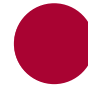
View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Copenhagen University Research Information System

UNIVERSITY OF COPENHAGEN



# Location-Centric View Selection in a Location-Based Feed-Following System

Chen, Kaiji; Zhou, Yongluan

*Published in:*
Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems, DEBS 2019, Darmstadt, Germany, June 24-28, 2019.

*DOI:*
10.1145/3328905.3329512

*Publication date:*
2019

*Citation for published version (APA):*
Chen, K., & Zhou, Y. (2019). Location-Centric View Selection in a Location-Based Feed-Following System. In *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems, DEBS 2019, Darmstadt, Germany, June 24-28, 2019.* (pp. 67-78). Association for Computing Machinery. https://doi.org/10.1145/3328905.3329512

Download date: 10. okt.. 2020

# Location-Centric View Selection in a Location-Based Feed-Following System


Kaiji Chen
Huawei Technologies, China
chenkaiji@huawei.com

Yongluan Zhou
University of Copenhagen, Denmark
zhou@di.ku.dk



## ABSTRACT

Location-based feed-following is a trending service that can provide contextually relevant information to users based on their locations. In this paper, we consider the view selection problem in a location-based feed-following system that continuously provides aggregated query results over feeds that are located within a certain range from users. Previous solutions adopt a user-centric approach and require re-optimizations of the view selection once users move their locations. Such methods limit the system's scalability to the number of users and can be very costly when a substantial number of users move their locations. To solve the problem, we propose the new concept of location-centric query plans. In this approach, we use a grid to partition the space into cells and generate view selection and query processing plans for each cell, and user queries will be evaluated using the query plans associated with the users' current locations. In this way, the problem's complexity and dynamicity is largely determined by the granularity of the grid instead of the number of users. To minimize the query processing cost, we further propose an algorithm to generate an optimized set of materialized views to store the aggregated events of some feeds and a number of location-centric query plans for each grid cell. The algorithm can also efficiently adapt the plans according to the movement of the users. We implement a prototype system by using Redis as the back-end in-memory storage system for the materialized views and conduct extensive experiments over two real datasets to verify the effectiveness and efficiency of our approach.


## CCS CONCEPTS

• **Information systems** → *Application servers*; *Location based services.*


## KEYWORDS

feed following, query optimization, view selection

**ACM Reference Format:**
Kaiji Chen and Yongluan Zhou. 2019. Location-Centric View Selection in a Location-Based Feed-Following System. In *DEBS '19: The 13th ACM International Conference on Distributed and Event-based Systems (DEBS '19), June 24–28, 2019, Darmstadt, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3328905.3329512



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



*DEBS '19, June 24–28, 2019, Darmstadt, Germany*



© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.



ACM ISBN 978-1-4503-6794-3/19/06…$15.00
https://doi.org/10.1145/3328905.3329512


## 1 INTRODUCTION

In a location-aware feed-following system, the users are interested in receiving information relevant to their current spatial contexts. Such systems can be characterized by a number of news feeds with spatial properties and a number of moving users who would like to receive the latest messages from their nearby feeds. Such a system has a wide application, ranging from social networks to mobile games. Below are two motivating examples based on real applications.

*Example 1. Ingress [18] is a location-based augmented reality mobile game. The game's objective is to capture portals, which are allocated across the world. Each portal continuously generates update messages, which include advertisements attached to the portals and each portal's status of being captured or not. Periodically, each user needs to receive the update messages from the portals located within a constant range from his current location. A location-based feed-following system can be applied in such an application by setting the portal updates as news feeds and letting the moving users to follow the feeds within a constant range from their locations.*

*Example 2. Vehicle-to-everything (V2X) [7, 23] communication enables the exchange of information among vehicles and infrastructures, such as vehicles' positions and speeds and traffic light status. The V2X technology can be used for improving road safety, increasing efficient flow of traffic, reducing environmental impacts and provide additional traveler information services, etc. A location-based feed-following systems can be used in many V2X applications. For example, each vehicle can subscribe to the status of the nearby traffic lights and the surrounding vehicles to calculate the best route and speed.*

In a location-based feed-following system, when a user logs in or refreshes his/her view, a query is issued and the latest updates from nearby news feeds are retrieved and displayed. Previous works on feed-following systems, such as Feeding-Frenzy [22] and GeoFeed [1], have studied how to optimize a large number of feed-following queries. There are basically two query processing strategies. In a *pull* strategy, the query result is produced on the fly. Given the latest user location, nearby messages are retrieved with the help of a spatial index. In contrast, a *push* strategy maintains a materialized view for a user by pre-computing the query results. When the user triggers a new query, the materialized view is delivered to the user. Hence, the *push* strategy is cheaper if the user logs in or refreshes relatively frequently in comparing to the updates of the news, while the pull strategy is better for the opposite case. The methods proposed in Feeding-Frenzy [22] and GeoFeed [1] optimize queries by choosing between a pull or a push plan for each user-feed pair.

In this paper, we enhance the usability of location-based feed-following systems by considering *mobile users* whose news feeds are *aggregated from multiple sources* to fulfill the requirements of

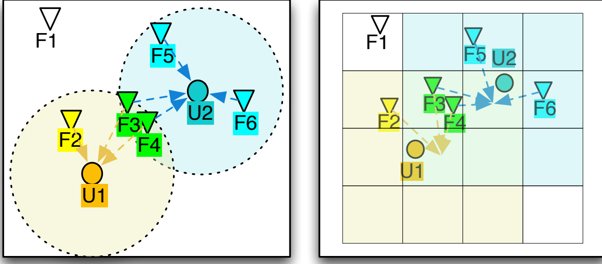



**Figure 1: Examples Feed-Following Relation**

the aforementioned applications. There are two distinctive features that differentiate our work from previous works. First, the existing methodologies, such as Feeding-Frenzy [22] and GeoFeed [1], fail to work well in the context of mobile users. The *push* strategy in GeoFeed assumes a static user location. If the location of a user updates, the materialized view has to be invalidated and reconstructed by employing a *pull* approach to retrieve new results. For example, as depicted in Figure 1, the feeds followed by user $Ui$, denoted by $F_{Ui}$, are dependent on $Ui$'s location. At time $t_1$, we have $F_{U1} = \{F2, F3, F4\}$ and $F_{U2} = \{F3, F4, F5, F6\}$, while at $t_2$, $U1$ moves to the location of $U2$ and the original view of $U1$ becomes invalid. We need to create a new view based on the new location of $U_1$, which incurs continuous overhead if the user is moving continuously. Note that, from the point of view of the whole system, even if each user moves relatively slowly over time, the update cost of the materialized views and query plans would still be significant as long as a significant portion of users who issue queries are moved (see the further analysis at Section 3.1).

