Materialized View Selection in Feed Following Systems

Kaiji Chen, Yongluan Zhou
University of Southern Denmark
Email: chen@imada.sdu.dk, zhou@imada.sdu.dk

***Abstract*—Recently emerging feed-following applications generate personalized event streams from various feeds and deliver them to a large number of users. To provide a low-latency service, a feed-following system has to buffer the events in a number of tables, called materialized views, and choosing views to materialize is critical to the system performance. State-of-the-art solutions only consider view selections for each individual user. Due to the existence of very popular feeds and social communities, users often share a lot of feeds that they follow and hence performing a global optimization by considering all the users can significantly enhance the system performance. However, performing such a global optimization needs to choose views for materialization from an exponential number of possible ones. To solve the issue, we propose an effective method to generate candidate views that are potentially beneficial. We then propose several cost-based algorithms to solve the global view selection problem, which adopt a cost model that captures the cost of both user query processing and view maintenance and make use of the containment relationships among the sets of feeds followed by the individual users. We implement the complete approach in a prototype system and perform experiments on a computing cluster using both real and synthetic data. The results indicate that our approach outperforms the state-of-the-art approaches significantly.**

I. Introduction

In online social networks like Facebook and content aggregators like Google Now, users may follow the contents generated by other users or articles from news websites with one of the server provided orders. We call the services that provide latest aggregated messages from multiple sources to users as feed-following services. A feed in a feed-following service generates time-ordered events such as news, locations or status updates, which will be distributed to all the users who follow this feed. According to the users' requirements, a feed-following system typically provides a personalized *view* for each user, which aggregates the feeds that are followed by the user.

As applications like Facebook, could have millions of daily active users and millions of messages or links are generated per 20 minutes [1], building a large-scale feed-following system is inevitable. However, scaling a feed-following system is challenging because it has to aggregate messages from a large-number of sources and provide different personalized views to a large number of users. As shown in previous work [2], a critical optimization is to determine when and how the system should update the users' personalized views. There are two possibilities: (1) *push*, where upon the arrival of updated messages from a feed, the relevant personalized views will be updated, and (2) *pull*, where a view will be updated only when a user poses an update request to the system. The solution proposed in [2] provides an optimization algorithm to determine whether a view of each user should be updated using push or pull. Intuitively, if the cost of updating a view using the push strategy can be paid off by its high access frequency by the users, then using the push strategy can not only reduce the runtime cost of view maintenance but also reduce the latency for the users to access the view.

As indicated by statistics of popular online social networks, such as [3], there exist feeds followed by millions of users and users following thousands of feeds. In other words, there exist many very popular feeds and many users may follow many common feeds. Updating and optimizing individual user views as done in previous approaches [2] misses the opportunities to minimize the system workload by sharing arbitrary subset of user's following set among users and hence is suboptimal. More specifically, one can maintain an optimized set of *materialized views*, each of which aggregates over some subsets of feeds using the push strategy, so that these views can be used to generate the personalized views for all the users. By carefully optimizing such shared views, one can minimize the cost of generating and maintaining personalized views for individual users.

However, exploiting the aforementioned opportunities is nontrivial. Given the set of feeds followed by the users, there is a very large number of possible subsets of feeds that could be maintained as materialized views. It has been shown that the possible number of views for a group of users is exponential to the maximum number of feeds followed by the users [4]. Note that even with a given set of materialized views, choosing a minimum subset to generate the personalized views for each user is equivalent to the minimum set cover problem, a well-known NP-hard problem, and hence expensive to compute, not to mention the complexity of choosing the optimized set of materialized views. A good news is that the size of network community which share common connections among each other will be around 100 as presented in [5] even for large social network graphs. Their results agreed with Dunbar [6]'s prediction about the size of human community.

In this paper, we formulate the view selection problem in a feed-following system and we propose several techniques to address the challenges. We propose a method to divide the set of feeds followed by each user into candidate views by taking the benefits of their materializations into account. By generating candidate views in this way, our optimization algorithm can guarantee that materializing a candidate view would reduce the cost of producing the personalized view for at least one user. With this guarantee, materializing any of these candidate views

can be beneficial to the system performance. Another salient technique that we propose is building a transitive closure graph of the subset partial order relation among the candidate views. We then use a bottom-up search algorithm to traverse the graph and choose the set of views for materialization. In this bottom-up algorithm, a candidate view that has the highest potential to be used for generating personalized views for more users will be considered before the others. In summary, we make the following contributions in this work,

- We formulate the view selection problem in feed-following system and prove that it is NP-hard.
- We propose a practical cost model to estimate the benefit of maintaining a materialized view under feed-following system and compare the *Push* and *Pull* strategies with our cost model.
- We present a greedy algorithm that chooses the materialized views iteratively by using the cost models that we develop. We then propose to use the transitive closure graph of the subset partial order relation among the candidate views and present our hierarchical algorithm. To limit the size of candidate views, we adopt a heuristic to divide the feeds followed by each user into multiple sets as the candidate views.
- We implement a prototype system and evaluate our algorithms by comparing two state-of-the-art algorithms using both real datasets and synthetic datasets. The results show our methods significantly outperform the state-of-the-art algorithms in various situations.

II. Related Work

View selection in feed-following systems has been studied in existing work. Feeding-Frenzy [2] provides a view selection solution creating separate views for each user-feed pair. To avoid the exponential number of candidate views, they consider a candidate view for each user-feed pair. But as shown in our evaluation results, there is a lot of potential to improve system performance if we consider more candidate views and the sharing them between users. GeoFeed [7] propose a geographical feed-following view selection problem and present method that considers the sharing of views that containing one feed. Our method considers sharing views that containing more feeds and the experiments verify doing so can significantly reduce the system running cost.

View selection is one of the most challenging problems in data warehouses and is known to be NP-complete [8]. It has also been proved that view selection is inapproximable for general partial orders. This area has also been extensively surveyed in [9], [10], [11], [12]. This paper considers the view selection problem in the context of a feed-following system, which is also proved to be an NP-hard problem.

View selection in feed-following systems is also related to partial indexing and partial materialized views. Luo [13] proposes a partial materialization method to maintain frequently accessed results to minimize the response time of popular queries. Wu et al [14] present a partial index techniques that only made index for frequently accessed tuples while keeping others to be pulled from the sources. Aristides et al [15] present an algorithm pulling social contents from *hub* sites to reduce the maintenance cost and improvement social network feed following query efficiency.

