPS reading from P (GPS-P) needs to be correlated with the social network (SN_P). Each edge is associated with a rate r indicating the update rate of the feed data.

In a demand graph, intermediate views (resulted from join operations) are treated as virtual feed data sources in the demand graph (as they are producing join results as feeds). Actually, there is a one-to-one mapping between demand graphs and view graphs. Given a view graph $G^Q = (V^Q, E^Q)$, Algorithm 1 generates the corresponding demand graph $G^D = (V^D, E^D)$; the view graph can be re-engineered from the demand graph by following a similar algorithm.

3.4 Assignment: Download vs. Upload

In general, deciding optimal view placement for distributed view maintenance is a classic research topic in database community, and is known to be a hard problem. The essence of our proposed algorithm is rooted in leveraging a special network property of the Cloud-Edge architecture: edge devices cannot communicate with each other directly; to exchange information, they have to *upload* or *download* data through the Cloud-side servers.

Definition (Upload and Download) *Given a demand graph $G = (V, E)$, for an edge $(v_i, v_j) \in E$, we say that v_i is "*uploading*" on (v_i, v_j), if, regardless of where v_i is located (either at an edge device or the Cloud server), v_i transmits the corresponding feed (demanded by v_j) to the Cloud server; otherwise, we say that v_i is "*downloading*" on (v_i, v_j).*

Intuitively, once a vertex decides to upload on an edge (which represents a required feed data correlation), there is no reason for it to download any data for this correlation from the Cloud-side server, because the correlation can be simply performed at the Cloud (as the feed data needed for the correlation has been made available at the Cloud).

Given a demand graph $G = (V, E)$, a view placement for G can be mapped to a set of upload vs. download decisions (called an *assignment*) made on each vertex.

Definition (Assignment) *Given a demand graph $G = (V, E)$, an assignment $\mathcal{A} : E \rightarrow \{D, U\}$ is defined as follows: $\mathcal{A}_{v_i,v_j} = U$ if vertex v_i decides to upload its feed data on edge (v_i, v_j), otherwise, $\mathcal{A}_{v_i,v_j} = D$.*

We denote the optimal placement and its corresponding assignment as $\mathcal{P}^{opt}$ and $\mathcal{A}^{opt}$. Figure 7 shows the optimal placement ($\mathcal{P}^{opt}$) for the demand graph of Figure 6, and Figure 8 shows the corresponding assignment ($\mathcal{A}^{opt}$). In the optimal view placement, the resulting view for the correlation between GPS-P and SN_P is performed at node P, which means that the partitioned social network graph SN_P should be shipped to node P, i.e., SN_P is "uploaded" to the Cloud, and GPS-P is not. This is consistent with the assignment given in Figure 8.

It is natural to ask the questions 1) whether there exists a reverse mapping from $\mathcal{A}^{opt}$ to $\mathcal{P}^{opt}$, and 2) whether there exists an efficient algorithm to find $\mathcal{A}^{opt}$ for a given demand graph. We next present our conclusion for first question, and gradually develop the answer for the second question in the rest of this section.

Not all assignments can be mapped to a *viable* evaluation plan. There is a fundamental constraint: join requires the colocation of all its inputs. Therefore, for any join that takes inputs from different feed sources (edge devices), *at most* one device is downloading.

Definition (Viability and Conflict) *Given a demand graph $G = (V, E)$, an assignment $\mathcal{A}$ is* viable *if it satisfies the following con-*

Algorithm 2 Compute Placement from Assignment

```
1: func Placement(G^Q = (V^Q, E^Q), Assign)
2: // Initialize the placement of leaf vertices (i.e., raw sources)
3: Placement ← {}
4: for ∀ v ∈ V^Q do
5:    if !∃ e = (v', v) ∈ E^Q then Placement_v ← v
6: // Determine view placement in a bottom-up fashion
7: TopoOrder ← V^Q sorted by topology sort.
8: for ∀ v ∈ TopoOrder in the bottom − up order do
9:    Suppose e_1 = (v_1, v) ∈ E^Q, e_2 = (v_2, v) ∈ E^Q
10:   if Assign_{e_1} = D then Placement_v ← Placement_{v_1}
11:   else
12:      if Assign_{e_2} = D then Placement_v ← Placement_{v_2}
13:      else Placement_v ← Cloud
14: return Placement
```

dition: $\forall e = (v_i, v_j) \in E$, $\mathcal{A}_{v_i,v_j} \neq D \vee \mathcal{A}_{v_j,v_i} \neq D$. *We call an edge that breaks this condition a* conflict *edge.*

For example, Figure 8 illustrates a viable assignment given the demand graph shown in Figure 6, as for any correlation, at most one data source is deciding to download. If the $\mathcal{A}_{\texttt{SN}_\texttt{P},\texttt{GPS-P}}$ is changed to download, it will invalidate the assignment, as the edge $(\texttt{SN}, \texttt{GPS-C})$ is a conflict edge.

Lemma 1 *Given a viable assignment $\mathcal{A}$, $\mathcal{A}$ can be mapped, by applying Algorithm 2, to a corresponding view placement $\mathcal{P}$, in which the feed data flow is compatible with the upload / download decisions in $\mathcal{A}$. (We say, $\mathcal{A}$ is* consistent *with $\mathcal{P}$.)* [7]

Lemma 2 *A viable assignment $\mathcal{A}$ is consistent with only one view placement $\mathcal{P}$.*

Theorem 1 *The optimal view placement problem can be reduced to finding viable assignment with optimal cost (directly from Lemma 1 and Lemma 2).*

3.5 View Graphs with Single-level Joins

We start with a simple scenario, where feed-following CloudEdge applications are specified as view graphs with only single-level joins. We will extend the discussion to view graphs with multi-level joins in the following section.

3.5.1 Same Demand Rate

We first consider a special case of the view graphs with single-level joins, in which, for any vertex i in a demand graph, the feed data rates for all outgoing edges are the same, namely, $\forall (v_i, v_j) \in E, r_{v_i,v_j} = r_{v_i}$. Basically, a join operation requires the full feed data from each data source. This corresponds to the queries where no filtering (such as projection or selection) is performed before the join operation.