Second, our model supports news aggregator from multiple feeds, while the existing location-based feed following systems like GeoFeed [1] are designed to retrieve $k$ most recent messages from each feed without further aggregation. On one hand, the aggregation feature provides opportunities for sharing the materialized views of aggregated results among multiple nearby users. On the other hand, it brings additional complexity to the view selection problem due to the large number of possible aggregated views.

Our main objective is to produce optimal query plans, i.e., the materialization strategies, for the moving users. We refer to the strategies in previous work, such as [1, 22] *user-centric*, because they generate query plans at the user level. In a user-centric strategy, once a user moves to a new location, a new query plan has to be generated, which is infeasible for systems with frequently moving users. To efficiently support view maintenance for mobile users, we propose a new paradigm which is *location-centric*. In particular, we split the space into grid cells and generate query plans for each cell. Figure 1 illustrates an example to demonstrate the superiority of a location-centric strategy over a user-centric one. When $U1$ moves to the location of $U2$, the query plan previously maintained for $U2$ can be re-used by $U1$ and there is no need to create new views for $U1$, as done in a user-centric approach.

To generate plans for each cell, a global optimization is performed based on the statistics of user popularities and their subscriptions on each cell. The rationale is that such statistics are relatively stabler than the location of each individual user. The plans will only be updated when such statistics are changed significantly. In the example shown in Figure 1, $F3$ and $F4$ are located at the same grid cell and they will be followed by $U1$ and $U2$. We can generate a materialized view containing aggregated events from both $F3$ and $F4$ to reduce the query cost of users whose query ranges cover the grid cells containing $F3$ and $F4$. The cell size should be determined by the location accuracy that the system provides.

In summary, we make the following contributions:

- We formulate the dynamic view selection problem in a new location-based feed-following system.
- We use a grid structure to characterize location-based feed-following queries and user movements. The query plan is generated and stored at the cell level.
- We propose a practical cost model to estimate the benefit of maintaining materialized views in a feed-following system and compare the *Push* and *Pull* strategies using our cost model. We show that materialized views can be used to reduce the pull cost of user queries. The analysis show that materialized view selection for individual users are correlated and hence, to choose an optimal set of materialized views, one should perform a global optimization by taking all users into account.
- We present a *Composite-view* algorithm that chooses the materialized views iteratively by using the cost models that we develop. To deal with changes of user statistics, the *Composite-view* algorithm is designed to be able to progressively optimize the current plan by adding beneficial and removing non-beneficial views according to the current statistics.
- We implement a prototype system that uses Redis [20], an open source in-memory data store, for storing the materialized views. We evaluate our algorithms using two real datasets by comparing with the state-of-the-art methods. The results show that our methods significantly outperform the state-of-the-art algorithms in various situations.

# 2 PROBLEM FORMULATION

## 2.1 Location-Based Feed-Following System

A location-based feed-following system consists of a set of feeds $\mathcal{F}$ and a set of users $\mathcal{U}$. Each feed $f \in \mathcal{F}$ is an event producer that is located at a time-varying position $f.pos$ and generates new events with an expected frequency $f.\phi$. Each user $u \in \mathcal{U}$ is modeled as a moving object with a time-varying position $u.pos$ and can subscribe to a set of feeds $F_u$ within a user specified query range $r$:

$$F_u = \{f | d(f.pos, u.pos) \leq r, \forall f \in \mathcal{F}\} \quad (1)$$

where $d(f.pos, u.pos)$ is a configurable distance function.

When a user requires an update for the latest events from the subscribed feeds, a *user query* is issued, denoted as $Q_u$. The system returns aggregated events from $F_u$ sorted by a ranking function $\sigma$, which can be a function upon one or more attributes (such as timestamp, popularity, importance and so on). Our system allows the application to pre-define a few ranking functions and each user can then chose the $\sigma$ from the provided options.

In addition, our system supports aggregating the events from multiple feeds. Let $f.S$ be the news stream generated by a feed $f$, and $\sigma_k$ be the top-$k$ events sorted by the ranking function $\sigma$. Two types of aggregation function are considered:

- **Top-k Aggregation** [22]. A user receives top-k sorted events from all the feeds that he/she follows:

$$Q_1(F_u) = \sigma_k(\bigcup_{\forall f \in F_u} f.S) \tag{2}$$

- **Diversified Top-$k$ Aggregation.** When $\sigma$ is customized as a function sorted by timestamps in descending order, the user is interested in top-$k$ recent events. If a feed has very high update frequency, it is likely that the user will always receive events from this feed. To avoid such a case, we can employ a simple diversified top-$k$ aggregate to allow at most $t$ events ($t \leq k$) from each feed in the aggregate function:

$$Q_2(F_u) = \sigma_k(\bigcup_{\forall f \in F_u} \sigma_t(f.S)) \tag{3}$$

$Q_2$ can be considered as a general case of $Q_1$. When $t = k$, it reduces to $Q_1$.

## 2.2 Query Processing

To process the user queries in a location-based feed-following system, there are two correlated optimizations: 1) determining the materialized views to store the aggregated events, and 2) generating a query plan for each user.

**Materialized Views.** A materialized view $v$ is a 2-tuple $\langle Q, F\rangle$, which is a dynamic dataset that contains the current results of the continuous query $Q$ , which is either $Q_1$ (Eqn. 2) or $Q_2$ (Eqn. 3). $v$ will be continuously updated upon updates of the feeds in $v.F$. We denote the whole set of materialized views as $\mathcal{V}$. $v \in \mathcal{V}$ can be used to answer the query of a user who follows all the feeds in $v.F$. Since each user's following feeds are related to his location, we need to decide the views that can be used by a user dynamically.

**Query Plans.** To optimize a user query, we have to decide the data access path, i.e., the set of materialized views to retrieve the necessary data. Therefore, we define the query plan of a user $u$ as a set of materialized views, denoted as $V(u)$, such that $F_u = \bigcup_{v \in V(u)} v.F$ and $\forall v \in V(u), v.Q = Q_u$. If $V(u)$ contains multiple materialized views, then an aggregation is needed to produce the final output, as defined in Eqn. 2 and Eqn. 3. Once a user moves to a new location such that his following feeds are changed, then a new query plan has to be used.

In summary, the overall optimized plan $\mathbb{P}$ is a 2-tuple $\langle \mathcal{V}, \mathcal{QP}\rangle$, where $\mathcal{V}$ is the set of materialized views and $\mathcal{QP}$ is the set of query plans for all the user queries.