Another related research area is multi-query optimizations. Mistry et al [16] attempt to make use of multi-query optimization techniques to do view selections. They find that there exist common subexpressions among multiple views, which could be shared by multiple queries to significantly decrease the system running cost. They generate each query's alternative plan and search for a multi-query plan exploiting common subexpressions to minimize the overall maintenance cost. Similar to this line of work, we also consider the feed-following graph to examine the sharing of common subexpressions and address the challenges specific to the feed-following context.

Feed-following can be considered as a special type of publish/subscribe service like [17], [18], [19], [20], [21]. These systems typically adopt a push approach to fan out events from producers to consumers, for example using distributed brokers or P2P techniques. While they consider the sharing of processing and communication among different pub/sub queries, we mostly focus on sharing the maintenance of materialized views. Mondal and Deshpande[22] proposed an optimization method considering the share of partial aggregates computation for continuous ego-centric aggregate queries, which is also an example of publish/subscribe system. Our hybrid push and pull approach can also be applied in the publish/subscribe context.

III. Problem Formulation

A. Feed-following model

A feed-following system consists of two main parties that deal with data streams: feeds and users. The goal of a feed-following system is to deliver aggregated events from all the feeds followed by users correctly and promptly.

A feed f_j is an event stream generator which periodically emits timestamped events to all of its followers. We denote the event stream generated by feed f_j as FE_j, which will emit events periodically to the users who follow f_j.

A user is an event consumer that receives aggregated events from all the feeds' streams that he follows with a selected ranking function, $Rank$. Without loss of generality, we assume $Rank$ assigns higher scores to events with higher priorities. The ranking function can be defined on, for example, the event's timestamp, popularity, importance and so on. A small number of ranking function will be provided by the server and user need to choose one from the available selections.

In addition to $Rank$, a user is also associated with an aggregate function, $Aggr$, which determines how events from different feeds should be aggregated to form the personalized view. We consider two types of aggregations:

- **Top-k.** A user will get the top-k ranked events from the feeds' streams that he follows. This could be defined as a feed-following query, denoted as,

$$\sigma_k(\bigcup_{\forall f_j \in F_{u_i}} FE_j) \tag{1}$$

where F_{u_i} is the set of feeds followed by u_i and σ_k is a function that selects the k highest ranked events

from a set of events and produces results sorted in descending order of their ranking scores.

- **Diversified top-k.** A user may also require that the top-k events should come from diversified feeds to avoid only getting results from very few feeds that generate very highly ranked events. To achieve that, we can limit the number of events from each feed to be at most t_j. Such a user query can be formally defined as,

$$\sigma_k(\bigcup_{\forall f_j \in F_{u_i}} \sigma_{t_j}(FE_j)) \tag{2}$$

In summary, a user query $Q(u_i)$ can be defined as a triple: $\langle F, Rank, Aggr\rangle$, where F is a set of feeds, $Rank$ and $Aggr$ are the ranking function and aggregate function respectively.

The feed-following relations can be defined as a bipartite graph $G(\mathcal{U}, \mathcal{F}, \mathcal{E})$, where a vertex $u_i \in \mathcal{U}$ has an edge $e_k \in \mathcal{E}$ to the vertex $f_j \in \mathcal{F}$ if u_i follows f_j. $\mathcal{U}$ and $\mathcal{F}$ are disjoint sets as they are separate roles in a feed-following system. Note that the feed-following bipartite graph is only a logical relation between users and feeds, and we need query plans processing all the users' feed-following queries over the feeds' event streams. Considering the different characteristics of feeds and users, an optimization framework is required to build an efficient query processing plan for all the users and feeds.

B. Feed-following query processing

We now consider query processing plans for users' feed-following queries. We assign each feed an update frequency ϕ_{f_j} that indicates how frequent it will emit a new event to its event stream. For each user, a query frequency θ_i indicates how often it will execute the aggregate query to get the results from all the feeds he follows is assigned as well.

Events from feeds will be stored in some materialized views within the feed-following system and delivered to the users periodically or when the users actively pose requests. Each materialized view v corresponds to one query, $Q(v) = \langle F_v, Rank, Aggr\rangle$, where F_v is the set of feeds whose event streams are ranked using function $Rank$ and aggregated using function $Aggr$, and v stores the query results of $Q(v)$. Upon the arrival of new events from a feed, the relevant materialized view has to be updated. Therefore, maintaining materialized views is resource consuming.

Theorem 3.1: A materialized view v can be used to answer the query of user u_i if $F_v \subseteq F_{u_i}$ and both $Q(u_i)$ and $Q(v)$ share the same aggregate and ranking functions.

For a user u_i, the subset of views that can be used to answer his query is denoted as V_{u_i}. An aggregation need to be done if multiple views are used to generate the result.

C. Cost Model

As shown in Theorem 3.1, views can be used for answering a user query if they share the same ranking and aggregate functions. Therefore, we first divide the user queries in to multiple partitions according to their ranking and aggregate functions. So within each partition, all the queries have same ranking and aggregate functions. Then we can optimize the selection of materialized views for each partition of users independently. Hereafter, we only consider one partition of users and assume all user queries have the same aggregate and ranking functions. For a user u_i following a set of feeds F_{u_i}, its query $Q(u_i)$ will be answered by a set of materialized views, denoted as V_{u_i}. The query plan of $Q(u_i)$ is just to aggregate the contents of all the views in V_{u_i}.

Given the query plan of $Q(u_i)$, the aggregate operation needs to collect top-k events from all the views in V_{u_i} to answer the user query. The amount of data need to be transferred and the size of the data need to be sorted is proportional to the number of views within a query plan. Therefore, the evaluation cost $EV(u_i, V_{u_i})$ is estimated as:

$$EV(u_i, V_{u_i}) = \begin{cases} \theta_i \sum\limits_{v_j \in V_{u_i}} L_j, & |V_{u_i}| > 1 \\ 0, & |V_{u_i}| = 1 \end{cases} \tag{3}$$

where θ_i is the query frequency of user u_i and L_j is the cost for transferring and sorting the top-k events from v_j to query processor, which depends on the geographical location of the data and the type of back-end system that stores the data. Note that in the case where V_{u_i} contains only one view, the aggregation cost is negligible.