Instead of directly considering the cost of an assignment, we compute the gain of switching upload and download (which could be positive or negative) compared to a base viable assignment – a naïve solution that all vertices decide to upload their feed data. By switching a vertex v_i from uploading to download, the gain can be computed as follows: $gain_{v_i} = r_{v_i} - \sum_{(v_i,v_j)\in E} r_{v_j}$, namely the benefit of not uploading v_i's feed data at a cost of downloading all the feed data that are correlated with v_i's feed data.

Definition (Global optimality) *Given a demand graph $G = (V, E)$, the global optimal assignment is a viable assignment $\mathcal{A}$ that maximizes the total gains.*

[7]Proofs for all the lemmas are presented in a technical report [6].

To find an assignment $\mathcal{A}^{opt}$ that gives the global optimality, we consider a greedy approach where each vertex in a demand graph locally decides the assignment for its own benefit:

Definition (Local optimality) *Given a demand graph $G = (V, E)$, for each vertex $v \in V$, the local optimal assignment for v is a local decision on $\mathcal{A}_v$ that maximize the local gain. Specifically, $\mathcal{A}_v = D$ if and only if $gain_v > 0$.*

We then prove that the local optimality is actually consistent with the global optimality, which has two implications: First, the overhead for computing the local optimality is low, which is linear to the number of degree of the vertex in the demand graph. Second, we can partition the assignment problem and solve it in parallel. This is particularly important in cases where the demand graph is huge, as we can leverage the vast computation resources at the Cloud to solve it efficiently.

Theorem 2 *Given a demand graph $G = (V, E)$, the assignment $\mathcal{A} = \{\mathcal{A}_v | \mathcal{A}_v =$ local optimality at v, $v \in V\}$ is viable.*

Proof. Suppose there exists a conflict edge $e = (i, j)$, that is, $\mathcal{A}_i = D$ and $\mathcal{A}_j = D$. We have, from $\mathcal{A}_i = D$, that $gain_{v_i} = r_{v_i} - \sum_{(v_i,v_j)\in E} r_{v_j} > 0$. Therefore, $r_{v_i} > r_{v_j}$. Similarly, we can derive $r_{v_j} > r_{v_i}$ from $\mathcal{A}_j = D$. Contradiction. □

Theorem 3 *Local optimality is consistent with global optimality, namely, global optimality can be derived by individually applying local optimality.*

Proof. Theorem 2 shows that the assignment derived by individually applying local optimality is viable. Also, note that each local optimality is computing the maximal gain for an isolated physical link. Therefore, the global optimality is simply addition of the gains on the physical links. □

3.5.2 Different Demand Rates

We now extend the discussion to consider the scenario where, for a given vertex i, the feed data rates demanded by each of the other vertices may be different. For example, in the `nearby_friends` logical view, the feed rates for a particular user may be different with respect to each of its friends. Here, we assume that the feed data with a lower rate can be constructed using one with a higher rate, which corresponds to application of *sampling filters*. In other words, a filter that needs to sample x events/sec can be provided by another filter that samples y events/sec, for any $y \geq x$. In such a scenario, decisions on uploading vs. downloading are made for each edge (instead of each vertex) in the demand graph.

Assuming the rates r_{v_i,v_j} are sorted at vertex v_i, such that $r_{v_i,v_1} < r_{v_i,v_2} < ... < r_{v_i,v_p}$, it is not hard to see that an optimal assignment for the p sorted edges must have the pattern $[U, ..., U, D, ..., D]$ (because a lower-rate stream can be sampled from a stream with a higher rate, that is, the lower-rate feed data has been made available at the Cloud for "free").

Definition (Local optimality) *Consider the gain in an assignment $\forall j \leq k, \mathcal{A}_{v_i,v_j} = U, \forall j > k, \mathcal{A}_{v_i,v_j} = D$: $gain_{v_i,v_k} = r_{v_i,v_p} - r_{v_i,v_k} - \sum_{k+1\leq s\leq p} r_{v_s,v_i}$. We select $k = argmax_{0\leq j\leq p} gain_{v_i,v_j}$, and configure the assignment according to the $[U, ..., U, D, ..., D]$ pattern described above.*

It is not difficult to see, based on the similar reasoning, that the consistency theorem (Theorem 3) still hold. We next show that the viability theorem (Theorem 2) also hold.

Lemma 3 *After applying local optimality at vertex v_i, we have that $\mathcal{A}_{v_i,v_j} = D$ implies $r_{v_i,v_j} > r_{v_j,v_i}$.*

Theorem 4 *Given a demand graph $G = (V, E)$, the assignment $\mathcal{A} = \{\mathcal{A}_{v_i,v_j} | \mathcal{A}_{v_i,v_j} =$ local optimal assignment at $e = (v_i, v_j)$, $e \in E\}$ is viable.*

Proof. Suppose there exists a conflict edge $e = (v_1, v_2)$. Applying Lemma 3, we have $r_{v_1,v_2} > r_{v_2,v_1}$ from $\mathcal{A}_{v_1,v_2} = D$. Similarly, we have $r_{v_2,v_1} > r_{v_1,v_2}$ from $\mathcal{A}_{v_2,v_1} = D$. Contradiction. □

3.6 View Graphs with Multi-level Joins

In the case of view graphs with multi-level joins, we cannot naïvely apply the algorithm developed for view graphs with single-level joins: For single-level joins, the communication cost of the final resulting views is not considered (as the final notification cost is negligible). However, this is not the case for multi-level joins. For example, when naïvely applying the algorithm presented in the prior section, an edge device may individually decide to download other feed data and perform the computation locally. However, if the edge device is aware of the fact that the resulting view is then required as an input for the maintenance of a higher-level view (whose optimal placement is at the Cloud side), it may make a different decision. We next present how this challenge is resolved optimally by extending the single-level join algorithm.

3.6.1 View Placement in a Top-down Fashion

The intermediate views in the view graph can be considered as virtual feed sources, except that their locations need to be decided. Intuitively, given a view graph, we can make the upload vs. download decisions for the views in the top-down fashion: for a given vertex v_1 that corresponds to an intermediate view, as long as we know where it should be shipped to (based on the placement decision made by its parent view), the algorithm for the single-level joins can be straightforwardly extended by additionally considering the communication cost of the resulting views.