## 2.3 Cost Model

We quantify the workload of processing a user query and maintaining a materialized view using a cost model to provide a performance estimation of a given plan $\mathbb{P}$.

Consider a user query $u_i.query$ that aggregates top-k events from all the feeds within range $u_i.r$ using $V(u_i)$, the query is evaluated by aggregating all the views in $V(u_i)$. The aggregate operation can be omitted if $V(u_i)$ contains only one view. Therefore, we can define

| Notation | Explanation |
|---|---|
| $\mathcal{F}$ | The set of all feeds in system. |
| $\mathcal{U}$ | The set of all users in system. |
| $\phi$ | The update frequency of a feed or a view. |
| $\theta$ | The query frequency of a user or a view. |
| $F_{u_i}$ | A set of feeds followed by user $u_i$. |
| $F_p$ | The set of feeds followed by users at grid cell $p$ |
| $v.F$ | The set of feeds of materialized view $v$. |
| $V(u_i)$ | The query plan of user $u_i$'s following query. The query evaluation can be done by aggregating all the views in $V(u_i)$. |
| $V(p)$ | The query plan at a grid cell $p$. It can be used to answer user query executed at $p$. |
| $L$ | The cost of retrieving and sorting the top-k events from a view $v_j$. |
| $H$ | The cost of updating a materialized view for each new event. |
| $\mathcal{V}$ | The set of materialized views. |
| $\mathcal{QP}$ | The query plans of all the user queries. |
| $\mathbb{P}$ | The complete optimized plan containing both $\mathcal{V}$ and $\mathcal{QP}$. |
| $\mathcal{S}$ | A map from a grid cell to the sum of the frequencies of user queries executed at the cell. |
| $\mathcal{H}$ | A map from a grid cell to the set of feeds located at the cell. |

**Table 1: Frequently used Notations**

the query evaluation cost $EV(u_i.\theta, V(u_i))$ as follows,

$$EV(u_i.\theta, V(u_i)) = \begin{cases} u_i.\theta \cdot |V(u_i)| \cdot L, & |V(u_i)| > 1 \\ 0, & |V(u_i)| = 1 \end{cases} \tag{4}$$

where $L$ is the cost of retrieving and sorting the top-k events from a view $v_j$, which depends on the back-end system storing the view.

Besides query evaluation, the system also needs to update the materialized views when new events arrive, which is referred to as the maintenance cost. A materialized view $v_i$ needs to be updated when new events are produced by any feed within $v_i.F$, so we can define $v_i$'s update frequency as,

$$v_i.\phi = \sum_{f_j \in v_i.F} f_j.\phi \tag{5}$$

By using $H$ to denote the cost of updating a materialized view for each new event, we define the maintenance cost of $v_i$ as:

$$M(v_i) = v_i.\phi \cdot H \tag{6}$$

The cost of an optimized plan $\mathbb{P}$ is the sum of the maintenance cost of all the materialized views and the evaluation cost of all the user queries:

$$Cost(\mathbb{P}) = \sum_{v_i \in \mathcal{V}} M(v_i) + \sum_{u_j \in \mathcal{U}} EV(u_j, Vu_j) \tag{7}$$

In general, the cost discussed above is considered as the system resource utilization, which can be CPU, disk I/O or network I/O depending on which resources being the bottleneck of the system. 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, Therefore, we use CPU utilization as the optimization goal from now on. In addition, CPU utilization is generally used as the main metric for measuring the energy consumption of a cluster [2]. Therefore, minimizing the CPU utilization would in general minimize the energy consumption of the system, which brings significant financial gains for the service provider.

### 2.4 Problem Statement

Now we can formally define the *dynamic query optimization* problem in a location-based feed following system. Given a set of continuously moving users $\mathcal{U}$ and a set of moving feeds $\mathcal{F}$ spread in a space $\mathcal{G}$, the dynamic query optimization problem is to dynamically generate a plan $\mathbb{P} = \langle \mathcal{V}, \mathcal{QP} \rangle$, such that $Cost(\mathbb{P})$ is minimized at any moment and the query of each moving user in $\mathcal{U}$ can be answered by a query plan in $\mathcal{QP}$ at any moment.

## 3 LOCATION-CENTRIC OPTIMIZATION

The optimizer takes $\mathcal{U}$ and $\mathcal{F}$ as the input and generates a set of materialized views and query plans. We divide the users into multiple groups by their query ranges, aggregate functions, and ranking functions such that each user group has the same query range, aggregate and ranking functions. We then generate an optimized plan for each user group. As we assume each user can only choose the query range, aggregate or ranking functions from a few pre-defined options, we expect there are a sufficient number of users in each group.

### 3.1 Motivation

There are two previous studies on feed-following systems that are very closely related to ours: Feeding-Frenzy [22] and GeoFeed [1].

Feeding-Frenzy [22] presents a method to make an optimization decision for each pair of user and feed. It adopts either a producer-pivoted view, which maintains the latest $k$ events from a feed followed by the user, or a consumer-pivoted view, which incrementally maintains the results of the user query. As indicated in [1], Feeding-Frenzy does not perform well in a location-based feed-following application. In addition, when the users move their locations, their following feeds will be changed and hence a new query plan and a new view has to be generated, which could be highly costly when a large number of users change their query locations.

GeoFeed [1] proposes a geographical feed-following system, where each event is associated with a location and a user only receives events whose locations are within a range from the user's position. The query optimizer considers the sharing of materialized views among users, but to simplify the optimization problem, only views containing a single feed are considered. They propose three possible query plans for each user-feed pairs: pull, push and shared push. Multiple query plans will be generated for the same user if the user has multiple locations. This will again be very costly if a user have many possible locations and move arbitrarily.

Note that even if each user does not move frequently, as long as a certain portion of user queries require new query plans, the overhead of creating new views and new plans would be significant. For instance, suppose there are 1 million active users at the system and each user issues a query every 10 minutes and changes his location every 50 minutes. Then we have on average 10 thousand user queries per minute (1 million/10 minutes), and on average 1/5 of them (10 minutes/50 minutes) issue queries at locations different from last time. This means 20% of the queries would incur extra overhead of creating new views and new plans, which would be significant.

We can categorize the above approaches into the user-centric paradigm in the sense that they generate plans for each individual user. Such approaches are inevitably expensive in a large-scale system with a large number of users, which may move arbitrarily. The number of plans are proportional to the number of users and the possible locations of the users.

To address the problem, we propose a location-centric optimizer, which partitions the space into grid cells and generates query plans for each cell. In this way, we can limit the optimization complexity by the number of grid cells. In general, a coarser-grained grid partitioning can reduce the complexity while providing less accurate spatial range query results. Therefore, the grid granularity should be chosen by the application's requirements on spatial accuracy.