Besides the query evaluation cost, the system need to maintain all the materialized views. We call the cost of updating the materialized views as maintenance cost. A materialized view v_i needs to be updated when new events are produced by any feed within F_{v_i}. We then define a materialized view v_i's update frequency as $\phi_i = \sum\limits_{f_j \in F_{v_i}} \phi_{f_j}$, where ϕ_{f_j} is the update frequency of f_j. By using H to denote the cost of updating a materialized view for each new event, we estimate the maintenance cost of v_i as follows,

$$M(v_i) = \sum_{f_j \in F_{v_i}} \phi_{f_j} H \tag{4}$$

In the above discussions, the cost we define can be considered as the consumption of a system resource, which could be CPU, disk I/O or network I/O. In practice, we can quantify the above cost model by considering the system's bottleneck resource and use it as the optimization goal. As discussed in previous work [2], the bottleneck resource depends on how the feed following system is implemented. In this paper, to fulfill the low-latency requirements of feed following applications, we assume the materialized views are stored in a distributed in-memory database and the system is run on a cluster of servers with sufficient main memory and a high-bandwidth network. In such a system, CPU is the system's major bottleneck resource. Optimizing the CPU usage of feed-following system will naturally improve the system performance that the potential throughput is higher. Therefore, we use CPU consumption as the optimization goal from now on.

D. Problem Statement and Hardness

A view selection plan $\mathbb{P}$ is 2-tuple $\langle \mathcal{V}, \mathcal{QP}\rangle$, where $\mathcal{V}$ is the set of views that are selected to be materialized in the system and $\mathcal{QP}$ are the optimized query plans for all the user queries. The cost of a view selection plan is the sum of maintenance cost of all the materialized views and evaluation cost of all the

users' queries. The cost of $\mathbb{P}$, denoted as $Cost(\mathbb{P})$, is defined as follows,

$$Cost(\mathbb{P}) = \sum_{v_i \in \mathcal{V}} M(v_i) + \sum_{u_j \in \mathcal{U}} EV(u_j, V^o_{u_j}) \quad (5)$$

where $V^o_{u_j}$ is the optimal query plan for user u_j using $\mathcal{V}$.

Now we can formally define the View Selection problem as follows. *Given a feed-following bipartite graph $G(\mathcal{U}, \mathcal{F}, \mathcal{E})$, the View Selection problem is to find a view selection plan $\mathbb{P} = \langle \mathcal{V}, \mathcal{QP} \rangle$, such that $Cost(\mathbb{P})$ is minimized and all the user queries can be evaluated using query plans in $\mathcal{QP}$ over the materialized views $\mathcal{V}$.*

The following theorem states that the View Selection problem is an NP-hard problem, which means we cannot develop an optimal algorithm that has a polynomial time complexity unless $P = NP$.

Theorem 3.2: View Selection is an NP-Hard problem.

Proof: We prove this by restriction. We first restrict the View Selection problem by ignoring the maintenance cost of the materialized views and by considering only a fixed set of materialized views. Then finding the optimal query plan for each user query is equivalent to finding the minimum subset of materialized views that can cover all the feeds followed by the user. In other words, it is equivalent to a minimum set cover problem, which is known to be NP-hard. ■

IV. View Selection Algorithms

A. System Overview

A typical architecture of a view selection enabled system is shown in Figure 1. The view selection planning component takes the feed-following bipartite graph, the query frequency of each user and the update frequency of each feed as inputs. The system will store each feed's event stream as a view for storage. This kind of view will be maintained whenever a new feed is added to the system and we call it the feed's native view. When executing a user query in the system, the query processor will calculate query results using query plan generated by the view selection component and route the query to the corresponding storage nodes. It needs to select a set of views that are maintained in the system with minimum overall cost.

B. Basic View Selection

For a given user u_i and his following set F_{u_i}, there are basically two query evaluation strategies for each feed followed by the user, called pull and push respectively.

Take the bipartite graph in Figure 2 as an example. There are 4 users each following a subset of the 5 feeds. The statistic of update and query frequencies are listed in the figure. We only have the native views maintaining each feed's events before we select more views to materialize. We need to find how to answer each user's query and what extra views we need to maintain.

Pull is a query-on-demand strategy. The system will generate the personalized view for a user only when an update of the view has to be generated and send to the user. The cost of generating the view on demand depends on the frequency of the user query and the number of sources being used to calculate the aggregated results.

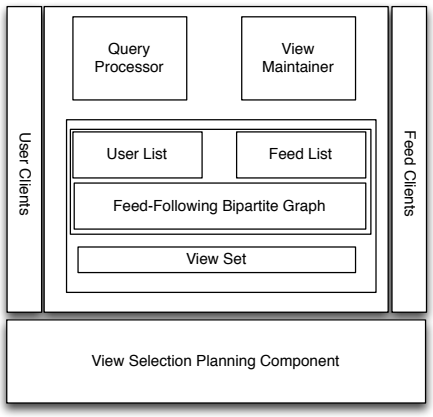


Fig. 1. The generic system architecture

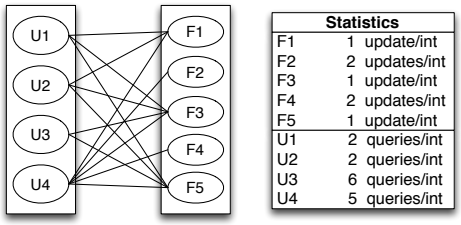


Fig. 2. Feed-following bipartite graph and statistics

Push is a query-on-update strategy. A user's personalized view is materialized and actively being updated upon each new event that arrives. The cost of a push strategy is only related to the frequency of event updates and independent on the user query frequency.

We can naturally introduce two basic view selection solutions for a given feed-following bipartite graph, which are the *PullALL* and *PushAll* algorithms. Considering our example relation in Figure 2, a *PullALL* algorithm will only use the native views and generate the personalized view for all users on demand. We assume all the views are stored in same type of node, so we have the same L_j for each nodes j and use a L to denote it. The total cost will be $49L+7H$ for *PullAll* algorithm under our cost model. *PushAll* needs to materialize each user's personalized view in addition to the native views. The total cost for *PushAll* algorithm will be $22H$. Both of the two algorithms do not consider the users' query frequencies and the feeds' update frequencies. They cannot provide a robust performance for different kinds of feed-following workload. We need to find algorithms to generate hybrid plans with both *Push* and *Pull* strategies.

C. Candidate View Generation

To obtain a hybrid plan, the first step is to generate a set of candidate views to be considered for materialization. Candidate view should be used by at least one user to ensure it has the potential capability to reduce overall evaluation and

maintenance cost. With such a requirements we can restrict the candidate views as those maintain aggregated events from subset of each user's following feed set. The complete set of candidate views is the power set of the following feeds for all the users, which is exponential to the maximum following size among all the users. Searching with such a large number of candidate views is prohibitively expensive. On the other hand, considering more candidate views could explore more chances of sharing and has the potential to generate a better plan.