Note that the only destination we should consider is the Cloud side. Even if the destination is another edge device (as the intermediate view is required by another vertex v_2 located at the edge device), we should not consider the downloading part of the shipping cost (i.e., the cost of sending the output stream from *Cloud side* to that edge device), as this downloading cost is already considered in calculating the gain for v_2. Note that Assumptions 1 and 2 ensure that when considering vertex v_1, we can disregard the actual placement decision for its destination, as it will definitely be placed either at the Cloud or at some *other* edge that v_1 (or its subtree) do not overlap with. This key observation makes the extension of the algorithm possible, and it can be shown that the extended algorithm still guarantees a viable and optimal assignment.

3.6.2 Upload vs. Download in a Top-down Fashion

Our previous approach (for view graphs with single-level joins) derives the placement of views in the bottom-up fashion (see Section 3.5.1) after the upload vs. download decisions are made. We need to tweak the algorithm presented in Section 3.6.1 to decide the upload vs. download assignment, based on the parents' *assignment* instead of their placement.

Once we know the decision of the parent vertex v_1, we consider what decision should be made for a child vertex v_2. Again, v_2 has two choices – either upload or download.

- If the decision of the parent vertex v_1 is **download**, it means that there is no need to communicate the resulting view to the Cloud server. Therefore, when finding the local optimality for v_2, the cost of communicating the resulting view is not considered in computing the gains. Instead, we should consider the cost of downloading the corresponding feeds for performing v_1 (i.e., the other input feeds for v_1). This is because, if v_2 chooses to download, v_1 will need to be colocated with v_2.
- If the decision of the parent vertex v_1 is **upload**, it means that the resulting view of v_2 should be made available at the Cloud server. Therefore, when finding the local optimality for v_2, the cost of communicating the resulting view should be considered.

Algorithm 3 Compute Optimal Assignment

```
1: func Assignment(G^Q = (V^Q, E^Q), G^D = (V^D, E^D))
2: TopoOrder ← V^Q sorted by topology sort.
3: for ∀ v_i ∈ TopoOrder in the top-down order do
4:    EStart ← {e_i = (v_i, v_i') | e_i ∈ E^D}
5:    Sort EStart according to r_{v_i,v_i'}
6:    r^max ← max_{(v_i,v_i')∈EStart} r_{v_i,v_i'}
7:    for ∀ e_k = (v_i, v_k) ∈ EStart do
8:       gain_k ← r^max − r_{v_i,v_k}
9:       for k + 1 ≤ s ≤ p do
10:         gain_k ← gain_k − r_{v_s,v_i}
11:         // Next, consider the cost of join output
12:         v^o_{v_s,v_i} ← the parent of v_s and v_i in G^Q
13:         if ∃v_j such that Assign_{v^o_{v_s,v_i},v_j} = U then
14:            r^o_{v_s,v_i} ← the rate of join output stream
15:            gain_k ← gain_k − r^o_{v_s,v_i}
16:   k^opt ← argmax_{0≤k≤p} gain_k
17:   for ∀1 ≤ k ≤ k^opt do Assign_{v_i,v_k} ← U
18:   for ∀k^opt < k ≤ p do Assign_{v_i,v_k} ← D
19: return Assign
```

Algorithm 3 takes a view graph G^Q and its corresponding demand graph G^D as the input, and computes the optimal assignment. The algorithm applies to a generic scenario where it assumes a view graph with multi-level joins, and per-edge demand rates (i.e., the rates associated with the demand edges starting from a given vertex might be different). We show that the assignment generated by Algorithm 3 is viable and optimal.

Theorem 5 *Given a view graph $G^Q = (V^Q, E^Q)$ and its demand graph $G^D = (V^D, E^D)$, the assignment $\mathcal{A}$ generated by Algorithm 3 is viable (directly from Lemma 3).*

Lemma 4 *Given a view graph $G^Q = (V^Q, E^Q)$ and its corresponding demand graph $G^D = (V^D, E^D)$, the assignment $\mathcal{A}$ generated by Algorithm 3 is consistent with the optimal view placement $\mathcal{P}^{OPT}$.*

Theorem 6 *The assignment $\mathcal{A}$ generated by Algorithm 3 is globally optimal (directly from Lemma 4 and Lemma 2).*

3.7 Asymmetric Upload / Download Costs

So far we have assumed that the upload cost and the download cost are the same. However, in reality, it might not be the case. For example, the per-unit prices of bandwidth utilization for uploading and download might be different (e.g., a Cloud service provider may introduce asymmetric costs to encourage users to feed data into the Cloud); also, an edge device might exhibit different battery consumptions for uploading and downloading.

In this section, we further extend our discussion to consider asymmetric upload / download costs. We denote the per-unit cost for uploading and downloading as C^u and C^d. For scenarios where $C^u < C^d$, the results for $C^u = C^d$ presented in the previous sections still hold – the key viability theorem (Theorem 2) holds.

On the other hand, deciding optimal view placement is a harder problem for cases where $C^u > C^d$. For a special case where

$C^d = 0$, we can show that the optimal view placement problem is provably hard by reduction from the classic weighted min vertex cover (WMVC) problem. Essentially, the viability theorem breaks in these cases, therefore, having edge devices individually apply local optimality may result in conflicts. However, a viable assignment can still be obtained, if we can resolve the conflicts by setting some vertices in the demand graph to upload with higher rates.

Assume that given a demand graph G, after applying Algorithm 3, we remove the demand edges (v_i, v_j) if $Assign_{v_i,v_j}$ is consistent with $Assign_{v_j,v_i}$ (i.e., not both are D). We call the left-over graph as the *residual graph*, where switching the assignment of any edge results in increased cost. The problem reduces to the WMVC problem in the residual graph, which lacks an efficient general solution. In the rest of the section, we discuss a condition that if satisfied, the optimal view placement problem can be solved efficiently.

Definition (Skew) *Given a demand graph $G = (V, E)$, the* skew *of a vertex $v \in V$, S_v is defined as the ratio between the maximum and minimum rate associated with the outgoing edges from v. Namely, $S_v = \max_{(v,i)\in E} r_{v,i} / \min_{(v,j)\in E} r_{v,j}$.*

The skew of G is defined as the maximum skew among the nodes in G. Namely, $S = \max_{v\in V} S_v$.