### 3.2 Location-Centric Query Plans

For the ease of presentation, we first assume the feeds' locations are static and then extend our solution to mobile feeds in Section 3.5.

To generate location-centric query plans, we project the 2-D space $\mathcal{G}$ to a 2-D grid space, which is a discrete euclidean space. For a given point $q = (x, y)$ in $\mathcal{G}$,the projected grid location point $p$ is,

$$p = (\lfloor x/g_x \rfloor, \lfloor y/g_y \rfloor) \tag{8}$$

where $g_x$ and $g_y$ is a user defined parameter to scale the value on x-coordinate and y-coordinate and control the granularity of the converted grid space. We can have a constant number of points in a grid space if it is transformed from a bounded euclidean space. We call each point in the grid space a cell, denoted by $p$. For two points $p_{g1}$ and $p_{g2}$ in the grid space, the distance $d(p_{g1}, p_{g2})$ is defined as,

$$d(p_{g1}, p_{g2}) = \lceil \sqrt{(x_{g1} - x_{g2})^2 + (y_{g1} - y_{g2})^2} \rceil \tag{9}$$

The user-feed *follow*ing relation is then defined by using the new distance function. A user's position will be considered stable if his location is not changed in the grid space. We may have location inaccuracy for each user by using the grid space. By choosing different granularity of the grid space, we can have different sensitivity to user movements and provide different level of spatial accuracy in the system.

The spatial inaccuracy for each user only involves false positive feeds as we use a ceiling function to calculate the transformed query range. Assuming a user $u_i$ subscribes to feeds within range $r$. For a given grid partition with parameters $g_x$ and $g_y$, the transformed query range will be $\frac{r}{g_{min}}$, where $g_{min} = \min\langle g_x, g_y \rangle$. We then transform $u_i$'s query range as $r_g = \lceil \frac{r}{g_{min}} \rceil$. The query range is extended from a circle centered at $u_i.pos$ with a radius $r$ to a $(r_g * g_x)$ by $(r_g * g_y)$ rectangle centered at $u_i.pos$. Since the original query area is completely covered in the transformed one, we can ensure all the feeds within a user's original query range will be included in the transformed query range in the grid space. We have a false positive ratio bounded by $G_{bound} = \frac{r_g^2 g_x g_y}{\pi r^2} - 1$, and we have

$$\frac{g_x g_y}{\pi g_{min}^2} \le G_{bound} \le \frac{(\frac{r}{g_{min}} + 1)^2 g_x g_y}{\pi g_{min}^2 r^2} \tag{10}$$

We can see that when the grid cells are not squares, the false positive area will be larger. By assuming the grid is a square, we have,

$$\frac{1}{\pi} \le G_{bound} \le (\frac{g}{r} + 1)^2 \tag{11}$$

where $g$ is the edge length of a square cell.

As mentioned above, we put the user queries into groups so that the queries in each group have identical query range, ranking function, and aggregate function. We have the following two straightforward but useful lemmas.

LEMMA 3.1. *Users with the same query range, aggregate and ranking function located at the same grid cell have the same following list.* □

LEMMA 3.2. *If two feeds are located at the same cell and there is a user who follows one of them, then the same user also follows the other one.* □

Maintaining every single user's position and following list may incur unnecessary duplicate information because of Lemma 3.1. We use a location-centric query frequency statistic $\mathcal{S}$ instead of $\mathcal{U}$ to reduce the input size. We define $\mathcal{S}$ as a map $\langle p, \Theta \rangle$,

$$\mathcal{S} = \begin{cases} \langle p, \sum_{u_i \in \mathcal{U} \& u_i.pos = p} u_i.\theta \rangle & , \exists u_i \in \mathcal{U} \& u_i.pos = p \\ \langle p, 0 \rangle & , \text{otherwise} \end{cases} \quad (12)$$

$\mathcal{S}$ may be a sparse map and we can omit the statistics of the cells without any user.

Similarly, we can also store the feeds using a location-based map $\mathcal{H}$, which can be defined as,

$$\mathcal{H} = \begin{cases} \langle p, \{f_j | f_j.pos = p\} \rangle & , \exists f_j \in \mathcal{F} f_j.pos = p \\ \langle p, 0 \rangle & , \text{otherwise} \end{cases} \quad (13)$$

Furthermore, a user-centric query plan $V(u_i)$ can be transformed into its location-centric form $V(p)$ based on Lemma 3.1. $V(p)$ will be generated by using $\mathcal{S}$ as the input and optimize the user queries at $p$ and indexed by the cell id $p$. The location-centric query plan $V(p)$ can be used by any user located at cell $p$. The number of query plans is independent on the number of users but rather dependent on the number of grid cells, which can be controlled by setting the parameters $g_x$ and $g_y$ as stated above. We use $\mathcal{QP}$ to denote all the location-centric query plans of an optimized plan.

For each materialized view $v_i$, we can maintain a list of grid cells whose query plans involve $v_i$ for a given $\mathcal{QP}$. We call such a list as the *service list* of $v_i$, denoted as $SV$:

$$v_i.SV = \{p | v_i \in \mathcal{QP}[p]\} \quad (14)$$

The service list can be easily calculated after generating the query plans. We use it to quantify the influence of each materialized view on the system performance.

In summary, a location-centric optimization plan $\mathbb{P}_{loc}$ is a 2-tuple $\langle \mathcal{V}, \mathcal{QP} \rangle$, where $\mathcal{V}$ is the set of materialized views and $\mathcal{QP}$ is the set of location-centric query plans for all the user queries.

## 3.3 Grid-Based Approach

By assuming using the grid space to represent the locations of users and feeds, we can generate the basic candidate views by grouping all the feeds by their located grid cells. These candidate views are referred to as native views, which are needed to store the events from the feeds and have to be maintained regardless of the user query plans being used. They guarantee users can go offline at anytime while having the guarantee of receiving their messages.

By using the native views, we can answer each user query by aggregating the views that are within the query range of the user. We call this approach a *Grid-based* approach and use it as the baseline solution. The query plans under the grid algorithm can be easily generated by using a reversed index from the id of each cell, i.e. $p$, in $\mathcal{S}$ to the set of views whose *Effective Range* covers $p$. Based on the user-feed *following* relationship (Section 2.1), we define the *Effective Region* $ER(v_i, r)$ as the set of user locations where a materialized view $v_i$ can be used to answer the users' queries whose query ranges are equal to $r$. The formal definition is as follows:

$$ER(v_i, r) = \{p | d(p, f_i.pos) < r, \forall f_i \in v_i.F\} \quad (15)$$

where $d(p, f_i.pos)$ is defined in Equation 9 and $v_i.F$ is the set of feeds contained in $v_i$.