Intuitively, we can use all the users' personalized view as the set of candidate views to limit the search space. Then the number of candidate views is limited to the number of users. For the example graph in Figure 2, we can generate 3 materialized views to serve all the users in Figure 3a.

However, only using the users' personalized views may not provide sufficient system performance. This is because the feeds that followed by a user may have different update frequencies, materializing them altogether is often not an optimal solution. To solve this problem, for a given set of feeds followed by a user, we estimate the profit of selecting each of them into a materialized view. Then we divide the feeds into two groups: *PushPrefer* and *PullPrefer*, where *PushPrefer* contains the all the feeds that are beneficial to be included in a materialized view and *PullPrefer* contains the rest. The profit value is estimated as follows.

According to Eqn (4), the materialization cost incurred by a feed f_j is $AM(f_j) = \phi_{f_j} H$, where ϕ_{f_i} is the update frequency of f_j. On the other hand, if f_j is not included in a materialized view that can be used by u_i, then the query evaluation cost for u_i incurred by pulling data from the native view of f_j is $AEV(u_i, f_j) = \theta_i L_j$, where θ_i is the query frequency of u_i and L_j is the cost of pulling data from the native view of f_j. Then we estimate the profit of including f_j into a materialized view to be used for u_i as $Profit(u_i, f_j) = \frac{AEV(u_i, f_j)}{AM(u_i, f_j)}$.

Our candidate view generation algorithm is presented in Algorithm 1. Lines 2–11 generate two candidate views from each user's following set: $PushPrefer$ and $PullPrefer$. For each user u_i and each f_j that he follows, if $Profit(u_i, f_j)$ is greater than 1, then f_j is potentially beneficial to be put in a materialized view and hence we put it in $PushPrefer$. Otherwise, f_j is put in $PullPrefer$. It is clear that materializing $PushPrefer$ may benefit u_i. Furthermore, materializing $PullPrefer$ would increase the total cost if we only consider u_i. However, if the view $PullPrefer$ can be shared by multiple users, then the total cost of the users sharing it may be reduced. In other words, $PullPrefer$ may still be beneficial if we consider global optimization and we should put it in the candidate views. Generating more candidate views may potentially improve the view selection result, but it will increase the computation complexity of the view selection algorithm.

In lines 12–21, we generate the candidate views and store their properties, including their maximal query frequencies, update frequencies and sets of potential users. The maximal query frequency is the sum of the query frequencies of all the users who may use this view and the update frequency is the sum of all the view's feeds' update frequencies.

Algorithm 1: Candidate view generation

Data: Feed-following bipartite graph $G(\mathcal{U}, \mathcal{F}, \mathcal{E})$
Result: Candidate view set CV

1 Initial HashMap Users(Feed,UserSet); //mapping a feed to the set of users following it
2 **foreach** $u_i \in \mathcal{U}$ **do**
3 Initialize PushPrefer and PullPrefer;
4 **foreach** $f_j \in F_{u_i}$ **do**
5 $Users[f_j].add(u_i)$;
6 **if** $Profit(u_i, f_j) \geq 1$ **then**
7 Insert f_j to PushPrefer
8 **else**
9 Insert f_j to PullPrefer
10 Insert PushPrefer to VG; Insert PullPrefer to VG;
11 **foreach** $F_k \in VG$ **do**
12 Create view v using F_k;
13 $v.queryFreq \leftarrow \theta_i$;//$\theta_i$ is the query frequency of u_i
14 $v.updateFreq \leftarrow \phi_k$;//$\phi_k$ is the sum of the update frequencies for all the feeds in F_k
15 $v.users \leftarrow \bigcap_{f_j \in F_v} Users.get(f_j)$; //all the users who can use this view
16 **if** $v_k \in CV \ \ \& \ \ v_k == v$ **then**
17 //if we already have a candidate view with the same set of feeds
18 $v_k.queryFreq+ = v.queryFreq$;
19 **else**
20 Add v to CV;
21 return CV;

D. Cost-based Greedy View Selection

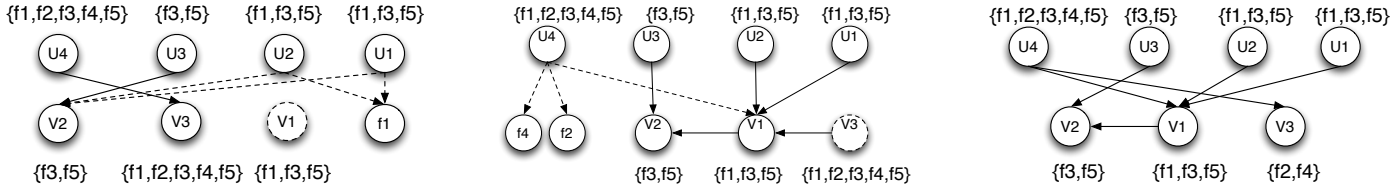
With the cost model for both push and pull strategies, we can develop a cost-based view selection algorithm. Note that the views of different users may share common feeds. Materializing a view for one user may also affect the query evaluation cost of other users. This means that view selections for different users are not independent.

As the view selection is an NP-hard problem, we need to find heuristic techniques to simplify the algorithm. Given the generated candidate views, we need to search for a subset of them to materialize and assigning optimal query processing plans to the users using the selected materialized views. A straightforward idea is to prioritize materializing views that brings higher benefits to the system performance. This Greedy algorithm is presented in Algorithm 2.

The benefit of a candidate view is quantified as the amount of cost reduction that can be obtained by materializing it. More precisely, the benefit of a view v_j is equal to the reduced amount of evaluation cost for all the queries that can use v_j subtracted by the maintenance cost of v_j.