Lemma 5 *Given the skew S of a graph G, if $C^d < C^u < (1 + 1/S) \cdot C^d$, after applying local optimality on all vertices, the residual graph G' that consists of the conflict edges is acyclic (i.e., separated trees).*

Theorem 7 *If $C^d < C^u < (1 + 1/S) \cdot C^d$, the optimal view placement can be found in P-time.*

Proof. By applying Lemma 5, we can conclude that G' is acyclic. We show that, for each tree in the residual graph G', we can find its weighted minimal vertex cover in linear time, using a dynamic program algorithm:

Starting from leaf vertices, for each vertex v, consider the cost of the vertex cover for the subtree (rooted by v), having (or not having) v in the cover set. For any inner vertex v, if v is not in the cover set, then all the children of v should be in the cover set: $Cost_v^- = \sum_{i\in child(v)} Cost_i^+$. On the other hand, if v is in the cover set, then each subtree can independently choose its vertex cover: $Cost_v^+ = c_v + \sum_{i\in child(v)} \min(Cost_i^-, Cost_i^+)$. □

Note a special case where the stream rates required by different friends are the same, we have that the optimal placement can be found in P-time, if $C^d < C^u < 2 \cdot C^d$ (which holds in most practical scenarios). Empirically, even if $C^u \geq 2 \cdot C^d$, the conflicting edges still form isolated trees.

4. DISCUSSION AND EXTENSIONS

Section 3 focused on view templates with multi-level joins with sampling filters. We next discuss extensions that are beyond our current prototype implementation, including how our algorithms also support general view templates (§ 4.1, 4.2) and offline distributed view maintenance (§ 4.3). We also cover other issues such as node failures, runtime changes in the query graph and event rates, and device constraints (§ 4.4, 4.5 and 4.6).

4.1 Handling General View Templates

General view templates can be specified as a *query graph* [22, 8] $\mathcal{G}$, a directed acyclic graph (DAG) over a set of *black-box* operators (denoted as $\mathcal{O}$), where the leafs in $\mathcal{G}$ are called *sources*, and the roots are called *sinks*. Each operator in $\mathcal{O}$ may take zero (for the sources) or more inputs, and its output may be used as an input to other operators. Operators may be single-input (e.g., selection) or multi-input (e.g., multi-way join).

Interestingly, the high-level intuitions of our placement algorithm from Section 3 continue to hold for general view templates—each (virtual) vertices in the view graph individually decides (in a top-down order) whether it should upload or download its output to optimize its local cost. The viability of the assignment is still guaranteed as before. Moreover, given that operators are black-boxes, there exist no further opportunities to exploit sharing across the different views in the view graph (as was done in Section 3.5.2). A similar reasoning as Theorem 3 shows that consistency between locally optimal choices and the globally optimal solution still holds. Thus, the problem can again be reduced to finding the optimal upload/download assignments, and our proposed local optimality algorithms can be used.

4.2 Handling General Network Topologies

We have focused on the Cloud-Edge star topology in this paper. In a Cloud setting, the overall network graph G has two components: a "core graph" G_c which is an arbitrary graph of machines within the Cloud; and an "edge graph" G_e that treats the Cloud as a single vertex with edge devices hanging off of this vertex in a star topology. General view placement for a large graph such as G is very expensive [22, 19]. However, we can still find the optimal solution as follows:

- First, we solve the cloud-edge view placement problem on graph G_e in linear time as shown earlier.
- Next, for views placed at the virtual "Cloud vertex", perform view placement on G_c using prior techniques [22, 19] for arbitrary graph or tree topologies. This is more expensive, yet feasible because $\|G_c\| \ll \|G\|$ (usually by orders-of-magnitude).

We show in Theorem 8 that this two-level approach *does not violate global view placement optimality* in the full graph (G), in the common case where network costs within the Cloud are symmetric.

Lemma 6 *The view placement $\mathcal{P}(G_e)$ for the edge graph is* compatible *with the view placement $\mathcal{P}(G_c)$ for the core graph, that is, a view placed at an edge device v in $\mathcal{P}(G_e)$ is placed at its corresponding node v' (which v' is directly connected to) in $\mathcal{P}(G_c)$.*

Lemma 7 *The view placement $\mathcal{P}(G_e)$ for the edge graph is* consistent *with the view placement $\mathcal{P}(G)$ for the complete graph, that is, a view is placed at an edge device A' in $\mathcal{P}(G_e)$ iff it is placed at the same edge device in $\mathcal{P}(G)$.*

Theorem 8 *View placement $\mathcal{P}(G_e) \cup \mathcal{P}(G_c)$ is globally optimal.*

Proof. By applying Lemma 6 and 7. Once the $\mathcal{P}(G_e)$ is decided, the problem reduced to optimized the cost of the communication within the cloud, which is exactly what $\mathcal{P}(G)$ achieves. □

4.3 Offline View Maintenance

The algorithms presented in Section 3 are not specific to online feed-following applications. It applies to offline distributed view mainenance in Cloud-Edge structured databases as well. For instance, service providers such as Facebook, Microsoft, Twitter, and Yahoo perform offine analytics to build user profiles and better tailor the services for the users. When performing the analytics based on data that is distributed across machines in one of several data centers, finding a low-bandwidth economic query plan is of great importance for better cluster utilization. Instead of naïvely fetching distributed data and collectively performing the analytics on dedicated machines in the Cloud, the Race system can be leveraged to compute the optimal view placement based on the data sizes (instead of feed rates in feed-following applications).

4.4 Fault Tolerance

Fault tolerance is easily achieved at the Cloud using techniques proposed in previous work [3]. We next discuss robustness to edge failures. Intermittent connection losses are handled in the data plane by logging data (either at the Cloud or edge) and sending them as a batch once connectivity is re-established. The engine can use application time for query semantics, such that query correctness is not affected except for some delay in result computation. In case of a permanent failure of some edge E, we guarantee that *only queries that use data from E are affected* (this cannot be avoided since such queries use data from E). This useful result guarantees that the effect of edge failures are localized, and follows directly from our restriction (see Assumption 1) that an edge E may only process parts of view maintenance that process data from E.

4.5 Handling Dynamism

The Race optimizer expects the availability of the view graph as well as rate statistics for all streams as input. However, the view graph may change over time, for example, due to the addition and removal of edges in the social network. We model on-demand queries by the addition and subsequent removal of a view, which requires support for a dynamically changing view graph. Similarly, event rates may also change as each edge device periodically reports the monitored feed rates to the optimizer. It is necessary to adapt to these changes during runtime. Since our optimizer is very efficient (it takes only 3.2 seconds to optimize for 186K edges; see Section 5), periodic re-optimization and adjustment of view placement is viable.