**Algorithm 1:** Grid Algorithm

```
   Data: S⟨p, Θ⟩, H⟨ p,FeedSet ⟩
   Result: Query optimization plan P_loc(V, QP)
1  Initial HashMap V
2  Initial HashMap QP
3  foreach p ∈ H.keys do
4      v ← View(H[p])
5      v.φ, v.θ, v.pos ← 0, 0, p
6      foreach f ∈ H[p] do
7          v.φ ← v.φ + f.φ
8      V[v.id] ← v
9  foreach p ∈ S.keys do
10     foreach (v.id, v) ∈ V do
11         if p ∈ ER(v,r) then
12                 ▷ r is the selected user query range for the current user group
13             Update v.θ
14             Add p to v.SV
15             Add v to QP[p]
16 Return P(V, QP)
```

Algorithm 1 presents the details of this algorithm. We initialize the two hash maps, $V$ and $QP$, in lines 1 – 2, which are to store the materialized views and query plans on each grid cell. In lines 3 – 10, we create views for each distinctive $p$ in $\mathcal{H}$. For each cell id $p$, we calculate the update frequency of the generated view by summing up the update frequencies of all the feeds in this cell's view. The generated view is stored in the set of materialized view $\mathcal{V}$. In lines 11 - 17, after all the views are generated, the query plans are generated by assigning the native views to the grid cells that are within their effective range and contain users. We also compute the query frequency of the views and add the corresponding $p$s to the views' service lists. Finally, line 18 returns the plan.

**Adaptation to user movements.** Since we assume feeds do not move in this algorithm, we only need to update a grid's query plan when a user appears in a grid cell where there is no existing query plan. When a new user query arrives or a user moves to another cell, we can check the hash map $\mathcal{QP}$. The user's query will be answered by the stored plan at the new location if it exists, otherwise a new query plan will be generated and stored into $\mathcal{QP}$.

## 3.4 Composite-View Approach

In the *Grid-based* algorithm, we only generate a native view for each cell. However there are many potentially beneficial views that contain feeds from multiple grid cells. We refer to such a view as a *composite view*, denoted as $v_c$. The number of composite views is exponential to the number of feeds in the system. These views can potentially be shared by multiple users. In general, maintaining a

composite view $v_c$ may introduce an extra maintenance cost but may reduce the query evaluation cost for users located within $v_c$'s effective range, $v_c.ER$.

With the movements of users and the changes of the characteristics of event producers, the query frequency at each cell and the update frequency of each feed may change over time. Therefore, we need an algorithm that can be adaptive to such changes progressively. In this section, we present an iterative *Composite-view* algorithm which not only generates better optimization plans by considering composite views, but can also adapt to the system changes over time by re-applying the algorithm over the current plan with updated statistics.

**Estimating the characteristics of a composite view.** A composite view $v_c$ can be constructed by combining two materialized views $v_1$ and $v_2$ as follows: 1) the feeds involved in $v_c$ are $F_{v_c} \leftarrow F_{v_1} \cup F_{v_2}$; 2) the update frequency is the sum of the update frequencies of all the feeds in $F_{v_c}$; 3) the effective region is calculated as $v_c.ER \leftarrow v_1.ER \cap v_2.ER$; 4) the generated composite view $v_c$ can be used to answer a user query whose query plan uses both $v_1$ and $v_2$. We estimate the query frequency of $v_c$ by estimating $v_c.SV \leftarrow v_1.SV \cap v_2.SV$ with the assumption that all location-centric plans previously using both $v_1$ and $v_2$ will now use $v_c$. We can calculate the estimated query frequency of $v_c$ by summing up the query frequencies for all the users in $v_c.SV$, which is a upper bound of the query frequency of $v_c$. We denote the generation of $v_c$ using $v_1$ and $v_2$ using $v_c \leftarrow v_1 + v_2$.

To help deciding which view to materialize, we define the materialization benefit $B(v_j)$ for each candidate view. Given $v_j.SV$, the service list of a view $v_j$, we can calculate the evaluation cost of all the user queries using $v_j$,

$$MEV(v_j) = \sum_{p \in v_j.SV} EV(\mathcal{S}[p], V(p)_{new}) \tag{16}$$

On the other hand, we also estimate the evaluation cost of these queries in the case that we do not materialize $v_j$. We use $NEV(v_j)$ to denote this cost:

$$NEV(v_j) = \sum_{p \in v_j.SV} EV(\mathcal{S}[p], V(p)) \tag{17}$$

Therefore, the benefit of materializing $v_j$ can be calculated using the total cost reduction as,

$$B(v_j) = NEV(v_j) - (MEV(v_j) + M(v_j)) \tag{18}$$

Calculating the accurate benefit values requires the query plans of the relevant users, which are time consuming to generate. We simplify the process by estimating the benefit of $v_j$ as follows. For each grid cell $p$ within $v_j.SV$, we use $v_j$ to replace the views in the query plan of $p$ that are subsets of $v_j$. We denote the views in the query plan of $p$ that are subsets of $v_j$ as $R(v_j, p)$, i.e.

$$R(v_j, p) = \{v | v \in V(p) \& F_v \subset F_p\} \tag{19}$$

Then for each grid cell $p$ in $v_j.SV$, the potential reduction of the evaluation cost by using $v_j$, denoted by $EVR(p, v_j)$, is

$$EVR(p, v_j) = \begin{cases} \mathcal{S}[p] \cdot |R(v_j, p)|, & |R(v_j, p)| = |V(p)| \\ \mathcal{S}[p] \cdot (|R(v_j, p)| - 1), & 1 \leq |R(v_j, p)| \leq |V(p)| \\ 0, & |R(v_j, p)| \leq 1 \end{cases} \tag{20}$$

We have the estimated benefit $B'(v_j)$ defined as follows,

$$B'(v_j) = \sum_{p \in v_j.SV} EVR(p, v_j) \tag{21}$$

**Algorithm.** The details of the *Composite-view* algorithm are presented in Algorithm 2. For the ease of presentation, we define a function $Neighbor(v_j, r)$ as follows:

$$Neighbor(v_j, r) = \{v_i | ER(v_i, r) \cap ER(v_j, r) \neq \emptyset\} \tag{22}$$

**Algorithm 2:** Composite-view Algorithm

**Data:** $\mathcal{S}\langle p, \Theta \rangle$, query range $r$, initial plan $\mathbb{P}_{loc0}$
**Result:** Query optimization plan $\mathbb{P}_{loc}(\mathcal{V}, \mathcal{QP})$