Given a set of materialized views $\mathcal{V}$, the additional benefit of materializing v_j denoted as $B(v_j, \mathcal{V})$ can be defined as:

$$B(v_j, \mathcal{V}) = \sum_{u_i \in v_j.users} (EV(u_i, V_{u_i}) - EV(u_i, V'_{u_i})) - M(v_j) \quad (6)$$



(a) Greedy Algorithm (b) Hierarchy-1 Algorithm (c) Hierarchy-2 Algorithm

Fig. 3. Example View Selection Plans on Example Graph

Algorithm 2: Greedy Algorithm

Data: Feed-Following bipartite Graph $G(\mathcal{U}, \mathcal{F}, \mathcal{E})$
Result: View selection plan $\mathbb{P}(\mathcal{V}, UserPlan)$
1 $\mathcal{V} \leftarrow V_{\mathcal{F}}$; //native views as materialized views
2 Initialize HashMap UserPlan;//User id to ⟨QueryPlan,Cost⟩
3 Generate CV;
4 **foreach** $v \in CV$ **do**
5 **foreach** $u_i \in v.users$ **do**
6 //$v.users$ is defined in line 16 of Algorithm 1
7 Calculate ⟨QueryPlan,Cost⟩ and stored in UserPlan
8 v.b $\leftarrow B(v_j, \mathcal{V})$; //Assign materialization benefit
9 **if** *v.b* $\leq$ *0* **then**
10 Remove v from CV;
11 Sort CV by benefit in descending order;
12 **while** $CV \neq \emptyset$ **do**
13 v $\leftarrow$ Pop(CV); //Candidate view with the highest benefit
14 Add v to $\mathcal{V}$;
15 **foreach** $u_i \in v.users$ **do**
16 Update UserPlan[u_i] using $\mathcal{V}$
17 **foreach** $v_j \in CV$ **do**
18 **if** $F_v \subseteq F_{v_j}$ **then**
19 **foreach** $u_i \in v_j.users$ **do**
20 Update UserPlan[u_i]
21 $v_j.b \leftarrow B(v_j, \mathcal{V})$;//update v_j's benefit
22 **if** *v.b* $\leq$ *0* **then**
23 Remove v from CV;
24 Sort CV by benefit in descending order;
25 return $\mathbb{P}(\mathcal{V}, UserPlan)$;

where V_{u_i} and V'_{u_i} are the sets of materialized views chosen from $\mathcal{V}$ and $\mathcal{V} \bigcup v_j$, respectively, to evaluate the query of u_i. $EV(u_i, V_{u_i})$ and $M(v_j)$ are defined in Eq. (3) and Eq. (4).

One may notice from the above formula that the evaluation cost of u_i depends on the materialized views chosen to evaluate his query and, with a given set of materialized views $\mathcal{V}$, there may exist multiple possible query plans for each user query. Hence, to estimate the evaluation cost of a query with a given set of materialized views $\mathcal{V}$, we have to first generate an optimized query plan, i.e choosing an optimal set of materialized views from $\mathcal{V}$ to evaluate the query of each user u_i. This optimization problem is equivalent to find the minimum set cover for F_{u_i}, the following set of u_i, using the views in $\mathcal{V}$. The minimum set cover problem is a well-known NP-hard problem. We make use of the greedy minimum set cover algorithm, the best-possible polynomial time approximation algorithm [23].

Algorithm 2 first adds all the feeds' native views to the set of materialized views $\mathcal{V}$ (line 1) and then generates candidate views CV using algorithm 1 (line 3). We can then generate the initial query plan for each user, calculate the materialization benefits for all the candidate views and sort them by their benefits in decreasing order (line 4–11). Note that we can safely remove the views whose benefit values are negative (lines 9–10). This is because the benefit of a candidate view would only decrease when more views are materialized and the subsequent steps of the algorithm would not choose to materialize these views anyway.

In lines 13–14, we take the candidate view v that has the highest benefit from CV and put it in the set of materialized view $\mathcal{V}$. Then for each user u_i whose query evaluation cost can be reduced by putting v in its query plan, we use the new and cheaper query plan (lines 15-16). With v being materialized, the benefits of other candidate views that is a superset of v could be decreased because the potential amount of cost reduction incurred by materializing them is decreased. Therefore, we need to update their benefit values and re-sort the set of candidate views (lines 17–24). Again, we remove candidate views whose benefit values are less than 0, which will not be materialized (lines 22–23).

Algorithm Complexity. The complexity of running the greedy minimum set cover algorithm on F_{v_j} using $\mathcal{V}$ is $\mathcal{O}(n \cdot |\mathcal{V}|)$, where n is the size of F_{v_j}. Since the native views of all the feeds are maintained in $\mathcal{V}$, there should exist a cover for any F_{v_j}. The minimum set cover algorithm will be executed at most n times if a candidate views v_j need to be materialized. We can then derive the complexity of the greedy algorithm as $\mathcal{O}(m \cdot n_{max}^2 \cdot (f+m))$, where f is the number of feeds in $\mathcal{F}$, m is the number of candidate views in CV and n_{max} stands for the candidate view that contains the greatest number of feeds.

Example. We use the 3 candidate views in Figure 3a as an example, which are supposed to be generated without dividing each user's following list. We use the same back-end system as discussed in the former example to define the same L for each node. We can calculate the benefit using the statistics in Figure 2 for these 3 views as $12L-3H, 12L-2H, 25L-7H$ respectively. The H/L value may depend on the system being used to store the data and the selected user queries. It can be estimated by executing push and pull only queries on the given system. We have got a value of 2.83 in our system for top-k newest queries by linear regression using the result from 10 different number of random queries' CPU usages with only one push or pull operation. We can then sort the

views by their benefits in descending order as V_2, V_3, V_1. We materialize top-beneficial view V_2 first, then update the benefit of the remaining views V_3 and V_1 as $20L-7H$ and $8L-3H$. Now the remaining candidate views will be sorted again with the new benefit values and V_1 is removed from the candidate view due to its negative benefit value. This process goes on and, this time, choses to materialize V_3, the new top-beneficial candidate view. The total cost of this chosen plan is $16H+8L$ with the materialization of the native views, V_2 and V_3. In the query plans, $U1$ and $U2$ pull results from $V2$ and f_1's native view, and $U3$ and $U4$ get pushed results from $V2$ and $V3$ respectively.

E. Hierarchical View Selection

The candidate views may have subset partial order relation between each other, which can be used as heuristic information in the view selection process. We can calculate the transitive closure graph of the candidate views' subset relation. This graph contains all the subset partial order relations among all the candidate views. The transitive closure graph $G(CV, CV^+)$ is a directed acyclic graph, because we have combined views maintaining the same set of feeds. In the graph, each vertex represents a candidate view and each edge is the subset partial order relation between each pair of candidate views. For a given set of candidate views CV, the transitive closure graph is defined as $G(CV, CV^+) = \{(v_k, v_j) | \forall F_{v_j} \subseteq F_{v_k}, v_j, v_k \in CV\}$.

Note that materializing v_j may decrease the potential evaluation cost of v_k, but materializing v_k will not influence v_j's cost. Therefore, we may get a better plan if we consider v_j's materialization before v_k. Another observation is that if there is no path between two views in the transitive closure graph, the materialization decisions for these two views are independent from each other because there is no sharing possibilities here. With these two observations, we may be able to generate better optimization results if we enumerate the candidate views in an order that considers their subset relationship instead of only using the descending order of the materialization benefit as in the greedy algorithm.