However, re-optimization may encounter deployment overhead (e.g., sending query definitions to edges). If we re-optimize frequently, the re-optimization overhead may overshadow the benefits of optimization. One solution is to use a cost-based online algorithm [4]: estimate and maintain the accumulated loss due to *not* performing re-optimization, and perform re-optimization only if the accumulated loss exceeds the overhead of re-optimization. A nice property of this approach is that it is 3-competitive—it is guaranteed that the overall cost is *at most* three times as much as the optimal (even with a priori knowledge of the changes).

4.6 Device Constraints

Consider the case where some individual devices have constraints on how much computation can be placed on them. These constraints may, for example, be based on battery life, device capabilities, and connectivity. Assume that each view V_i is associated with a cost c_i of maintaining the view. Each vertex (device) has a constraint on how much computation can be performed on the vertex. We proceed as follows: first, we find the optimal solution as before without considering these constraints. Then, for each vertex whose constraint is violated, we switch some subset of **download** decisions for that vertex to **upload**. Note that this modification is localized; it does not affect any other Cloud-Edge link. The challenge of determining the set of decisions to switch from download to upload to minimize the increase in network cost while meeting the device constraint is equivalent to the 0-1 knapsack problem, for which we can use well-known greedy approximations.

5. EVALUATION

We answer the following questions on a real large-scale deployment of Race: (1) how necessary and efficient is our optimizer, for realistic workloads and as compared to current solutions? (2) how well does our real deployment of Race in a data center behave during runtime? (3) what is the cost of redeployment after reoptimization due to changing statistics?

```
var query0 = from e1 in location
    from e2 in socialNetwork
    where e1.UserId==e2.UserId
    select new { e1.UserId, e1.Latitude,
                 e1.Longitude, e2.FriendId };
var query1 = from e1 in query0
    from e2 in location
    where e1.FriendId == e2.UserId &&
          Distance(e1.Latitude, e1.Longitude,
           e2.Latitude, e2.Longitude) < THRESHOLD
    select new { User1 = e1.UserId, User2 = e2.UserId };
```

Figure 15: Specification of the `nearby_friends` logical view.

5.1 Implementation Details

We have implemented the Race system in C# with Microsoft StreamInsight [25] as the DSMS running at the Cloud and on edge devices. For edges, we built a lightweight version of StreamInsight that runs as a Silverlight service for Windows Phone and similar devices (which is demonstrated in [5]).

Specification. We adopt LINQ [30] as the specification language for Race. CQ languages such as LINQ and StreamSQL [17] enable a declarative expression of time-oriented operations such as temporal joins and windowed aggregations over streaming temporal data. The declarative nature of CQs allows app developers to express their core application logic in a *network-topology-agnostic* manner, where they can focus on "*what*" their apps do, instead of "*how*" they are implemented.

Recall that the `nearby_friends` logical view finds all user pairs `(User1, User2)` that satisfy the conditions: 1) `User2` is a friend of `User1`, and 2) the two users are geographically close to each other at the same time. There are two inputs to the view maintenance of `nearby_friends`, namely the location feeds (`location`) reported by the edge devices, and the social network reference data (`socialNetwork`). The locations are actively collected at runtime, whereas the social network data is relatively slow-changing and is available at the Cloud. The `nearby_friends` logical view can be defined as a two-stage temporal join query (see Figure 15). The first query (`query0`) joins locations with the social network, while the second query (`query1`) correlates friend locations.

The query specification in Figure 15 operates over conceptually unified location and social network feeds, and is thus network-topology-agnostic. As another example, suppose we want to find friends who visited our location (e.g., a restaurant) within the last week. We can simply replace the `location` input in `query1` with `location.AlterLifetime(TimeSpan.FromDays(7))`. This extends the "lifetime" of `location` events to 7 days, allowing the join to consider events from friends within the last week.

Query Processing. The DSMS accepts query and metadata specification in XML format. Hence, the Race processor intercepts and parses XML documents describing the view templates, necessary adapters, and event types. The outputs of the view placement optimizer are also encoded as XML documents so that the DSMS can seamlessly execute them.

5.2 Setup, Query, and Datasets

We run the Race optimizer on the Cloud, which consists of a single quad-core Intel 2.8GHz machine running 64-bit Windows 7 Enterprise. In order to stress-test the server with real client connections, we use 10 client machines (on a different data center from the Cloud server), each running up to 50 instances of the client application. Windows OS restrictions allow us to only run 50 client application instances per client machine; this limits our "real deployment" experiments to 500 real clients. However, other exper-

Parameter	Value
# users in real social network	957K
# friends per user	$[5, 4956]$
download/upload cost ratio	1
Synth. friend dist. (N users)	$\min(0.1N, 4956)$
Zipf parameter	2
# real client connections	$[100 - 500]$
# users in real GPS trace	167
distance threshold for meeting	100 meters

Figure 9: Experimental parameters.

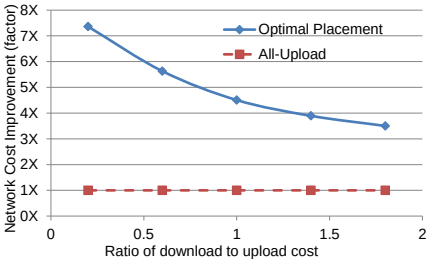


Figure 10: Cost improvement vs. C^d/C^u.

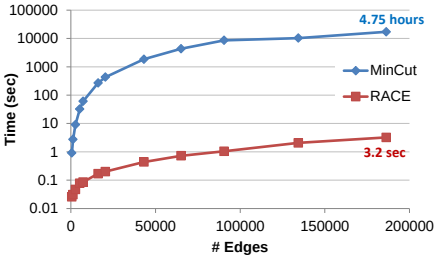


Figure 11: Time taken, increasing #edges.

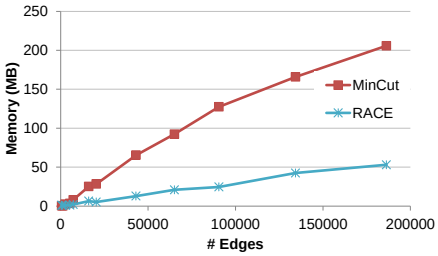


Figure 12: Memory usage vs. #edges.