1 $MV \leftarrow \mathbb{P}_{loc0}.\mathcal{V}$; ▷ storing views considered to be further combined to composite views
2 $\mathcal{V} \leftarrow \mathbb{P}_{loc0}.\mathcal{V}$; ▷ views selected to be materialized
3 $\mathcal{CV} \leftarrow \mathcal{V}-$ native views; ▷ storing the views to be considered to be materialized or un-materialized
4 $\mathcal{QP} \leftarrow \mathbb{P}_{loc0}.\mathcal{QP}$;
5 **foreach** $v_i \in MV$ **do**
6   Mark $v_i$ as visited;
7   $B_{max} \leftarrow B'(v_i)$;
8   $v_c, v_s \leftarrow null, null$;
9   **foreach** $v_j \in Neighbor(v_i, r)$ **do**
10     **if** $v_j$ *is not yet visited* **then**
11       $v_t \leftarrow v_i + v_j$;
12       **if** $B(v_t) \geq B_{max}$ **then**
13         $v_c, v_s \leftarrow v_t, v_j$;
14         $B_{max} \leftarrow B'(v_t)$;
15   **if** $v_c \neq null$ **then**
16     Insert $v_c$ into $MV$;
17     **if** $B(v_c) > \delta$ **then** insert $v_c$ to $CV$;
18     Remove $v_s$ from $MV$;
19 Sort $\mathcal{CV}$ by their benefits in descending order;
20 **foreach** $v_i \in \mathcal{CV}$ *in sorted order* ▷ *Select views to materialize or un-materialized from the candidate views* **do**
21   **if** $B(v_i) \leq \delta$ ▷ *if the benefit is less than a threshold, then un-materialize it* **then**
22     Remove $v_i$ from $\mathcal{V}$;
23     **foreach** $p \in ER(v_i, r)$ **do**
24       **if** $\mathcal{S}[p] \geq 0$ *and* $v_i \in \mathcal{QP}[p]$ **then**
25         $\mathcal{QP}[p] \leftarrow GreedySetCover(F_p, \mathcal{V})$;
26   **else**
27     **if** $v_i \in \mathcal{V}$ **then** Continue ;
28     Add $v_i$ to $\mathcal{V}$;
29     **foreach** $p \in ER(v_i, r)$ **do**
30       **if** $\mathcal{S}[p] \geq 0$ **then**
31         $\mathcal{QP}[p] \leftarrow GreedySetCover(F_p, \mathcal{V})$;
32 return $\mathbb{P}_{loc}(\mathcal{V}, \mathcal{QP})$;

The *Composite-view* algorithm takes the same input as the *Grid-based* algorithm plus an initial optimized plan. The initial plan can be the plan generated by the *Grid-based* algorithm or an existing plan that needs to be re-optimized. The algorithm starts by initializing the list of materialized views $MV$ to be used to generate composite views, the resulting materialized view set $\mathcal{V}$, and the user query plan $\mathcal{QP}$ using the initial plan. We also initialize $\mathcal{CV}$ with the existing materialized composite views whose materialization benefit should be re-examined. In lines 5–18, we generate a composite view that can achieve the highest benefit by combining a materialized view with other materialized views. In each loop, the size of $MV$ is decreased by 1 and a new composite view will be added to $\mathcal{CV}$ if it has a greater materialization benefit than that of $v_i$. We can expect the loop to end after $|MV|$ iterations and generate at most $|MV| - 1$ composite views.

After we have generated the composite views, we sort the candidate views in $\mathcal{CV}$ by their benefit values in descending order (line 19) and then decide whether we should materialize or un-materialize them. In lines 22–25, if a candidate view's benefit is less than a preset threshold $\delta$, we un-materialize it and remove it from the current query plans. In other words, all the query plans that use this view need to be re-calculated. Each query planning needs to solve a minimum set cover problem using all the materialized views to cover $F_p$. We generate query plans using a greedy minimum set cover algorithm, which has the best possible approximate ratio with a polynomial complexity [8], and denote the query planning function as $GreedySetCover(F_p, \mathcal{V})$. In lines 27–31, if a view's benefit is greater than the threshold, we materialize it if it is not yet in $\mathcal{V}$. In a similar way, we update the query plans within the effective region of the newly materialized view. The new plan is returned in line 32 after finishing the process of all the candidate views.

**Complexity analysis.** For the generation of composite views in the *Composite-view* algorithm (lines 5–18) , the worst-case complexity is $O(|\mathcal{V}|^2)$, where $|\mathcal{V}|$ is the number of views in $\mathcal{V}$. For lines 20–31, the worst case will be all the composite views in the initial optimization plan need to be removed and all the new composite views need to be materialized. The complexity of running the greedy minimum set cover algorithm in line 25 or 31 is $O(n \cdot |\mathcal{V}|)$ where $n$ is the number of native views covered by $F_p$ and $|\mathcal{V}|$ is the number of views in $\mathcal{V}$. Since $\mathcal{V}$ contains all the native views, there should always exist a cover for any $F_p$. The worst case is that we have to update the query plan of every cell. For updating the query plan for $p$, we need to execute *GreedySetCover* at most $n$ times. Therefore, the worst-case complexity of *Composite-view* algorithm is $O(m \cdot n_{max}^2 \cdot |\mathcal{V}| + |\mathcal{V}|^2)$, where $m$ is the number of grid cells and $n_{max}$ is the maximum number of native views covered by $F_p$ for any cell $p$.

**Adaptation to user movements.** Similar to the *Grid-based* approach, if a user appears in a cell that has no query plan yet, we should generate a new query plan. We also need to change the materialized views if their benefits are too low. However we should not update all the query plans if the statistics are just changed slightly. Instead, we collect the total changed distance of all the users as the metric to measure how good the current query optimization plan might be and to decide whether we should try to search for a new plan. We define the total changed distance $\Delta(\mathcal{U}, \mathcal{U}_{plan})$ as follows,

$$\Delta(\mathcal{U}, \mathcal{U}_{plan}) = \Sigma_{u \in \mathcal{U}} d(u.pos, \mathcal{U}_{plan}[u.uid].pos) \qquad (23)$$

We check the total changed distance periodically and decide whether a new optimization plan is needed by comparing it to a threshold. As Algorithm 2 is designed as an iterative process, the new plan generation can be done simply by running the algorithm with the current plan as its input plan.