We propose a traversal algorithm based on the idea of Depth-First Search (DFS), which is called Bottom-up Depth-First Search (BDFS) presented in Algorithm 3. A transitive reduction is done on the transitive closure graph to reduce the number of edges and it will not affect the final visiting order. We can get a unique visiting order of the candidate views, starting from a small view to its superset using the BDFS algorithm. The hierarchy algorithm runs similarly to the greedy algorithm except that it visits the candidate views in the order that is produced by the BDFS algorithm.

For the example inputs shown in Figure 2, we can generate the same 3 candidate views as described in the previous section under same system. However, unlike the greedy algorithm, the views are visited in the order of V_2, V_1, V_3. The different visiting order results in a different decision as shown in Figure 3b. Here we do not materialize V_3 because the pull cost of $U4$ is reduced after we materialize V_1. The total cost using the hierarchy algorithm is $12H+15L$, which is less than the greedy algorithm.

Furthermore, if we divide $U4$'s following set into two parts: f_1, f_3, f_5 which has a higher amortized benefit to materialize and f_2, f_4 which has lower benefit. We may have a different plan which materializes all the candidate views with the total cost $16H$ and the user query plan showing in Figure 3c using the Hierarchy algorithm. This cost is the minimum cost for all the plans presented in this section.

Algorithm 3: Bottom-up Depth-first Search

Data: Transitive closure graph of candidate views $G(CV, CV^+)$
Result: Ordered candidate view list CV_L

```
G(CV, CV^R) ← TransitiveReduction(G(CV, CV^+));
Stack S, List CV_L, HashMap Processed(View, Boolean) initialize;
Add all views CV to Processed, initial to false;
Sort Processed by F_{v_j} size in descending order;
foreach v_j ∈ Processed.KeySet do
    Push v_j to S;
    while S not empty do
        v_k ← S.peek();
        if Processed.get(v_k) = True then
            S.pop();
            if v_k ∉ CV_L then CV_L ← v_k; ;
        else
            Processed.put(v_k, True);
            foreach (v_k, v_r) ∈ G(CV, CV^R) do
                S.push(v_r);
return CV_L;
```

In addition, in the greedy algorithm, when we attempt to materialize a candidate view, the transitive closure graph can be used to find the views that could benefit from the materialization of the current view. In this way we can visit a smaller amount of views in each iteration, which reduces the complexity. However, when there is no subset relations among the candidate views, the hierarchy algorithm will select the same views as the greedy algorithm. The complexity of transitive closure mainly depends on the number of arcs in graph [24] and the worst case complexity is $\mathcal{O}(n^3)$ (n is the number of vertex in graph, i.e. the number of candidate views), which is much lower than the user query planning process in the view selection problem. Compared to the greedy set cover complexity in hierarchy algorithm, generating the new searching order will not increase the complexity of the hierarchy algorithm a lot.

V. Evaluation

A. Experiment Setup

Datasets. We conduct our experiments on both synthetic and real dataset. Synthetic dataset is generated using the Apache Math library. We create multiple synthetic datasets using different parameters for Zipfian distributions to generate various scenarios to examine our algorithms' sensitivity to different input parameters. We assume the generated events sizes are the same among all the feeds.

We also use a real world dataset from Google+ containing a sample snapshot of a real feed-following system [25], where each user follows some other users' feeds. This dataset only

contains the user relation graph and there exists a lot of users whose following set is not fully available. We read all the edges in the user relation graph and transform them into edges in a bipartite graph. Then we take the starting vertex of each edge as a feed and the rests as users to obtain a feed-following graph. In total, the dataset contains 99679 users and 72271 feeds. The average number of feeds per user is 137.1 and through analysis, we find that it closely follows a Zipfian distribution with a parameter of 0.98. The average number of users per feed is 189.1 and we find it closely follows a Zipfian distribution with a parameter of 1.43. We assign update frequency and query frequency using the parameters in Table I, which have the same distribution in [2]. In addition, we use top-k events aggregation with a ranking function that sorts events chronologically.

	Amount	Zipfian Parameter
Feeds per user	6.5	0.62
Update frequency of each feed	0.1 news/interal	0.57
Query frequency of each user	0.5 queries/interval	0.62

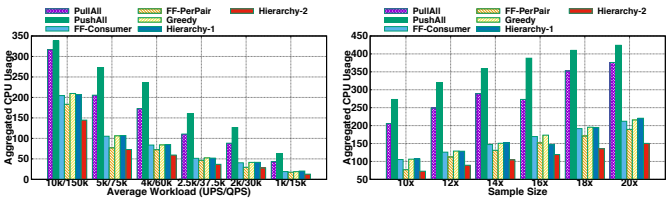
TABLE I. PARAMETERS FOR BASELINE SYNTHETIC DATASET

Implementation and Cluster Hardware. We implement our prototype system using JAVA 1.7 and Redis 3.0.1 [26], an in-memory key-value store system, as the backend storage system. The experiments are conducted on a cluster of 8 servers with 2×2.66Ghz Intel X5550 CPUs and 48 GB RAM. 6 nodes running Redis are used as storage server. We also have one node as the view maintainer and another one as the query processor to simulate the feeds updating and user query separately.

Algorithms. We compare the following algorithms:

1) *PushAll.* Materialize views for all the users' following sets and all the user queries are evaluated using the push strategy.
2) *PullAll.* Only maintain the feeds' native views. Different users sharing a common following feed would share the native view of the common feed.
3) *FF-Consumer.* The algorithm proposed in Feeding Frenzy [2] with the setting of per consumer strategy.
4) *FF-PerPair.* This is similar to *FF-Consumer*, except that it makes use of the per-pair strategy, where the pull or push decision will be made for each pair of user and feed.
5) *Greedy.* The greedy algorithm presented in Algorithm 2. Since each data node is running the same database system, the basic pull cost L_i is the same on different nodes. Here we use users' full following sets to generate the candidate views.
6) *Hierarchy.* Our Hierarchy algorithm. The model parameters are the same as those used in the *Greedy* algorithm. We have *Hierarchy-1*, which generates candidate views using each user's full following set, and *Hierarchy-2*, which use our candidate view generation algorithm to generate the candidate views.

To obtain stabilized results, each algorithm under each parameter setting is run for 20 minutes. Since we consider a static feed-following graph, the optimization of each algorithm is done offline.