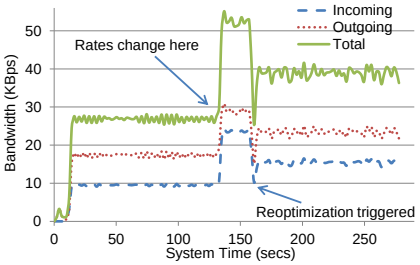


Figure 13: Behavior in real deployment.

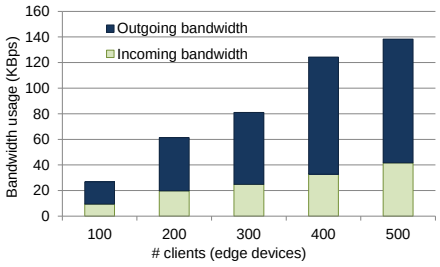


Figure 14: Bandwidth vs. #clients.

iments (such as those testing our optimizer's scalability) use up to $200K$ edge devices.

We use the social Foursquare application (defined by the `nearby_friends` logical view described above) as the declarative query submitted by the end user in LINQ. We use both real and synthetic datasets in our experiments (default experimental parameters are summarized in Figure 9).

Social Network Data. We use a Facebook social graph dataset [21], that contains 957K users obtained Facebook-wide by 28 independent Metropolis-Hastings random walks [24]. For synthetic data, we use the same distribution for number of friends as seen in the real data. We also experiment with a Zipf distribution ($z = 2$) for number of friends between 1 and $\min(4956, 0.1 \cdot N)$, where N is the number of users and 4956 is the maximum number of friends for a user in our dataset.

Location Data. For each user in the social network, we synthetically generate its GPS location updates drawn from the distribution of speeds in a real-life data set. We use a GPS trajectory dataset collected in the MSR Geolife project [36] by 167 users from 2007 to 2010. It contains $17,355$ trajectories with a total distance of about 1 million kilometers and a duration of $48,000$+ hours.

Demand Rate. Finally, the demand between two friends (which determines the sampling rate) is drawn from a probability distribution of two users meeting (coming within 100 meters of each other) in the Geolife trace.

5.3 Experiments and Results

We evaluate Race in a combination of simulation and deployment experiments. We study the effectiveness (in reduction of communication cost) and efficiency of Race's view placement algorithm at scale in simulation (Bullet 1 and 2), and evaluate the actual system performance in deployment on the StreamInsight DSMS platform (Bullet 3, 4 and 5). We next present our observations in these experiments.

1) Need for Optimization. We use a social network with 100K users and 8.6M edges, where the distribution of number of friends uses the distribution found in the Facebook dataset. The base location data generation rates at all edge devices is set to 1, while each node independently samples from the "probability of meeting" distribution (see Section 5.2) derived from real data, to arrive at the actual sampling rate for each of its neighbors in the social graph.

We vary the ratio between the download and upload costs, i.e., C^d/C^u ranges from 0.2 to 2.0, and report the improvement (factor) as compared to All-Upload, the baseline approach of performing all computations at the Cloud. Compared to the baseline, we observe consistent and significant improvement in the full range of C^d/C^u ratios: Race's optimal placement improves performance (i.e., aggregate communication cost) by 3.5X to 7.3X over All-Upload. Our algorithms give greater benefit if the ratio C^d/C^u is small because with smaller C^d, an edge is more likely to benefit from choosing to download data. Note that on existing mobile phones, $C^d/C^u \leq 1$, which favors Race's optimizations.

Our algorithm provides a conflict-free assignment in all cases. Note that there are no unresolvable conflicts, even in the extreme case of $C^d/C^u = 0.2$ in which case 7836 residual graphs are left. However, all these residual graphs are acyclic, and thus the dynamic programming algorithm is able to resolve all conflicts and find the optimal solution in linear time. This observation empirically shows that we can find the optimal solution efficiently even in highly skewed scenarios.

2) Optimizer Efficiency. We now compare the efficiency of our optimizer with the current state-of-the-art min-cut-based algorithm for trees from [22]. We set $u = d$ as this algorithm does not directly address asymmetric networks. We implement min-cut using the max-flow algorithm in the QuickGraph library[8]. We include the overhead for constructing the demand and query graphs from a LINQ query (cf. Section 2) and optimization (cf. Section 3). The cost of deploying query fragments to edge devices is evaluated separately below.

We use a synthetic social network, with friends chosen from a Zipf distribution ($z = 2$) as in Figure 9. We assume that the distribution of feed rates (events/sec) follows a Zipf ($z = 2$) in the range $[1, 10]$, and vary the number of edge devices.

Figure 11 and Figure 12 show the execution time and memory overhead of the Race optimizer vs. the min-cut algorithm. As

[8] `http://quickgraph.codeplex.com/`

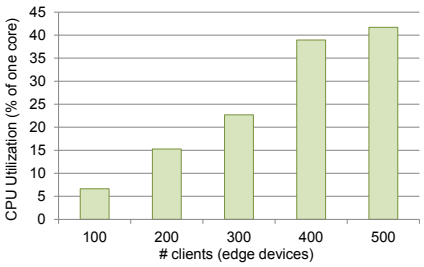


Figure 16: CPU utilization vs. #clients.

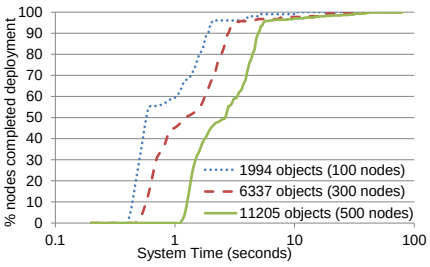


Figure 17: Query deployment time CDF.

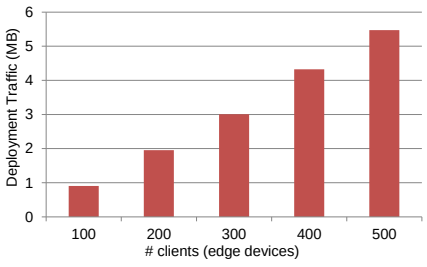


Figure 18: Deployment traffic vs. #clients.

shown, Race runs several orders of magnitude faster than min-cut (e.g., 3.2 seconds for Race compared to 4.75 hours for min-cut, on a social network with 20K nodes and 186K edges), making it suitable for our applications. Race also consumes less memory than min-cut. The memory utilization and completion time for Race are proportional to the size of the demand (or query) graph. Note that in addition to running faster and consuming less memory than min-cut [22], the Race algorithm also optimally handles the sharing of intermediate results and asymmetric networks.