## 3.5 Mobile Feeds

As mentioned in the beginning of Section 3.2, the above algorithms assume feeds are static. In some feed-following applications, such as augmented reality social network games like Ingress [18] and Pokemon Go [19], users may move around over time and requesting information from all nearby users continuously. Therefore, the feeds are also moving. Here, we extend our algorithms for scenarios with moving feeds. Recall that we assume a user only subscribes to feeds which are currently located within his query range. Therefore, we can model feeds that generate events at different locations as multiple "virtual feeds". More specifically, we can group events generated by all the feeds based on location. Using our grid space, events can be assigned to a finite number of grid cell and those allocated at the same cell can be aggregated as a new virtual feed. By creating such virtual non-mobile feeds, we can reuse our problem formulation and algorithms for static feeds. This again shows the benefit of our location-centric approach.

## 3.6 Grid Granularity

In this subsection, we analyze the implication of the granularity of the grid space. Suppose the query range of user $u_i$ is $r_i$ in the original space, then the equivalent query range in the grid space with parameter $g_x$ and $g_y$ is a rectangle containing $\lceil r_i/g_x \rceil \cdot \lceil r_i/g_y \rceil$ cells.

According to Equation 5 and 6, the total maintenance cost of all the native views is independent on the grid granularity. The evaluation cost of a user query, on the other hand, depends on the number of views used in evaluating the query, which is equivalent to the number of cells overlapping with the query range. So according to Equation 4, the query evaluation cost of user $u_i$ is equal to $u_i.\theta \cdot \lceil r_i/g_x \rceil \cdot \lceil r_i/g_y \rceil \cdot L$. We can see that the evaluation cost decreases with larger $g_x$ and $g_y$. However, the larger $g_x$ and $g_y$ values introduce a greater spatial inaccuracy such that a user $u_i$ can receive messages from feeds that are $\sqrt{(\lceil r_i/g_x \rceil \cdot g_x)^2 + (\lceil r_i/g_y \rceil \cdot g_y)^2}$ away. In other words, the feeds located $\sqrt{(\lceil r_i/g_x \rceil \cdot g_x)^2 + (\lceil r_i/g_y \rceil \cdot g_y)^2} - r_i$ away from $u_i$ are incorrectly followed by $u_i$. This trade-off between spatial accuracy and system workload should be determined according to the requirements of the application.

# 4 EVALUATION

**Datasets.** We conduct our experiments using two real datasets. The first one is GeoText[6], which is a Twitter dataset containing 377,616 messages over one week in March 2010 from 9,475 nodes approximately within the United States. The second one is a BrightKite dataset from SNAP[13]. Brightkite was once a location-based social-network service provider where users shared their locations by checking-in. The SNAP dataset consists of 58,228 nodes and 214,078 edges with a total of 4,491,143 check-ins of these users over the period of Apr. 2008 - Oct. 2010. We use the GeoText dataset to show the performance under a light workload situation and the SNAP dataset for heavy workload in the following experiments. We convert the location to ECEF (Earth-Centered, Earth-Fixed), a Cartesian coordinate system, and then apply Equation 8 to produce the grid cells.

To simulate different number of users and feeds, we sample some nodes in the datasets as users and the others as feeds. As the datasets do not contain feeds, we sample some nodes' initial locations and use them as the feeds' locations. For those nodes sampled as users, the locations extracted from the check-in records of the corresponding data node will be used. We calculate the average time period between each node's check-ins and use it as the update frequency if it is sampled as a feed or the query frequency if it is a user.

**Algorithms.**

(1) *GeoFeed.* GeoFeed [1] algorithm adapted to our system model. User-centric query plan is used.
(2) *GridView.* Materialize views for all the feeds within each cell. A user's query will be answered by push strategy if his query range covers only a single grid cell. For users following more feeds, pull strategy is applied to answer his query using the grid-based views. We use location-centric query plans to make fair comparison to the *CompositeView* algorithm.
(3) *CompositeView.* Our *CompositeView* algorithm presented in 2. The push and pull cost ratio $H/L$ is 2.83 based on the CPU usages of a push-only and a pull-only scheme.

To obtain stabilized results, each algorithm under each parameter setting is run for 20 minutes.

**Implementation and Cluster Hardware.** We implement our system prototype and optimizer using Python 3.2 and Redis 3.0.1 [20], an in-memory key-value store system, as the back-end storage system. The architecture of the prototype system is depicted in Figure 2. All the data in the materialized views is partitioned and stored at the Redis nodes using hash partitioning. The *optimizer* monitors the system log and calculate the optimization plan based on the statistics of the users and the feeds extracted from the log. The optimized query plans is maintained at the query router. We also use Java 1.7 to implement an executor to simulate the user queries and feed update operations.

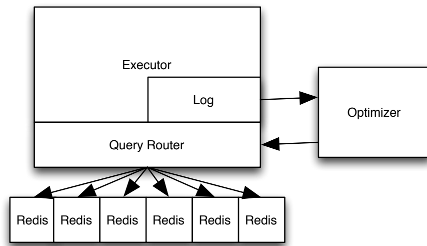



**Figure 2: System architecture**

The experiments are conducted on a cluster of 7 IBM iDataplex dx360 servers with 2 2.66Ghz Intel Nehalem-EP CPUs (X5550) and 48 GB Ram. The cluster nodes are connected using 40 GBps QDR Infiniband interconnect and an oversubscription ratio as 2:1 is used. 6 data nodes running Redis are used as storage servers that store the materialized views. We also have 1 separate node to maintain the materialized views and to process user queries , which we call it processor node. The query router modules is also running on this node.

**Metric.** According to our experiments, as long as the workload is under the system's maximum throughput, the query latency is insenstive to the view materilization plans and remains the same with different view selection algorithms. As we use an in-memory database with sufficient RAM and a high speed network, CPU is the major bottleneck and hence a lower CPU usage indicates a higher system throughput and higher capability of maintaining low query latency. Therefore, we use the total CPU usage of all the Redis server processes on each node in the cluster as the metric, which is collected by using the `pidstat` command in Linux. We use the sum of the average CPU usage (in the unit of percentage of the CPU's capacity) on each data node as the performance metric. Furthermore, CPU usage is a good indicator of the energy consumption of a cluster and CPU is the dominant energy consumer in Google servers [2]. Therefore, reducing the CPU usage even by a few percentage points could significantly cut down the operational cost of the service provider.

| | GeoText | SNAP |
|---|---|---|
| User Percentage | 0.9 | 0.9 |
| Load Level | 400 | 667 |
| Granularity | 250 | 250 |
| Query Range (cells) | 5 | 5 |