(a) Impact of System Load, Synthetic Dataset (b) Impact of User/Feed Size, Synthetic Dataset

Fig. 4. Scalability Experiments

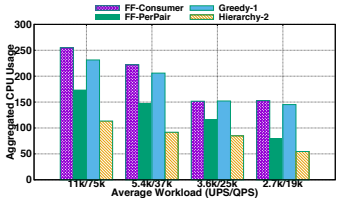


Fig. 5. Real Dataset Scalability Experiments

Metric. As we use an in-memory database with sufficient RAM on each server and a high speed network, CPU is the major bottleneck. The performance metric we use is the total CPU usage of all the server nodes in the cluster, which is collected by using the *sar* command in Linux. Note that a lower CPU usage indicates a higher system throughput.

B. Running Time

The results of the synthetic dataset are presented in Table II. *PullAll* and *PushAll* have the lowest running time due to its simplicity. *FF-Consumer* and *FF-PerPair* have similar searching times since they only consider each user's own following set and make decisions locally for each user. *Greedy*, *Hierarchy-1* and *Hierarchy-2* consume more searching time because they need to consider the dependency of different views to generate query plans for users. *Hierarchy-1* is more expensive than *Greedy* as it need to find the subset partial order relations.

C. Load Levels

We then simulate the different load levels by scaling the time unit under the same query and update frequency setting. We use an interval timer to simulate the feed-following system's workload. For a given dataset, changing the interval size is equivalent to varying all feeds' update frequency, UPS (update per second), and all users' query frequency, QPS (query per second) simultaneously. We use the same basic synthetic dataset in the previous experiment. The experiment results are presented in Figure 4a. We find that our cost model successfully estimates the actual workload of the system. Our *Greedy* and *Hierarchy-1* achieve similar performance as *FF-Consumer* and our *Hierarchy-2* achieves a good performance with an average CPU usage reduction of 16% compared to *FF-PerPair*. *Hierarchy-2* algorithm can achieve much more performance improvements when the workload is high. The improvement of *Hierarchy-2* persists even when the system load is low since the candidate view generation algorithm provides higher possibility to share views among users.

Algorithm	PullAll	PushAll	FF-Consumer	FF-PerPair	Greedy	Hierarchy-1	Hierarchy-2
Search Time (ms)	76	62	189	189	858	922	1080

TABLE II. ALGORITHM RUNNING TIME FOR SYNTHETIC DATASET

D. Data Sizes

We also examine our algorithms' performance under different dataset sizes by varying the size of the synthetic data while keeping the other parameters unchanged. We generate a basic dataset with $10,000$ feeds and $30,000$ users and then a series of datasets which is 20% larger than its antecedent. We denote the basic dataset size as 10x, and the subsequent ones as 12x, 14x, 16x, 18x and 20x respectively. We use the load level as 5k UPS (update per second) and 75k QPS (query per second) in these experiments.

From Figure 4b, we can find that all the samples have a better performance with *PullAll* than *PushAll*. Take the 16x results as an example, *Hierarchy-1* has a reduction of 16% CPU usage compared to *Greedy* and 13% to *FF-Consumer*. *FF-PerPair* has a similar performance as *Hierarchy-1* due to the fact that it makes separate decisions on each feed for each user. This shows the importance to split each user's following set while we are generating candidate views. This can be clearly verified by the results of *Hierarchy-2*. *Hierarchy-2* has the best performance under all the data sizes. Over all the experiments, it achieves on average an 18% reduction on the average CPU usage compared to *FF-PerPair* and 31% to *Hierarchy-1*. This indicates that our candidate view generating algorithm provides more opportunity to minimize the system cost and hence can achieve more robust performance with different data sizes.

E. Impact of Skewness

In this subsection, we examine the impact of the skewness of the size of each user's following set and the number of each feed's follower. We run two sets of experiments varying the parameter of the zipfian distributions. We use the same average number in Table I but change the zipfian parameters to see the sensitivity of the algorithms. We use the basic data size described above, 4k UPS and 60k QPS as adopted.

Figure 6a presents the results on the skewness of the size of each user's following set. All the advanced optimization algorithms can achieve better performance comparing to the two basic ones, while *Hierarchy-2* is the winner under all the situations. This indicates that *Hierarchy-2* is robust to the skewness of the users' following set.

The results on the skewness of the number of each feed's follower are presented in Figure 6b. *Hierarchy-1* can achieve a better performance with a low skewness because lower skewness results in more levels in the subset partial order graph. The Hierarchy algorithm takes advantage of the complex graph structure and produces a better plan. *Greedy* algorithm is getting worse when the skewness is low because the searching order does not consider future sharing possibility of a materialized view. *FF-Consumer* and *FF-PerPair* perform similarly since they only consider individual user's following relationship, and hence the skewness of the feed popularity will not affect the decisions. *Hierarchy-2* performs overall the best while being robust to skewness.

F. Impact of query/update ratio

In this experiment, we examine the cases with different query/update ratios, i.e. the ratio of user querying frequency to feeds' updating frequency. This ratio affects the average performance of pull and push strategy. We use 10x sample size and 4k UPS and 60k QPS load level in these experiments. To vary the query/update frequency ratio, we keep the query frequency unchanged and varying the average update frequency. The results are presented in Figure 6c. We can see that when the query/update frequency ratio increases, the push strategy becomes a better choice and all the other algorithms except *PullAll* get a better performance. Again, *Hierarchy-2* has the best overall performance and its edge over the other algorithms increases with the increase of the ratio. This is because it can materialize more views for sharing with a larger ratio. On the other hand, when the update frequency is much greater than the query frequency, *PullAll* has a relatively good performance and the improvements achieved by the other algorithms are less significant. This is understandable because the lower of the query/update ratio, the less benefit can be achieved by materializing views.