3) Runtime Behavior with Reoptimization. We set the number of edge clients to 100 (distributed across 10 physical machines) and connect to the Race server to demonstrate runtime performance and reoptimization. Edge devices generate location updates at a rate drawn from a Zipf distribution ($z = 2$) between 1 and 10. Figure 13 shows the incoming, outgoing, and total bandwidth (in KBps) incurred at the Cloud server (at the data plane) during the course of the experimental run. The system initially finds a good placement, incurring an average bandwidth cost of around 28KBps. After two minutes, the rates of half the edge devices are increased by a factor of 5, while the rates of the remaining devices are decreased by a factor of 2.33. As a result, the total bandwidth usage increases to around 51KBps. We next initiate re-optimization, and find that the newly deployed placement costs around 40KBps. We see that redeployment is dynamic, i.e., it does not result in any downtime in terms of bandwidth usage at the Cloud.

4) Bandwidth, CPU, and Memory in Deployment. We vary the number of clients (running on physical machines and connecting to the Cloud server) from 100 to 500, and plot incoming/outgoing bandwidth and memory usage at the Cloud server. Location event rates are chosen from a Zipf as before.

We observe from Figure 14 that server bandwidth increases with increasing number of clients as expected, but is quite low (less than 140KBps) even for 500 clients. Figure 16 plots server CPU utilization in terms of percentages, where 100% corresponds to one CPU core fully utilized. We found that with 500 active clients, actively generating events and running queries, the CPU load at the server was less than 42% of one core (the load was distributed evenly across 4 cores). Further, memory usage at the server was found to be less than 350MB. Thus, we expect a typical modern server to easily handle tens of thousands of active clients (or more, since in practice, many clients are likely to be inactive). Note that the Cloud server runs many query fragments. Since the query fragments are independent, the Cloud server can easily be scaled out by distributing the fragments across a farm of machines using simple (e.g., round robin) load balancing schemes.

5) Query Deployment Cost. We investigate the time taken by Race to deploy metadata objects (e.g., types, query templates, and query instances) to the DSMS engines located at the subscribed edges as well as at the Cloud. Metadata deployment is done in parallel threads (one for each destination), but needs to be serial for a given engine since any metadata object may reference items in previous objects. Figure 17 shows the cumulative distribution function — for a given time duration, we show the percentage of engines for which deployment has been completed within that time. We see that even for 726 objects (on 100 nodes) and 4625 objects (on 500 nodes), more than 95% of nodes complete their deployment within 2secs and 5.5secs respectively. Figure 18 shows aggregate network traffic due to deployment; we see that even for 500 nodes (11205 objects), the total traffic due to deployment is only around 5.5MB.

6. RELATED WORK

Feed-following Applications. Several recent works proposed various optimizations for feed-following applications [18, 26, 27]. In [26], authors gives an overview of various challenges, including the view placement problem we address in this paper, related to feed-following applications and explains why prior works on pub/sub systems, caching, and materialized views are inadequate to address these challenges. Feeding Frenzy [27] proposes a view selection algorithm using two types of views: *producer pivoted views*, which maintain the latest k events from every producer, and *consumer-pivoted views*, which incrementally maintain the result of a consumer feed. Feeding Frenzy selects either a producer-pivoted or a consumer-pivoted view for every edge in the communication network such that the sum of event and query processing cost is minimized. In contrast, we consider the view placement problem for many edge devices connected to the Cloud. All prior works assume that final feeds to consumers are produced by applying simple selection and union operations; in contrast, our algorithms are more general—they support views with selection, projection, correlation (join) and arbitrary black-box operators, and can even refer to other views. Finally, unlike prior works, we consider feed-following applications that are distributed over a large number of edge devices and the Cloud.

Publish/subscribe Systems. In publish/subscribe systems [9, 16, 11], the goal is to efficiently match events from a set of publishers to a set of relevant subscribers. Compared to feed-following Cloud-Edge systems, pub/sub systems are usually limited in expressiveness, often perform matching at the cloud, and usually have disjoint sets of publishers and subscribers. Distributed pub/sub systems focus on routing in a general network topology with heuristics; in contrast we provide optimal placement algorithms for a star-like topology formed by many edge devices and the Cloud.

Databases and Streams. Distributed database query optimization is a classic research topic (See the survey [20]). Most research in this field has focused on minimizing total communication cost for executing a database query by carefully choosing join orders and possibly the places to execute various join operations [10, 12, 13, 20]. Stream processing research has also considered operator placement in a network-aware manner [2, 29, 35, 37]. As in previous work, we separate join order optimization from placement.

Unlike these systems, however, we optimize multiple queries together, find a globally optimal solution, handle asymmetric links, and scale to millions of data sources.

Wolf et al. (SODA [33]), Li et al. [22] (which we refer as LI) and Kalyvianaki et al. (SQPR [19]) propose multi-query optimization techniques for more general network topologies. Unlike Race, SODA does not provide any optimality guarantee. Both LI and SQPR provide optimal solutions; LI uses a hypergraph min-cut algorithm for a tree topology, while SQPR uses linear programming. Unfortunately, these solutions are impractical for Cloud-Edge apps with millions of users in large dynamic scenarios where reoptimization must be done frequently due to user churn and data rate changes. In contrast, Race is an end-to-end system that exploits the special *star topology* of Cloud-Edge systems and provides very efficient and scalable yet optimal solutions for our applications. Note that even though solutions designed for more general networks can be used for a star topology, our solutions in this paper are several orders of magnitude faster (see Section 5).

There is a great deal of view materialization literature on choosing the right set of views in the OLAP [14] and SQL [1] settings. As discussed in [26], they do not provide a complete solution for feed-following applications. Traditional materialized views tend to be static, in that we analyze the query workload, choose a set of views, and maintain them. In feed-following, however, the query workload changes quickly, the number of possible views is huge, and each is a small list of events. Moreover, existing works on view materialization deal mostly with view selection and scheduling problem; while we focus on view selection problem.

Sensor Networks. Several papers consider the problem of simultaneously optimizing multiple queries. In the context of sensor networks, the problem has been studied for aggregate queries [28, 31, 34]. These works do not consider general SQL operations over millions of nodes. Moreover, they assume that data transferred between nodes are of the same size, aggregates have a constant size, and upload/download costs are equal. None of these assumptions hold for Cloud-Edge apps in our setting, which makes these techniques unsuitable for our purposes. Our results on local-optimization leading to global-optimization have a similar flavor to results shown by Silberstein et al. [27, 28]; however, the problems and solutions are fundamentally different.

Other Related Work. The use of database techniques to optimize distributed systems has been explored before for multi-player games [32] and network systems [23]. Race focuses on different applications (e.g., mobile apps), topologies (cloud-edge), scale (millions of devices), and optimization types (operator placement).

7. CONCLUSION

We proposed a new system called Race to support a broad class of *feed-following Cloud-edge* applications. We modeled general feed-following applications as a view maintenance problem, and developed novel algorithms that are highly efficient yet *provably* minimize global communication overhead. Our algorithms are widely applicable to general view graphs, asymmetric networks, and consider sharing of intermediate results. We implemented Race that leverages a Data Stream Management System (DSMS) as the computing engine for distributedly executing different portions of the application logic on edge devices and the Cloud. Experiments over real datasets indicated that the Race optimizer is orders-of-magnitude faster than state-of-the-art optimal techniques. Further, our placements incurred several factors lower cost than simpler schemes for a social Foursquare application (defined by the `nearby_friends` logic view) over a realistic social network graph with 8.6 million edges. While Race is easily parallelizable within the Cloud, experiments with a real deployment showed that is also scales very well using just a single server machine.

8. REFERENCES

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

[2] Y. Ahmad and U. Çetintemel. Network-aware query processing for stream-based applications. In *VLDB*, 2004.

[3] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the borealis distributed stream processing system. In *SIGMOD*, 2005.

[4] D. Black and D. Sleator. Competitive algorithms for replication and migration problems. Technical report, Carnegie-Mellon University, 1989.

[5] B. Chandramouli, J. Claessens, S. Nath, I. Santos, and W. Zhou. RACE: Real-time applications over cloud-edge. In *ACM SIGMOD*, 2012.

[6] B. Chandramouli, S. Nath, and W. Zhou. Supporting Distributed Feed-Following Apps over Edge Devices. `http://aka.ms/race-tr`.

[7] B. Chandramouli et al. Value-based notification conditions in large-scale publish/subscribe systems. In *VLDB*, 2007.

[8] B. Chandramouli et al. Accurate latency estimation in a distributed event processing system. In *ICDE*, 2011.

[9] A. Demers et al. Towards expressive publish/subscribe systems. In *EDBT*, 2006.

[10] A. Deshpande and J. M. Hellerstein. Decoupled query optimization for federated database systems. In *ICDE*, 2002.

[11] F. Fabret, F. LLirbat, J. Pereira, J. A. Pereira, I. Rocquencourt, I. Rocquencourt, and D. Shasha. Publish/subscribe on the web at extreme speed. In *ACM SIGMOD*, 2000.

[12] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query optimization for parallel execution. In *SIGMOD*, 1992.

[13] M. N. Garofalakis and Y. E. Ioannidis. Parallel query scheduling and optimization with time- and space-shared resources. In *VLDB*, 1997.

[14] H. Gupta. Selection of views to materialize in a data warehouse. In *ICDT*, 1997.

[15] W. Hong and M. Stonebraker. Optimization of parallel query execution plans in xprs. In *PDIS*, 1991.

[16] Y. Huang and H. Garcia-molina. Publish/subscribe tree construction in wireless ad-hoc networks. In *MDM*, 2003.

[17] N. Jain et al. Towards a Streaming SQL Standard. 2008.

[18] F. P. Junqueira, V. Leroy, M. Serafini, and A. Silberstein. Shepherding social feed generation with sheep. In *Proceedings of the Fifth Workshop on Social Network Systems (SNS)*, 2012.

[19] E. Kalyvianaki et al. SQPR: Stream query planning with reuse. In *ICDE*, 2011.

[20] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 2000.

[21] M. Kurant, M. Gjoka, C. T. Butts, and A. Markopoulou. Walking in Facebook: A Case Study of Unbiased Sampling of OSNs. In *IEEE INFOCOM*, 2010.

[22] J. Li, A. Deshpande, and L. Getoor. Minimizing communication cost in distributed multi-query processing. In *ICDE*, 2009.

[23] B. Loo et al. Declarative networking: language, execution and optimization. In *ACM SIGMOD*, 2006.

[24] N. Metropolis et al. Equation of state calculation by fast computing machines. In *J. Chem. Physics, vol. 21, pp. 1087-1092*, 1953.

[25] Microsoft StreamInsight. `http://www.microsoft.com/`.

[26] A. Silberstein, A. Machanavajjhala, and R. Ramakrishnan. Feed following: The big data challenge in social applications. In *DBSocial*, 2011.

[27] A. Silberstein, J. Terrace, B. F. Cooper, and R. Ramakrishnan. Feeding frenzy: selectively materializing users' event feeds. In *SIGMOD*, 2010.

[28] A. Silberstein and J. Yang. Many-to-many aggregation for sensor networks. In *ICDE*, 2007.

[29] N. Tatbul et al. Load shedding in a data stream manager. In *VLDB*, 2003.

[30] The LINQ Project. `http://tinyurl.com/42egdn`.

[31] N. Trigoni, Y. Yao, A. Demers, and J. Gehrke. Multi-query optimization for sensor networks. In *DCOSS*, 2005.

[32] W. White, C. Koch, N. Gupta, J. Gehrke, and A. Demers. Database research opportunities in computer games. *SIGMOD Rec.*, 36:7–13, September 2007.

[33] J. Wolf et al. SODA: an optimizing scheduler for large-scale stream-based distributed computer systems. In *Middleware*, 2008.

[34] S. Xiang, H. B. Lim, and K.-L. Tan. Impact of multi-query optimization in sensor networks. In *DMSN*, 2006.

[35] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic load distribution in the borealis stream processor. In *ICDE*, 2005.

[36] Y. Zheng, X. Xie, and W.-Y. Ma. Geolife: A collaborative social networking service among user, location and trajectory. *IEEE Data Engg. Bulletin,*, June 2010.

[37] Y. Zhou, B. C. Ooi, and K.-L. Tan. Dynamic load management for distributed continuous query systems. In *ICDE*, 2005.