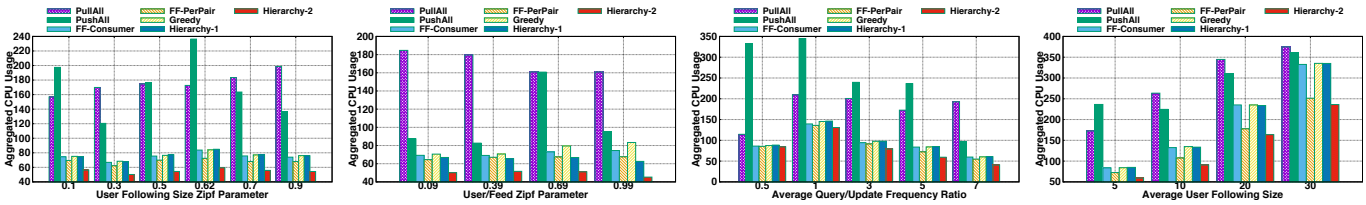
G. Impact of Following Size

We also examine the algorithms under different user following size. When users follow more feeds, it is more expensive to use a user's complete following set as candidate views. We use the basic 10x dataset under 4k UPS and 60k QPS load level but varies the average size of user following set. From the results in Figure 6d, we can see that *Hierarchy-2* can achieve a better improvement over *Hierarchy-1* with larger following sets. *FF-PerPair*'s performance gap between *FF-Consumer* is also larger when following set becomes larger. This is as expected because the larger the following set of each user, the more improvement can be achieved by sharing the views among users.

H. Real Dataset Experiments

Besides the synthetic dataset, we also run the experiment of 4 algorithms on the real dataset. We vary the interval size in a similar way as stated above to change the system load level. Due to the bad performance of *PushAll*, *PullAll* and *Hierarchy-1*, we disregard them in this subsection for brevity. As shown in Figure 5, the results are quite similar to those of the synthetic datasets. Since the average number of followers of each feed is much larger than the synthetic dataset (about 8 times larger), the possibility of sharing materialized views is larger. We can see our *Hierarchy-2* algorithm uses 32.5% less CPU compared to *FF-PerPair* on average. From the results of the real dataset experiments, we can verify that *Hierarchy-2* has the overall best performance when there are more users and feeds in the system.

We also collect the running time of the algorithms, which is shown in Table III. While *FF-PerPair* and *FF-Consumer* run faster than *Hierarchy-2*, *Hierarchy-2* can still complete in a



(a) Sizes of the Following Sets (b) Feed's Popularity (c) Query/Update Freq. Ratio (d) Avg. User Following Size

Fig. 6. Varying Parameters of the Synthetic Dataset Experiments

Algorithm	FF-Consumer	FF-PerPair	Greedy	Hierarchy-2
Search Time (s)	8	374	87,399	8,500

TABLE III. ALGORITHM RUNNING TIME FOR GOOGLE+ DATASET

reasonable time and produce a much better plan. Furthermore, we can see that Greedy is ten times slower *Hierarchy-2*, but its plan is worse than *Hierarchy-2*.

VI. CONCLUSION

In this paper, we present the view selection problem in feed-following systems and formally formulate the optimization problem. We present a cost model to measure the potential system load with different view selection plans. We observe that view selections for different users are not independent and we need to consider the possible sharing among users for a better view selection plan. We propose a candidate view generation algorithm that can generate potentially beneficial candidate views and present a greedy and a hierarchical algorithms. We conduct comprehensive experimental evaluation on our algorithms by comparing them to state-of-the-art solutions. The results show that our hierarchical algorithm with our candidate view generation algorithm can achieve the best performance under all of the tested situations, especially with the real datasets.

REFERENCES

[1] Number of monthly active facebook users, http://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/.

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

[3] Beevolve. An exhaustive study of twitter users across the world, http://www.beevolve.com/twitter-statistics/.

[4] R. Chirkova, "The view-selection problem has an exponential-time lower bound for conjunctive queries and views," in *PODS*, 2002.

[5] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Statistical properties of community structure in large social and information networks," in *Proceedings of the 17th international conference on World Wide Web*. ACM, 2008, pp. 695–704.

[6] R. Dunbar and R. I. M. Dunbar, *Grooming, gossip, and the evolution of language*. Harvard University Press, 1998.

[7] J. Bao, M. F. Mokbel, and C. Chow, "Geofeed: A location aware news feed system," in *ICDE*, 2012.

[8] H. J. Karloff and M. Mihail, "On the complexity of the view-selection problem," in *PODS*, 1999.

[9] I. Mami and Z. Bellahsene, "A survey of view selection methods," *SIGMOD Record*, 2012.

[10] C. A. Dhote and M. S. Ali, "Materialized view selection in data warehousing," in *ITNG*, 2007.

[11] A. Y. Halevy, "Answering queries using views: A survey," *VLDB J.*, 2001.

[12] A. Silberstein, A. Machanavajjhala, and R. Ramakrishnan, "Feed following: The big data challenge in social applications," ser. DBSocial '11, 2011.

[13] G. Luo, "Partial materialized views," in *ICDE*, 2007.

[14] S. Wu, J. Li, B. C. Ooi, and K. Tan, "Just-in-time query retrieval over partially indexed data on structured P2P overlays," in *SIGMOD*, 2008.

[15] A. Gionis, F. Junqueira, V. Leroy, M. Serafini, and I. Weber, "Piggybacking on social networks," *VLDB*, vol. 6, no. 6, pp. 409–420, 2013.

[16] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham, "Materialized view selection and maintenance using multi-query optimization," in *SIGMOD*, 2001.

[17] Y. Zhou, A. Salehi, and K. Aberer, "Scalable delivery of stream query result," *VLDB*, 2009.

[18] A. M. Ouksel, O. Jurca, I. Podnar, and K. Aberer, "Efficient probabilistic subsumption checking for content-based publish/subscribe systems," in *Middleware*, 2006.

[19] Y. Zhou, B. C. Ooi, K.-L. Tan, and F. Yu, "Adaptive reorganization of coherency-preserving dissemination tree for streaming data," in *22nd International Conference on Data Engineering (ICDE'06)*. IEEE, 2006, pp. 55–55.

[20] Y. Zhou, B. C. Ooi, and K.-L. Tan, "Disseminating streaming data in a dynamic environment: an adaptive and cost-based approach," *The VLDB Journal*, vol. 17, no. 6, pp. 1465–1483, 2008.

[21] Y. Zhou, Z. Vagena, and J. Haustad, "Dissemination of models over time-varying data," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 4, 2011.

[22] J. Mondal and A. Deshpande, "Eagr: Supporting continuous ego-centric aggregate queries over large dynamic graphs," in *SIGMOD*. ACM, 2014, pp. 1335–1346.

[23] U. Feige, "A threshold of ln n for approximating set cover," *J. ACM*, 1998.

[24] Y. E. Ioannidis and R. Ramakrishnan, "Efficient transitive closure algorithms." in *VLDB*, vol. 88, 1988, pp. 382–394.

[25] J. J. McAuley and J. Leskovec, "Learning to discover social circles in ego networks," in *NIPS*, 2012.

[26] S. Sanfilippo and P. Noordhuis, "Redis," 2010.