**Table 2: Default Parameters**

## 4.1 Experiment Setup

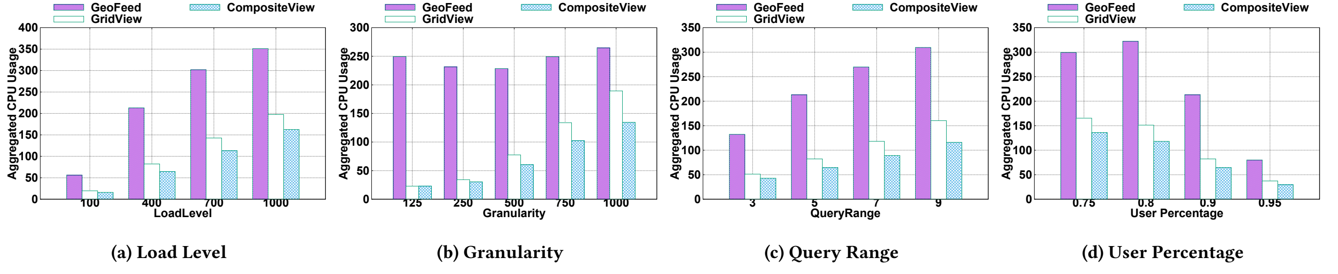
## 4.2 Static Scenario

We first present the algorithm's performance with a static scenario, where users and feeds and are fixed at their initial locations in the datasets. The basic parameters are set as stated in Table 2. We only vary one parameter in each of the following experiments. The meaning of the parameters are explained in the subsections where we vary them.

*4.2.1 Impact of Workload.* We simulate different amount of workloads to verify our algorithm's scalability. We use the load level to change the overall update/query frequency of users and feeds. The load level is used to control the overall update and query frequencies by multiplying them with the load level values. It will affect the query frequency of users and update frequency of feeds that we use to generate the optimization plan. A higher load level will result in a heavier workload for both users and feeds. We use this parameter to control the input user set and feed set's query or update speed and evaluate the algorithm's capability to process feed-following workload under different intensity.

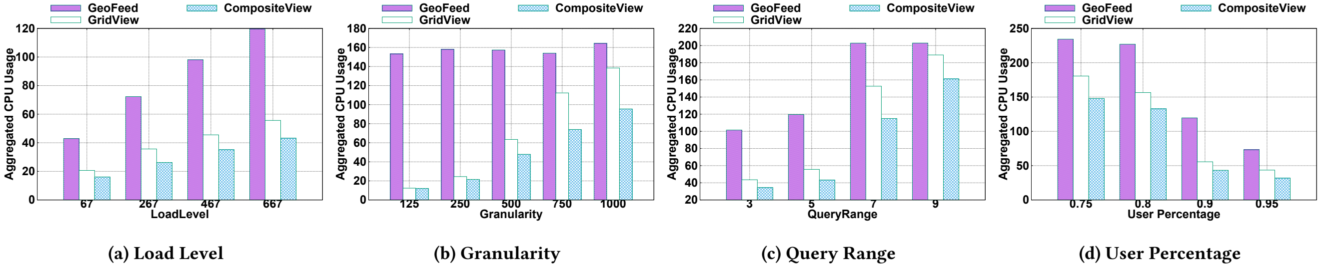
For the GeoText dataset, we can see from Figure 3a that, when the load level increases, all the three algorithms' CPU usages increase proportionally. *GeoFeed* creates too many views simply because it only considers views containing a single feed followed by a user. *GridView* creates location-centric views in the granularity of grid cells and achieves an average CPU usage 56% less than that of *GeoFeed* under all load levels. *CompositeView* achieves a further 20% improvement in CPU usage by using composite views.

In Figure 4a, we can find similar results with the SNAP dataset. There are more users and feeds in the SNAP dataset. This makes user-centric *GeoFeed* algorithm perform worse to process feed-following queries. We cannot handle higher workload as the processor node is overwhelmed under *GeoFeed*'s plan.We can find *CompositeView* achieves a greater improvement over *GridView* in comparing to using the GeoText dataset. This is because the larger number of users and feeds provides greater opportunities to generate composite view with more feeds and to share views among more user queries. However, when the load level increases, the improvement drops slightly. This is because the increase of the load level will simultaneously increase the feed update frequency and query evaluation frequency by the same scale, and the ratio between the




(a) Load Level (b) Granularity (c) Query Range (d) User Percentage

**Figure 3: Results of the Static Experiments, Geotext Dataset**



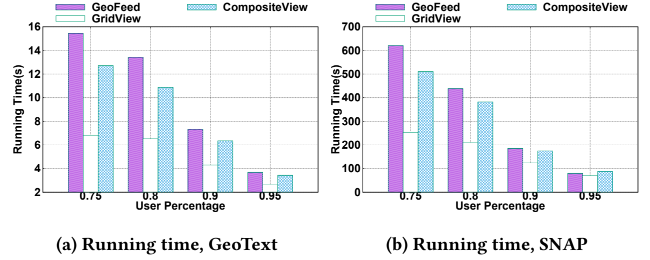

(a) Load Level (b) Granularity (c) Query Range (d) User Percentage

**Figure 4: Results of the Static Experiments, SNAP Dataset**

push and the pull cost is 2.83, which is greater than one. Hence, with the increase of the load level, the total view maintenance cost (update frequency * push cost) increases faster than the query evaluation cost (query frequency * pull cost). This makes the optimizer to choose to materialize less views with a higher load level. We can conclude that our location-centric algorithms can achieve better performance in comparing to the user-centric *GeoFeed* algorithm under different load level, and our *CompositeView* algorithm can further improve the performance under all the load levels for both datasets.

*4.2.2 Impact of Granularity.* We also simulate our algorithm under different granularity of the grid space, which affects the average size of each grid-based view. We collect all the records from the original check-ins. Then we calculate the maximum distance on each dimension of the transformed ECEF coordinates notated as $(d^x_{max}, d^y_{max})$ and select different granularity parameter $Gran$ and set $g_x = \frac{d^x_{max}}{Gran}$ and $g_y = \frac{d^y_{max}}{Gran}$. Here we set the query range with granularity 125 as one cell and scale its value proportionally with the increase of the granularity so that the actual spatial query range remains the same across different granularities.

For GeoText, we can see from Figure 3b that with a finer granularity, *CompositeView* has greater enhancements with regard to *GridView* simply because there are more opportunities to combine grid views into composite views to reduce the query evaluation cost. *GeoFeed* does not use the grid space for query optimization so its performance is insensitive to the granularity. With the granularity 125i, *CompositeView* has the same performance as *GridView*, besause the query range is one grid cell and there is no beneficial composite views. In Figure 4b, we can find similar results for SNAP. Based on the two experiment results, we can see that a coarser granularity can significantly improve the system performance by sacrificing the location accuracy and *CompositeView* can achieve the best performance under all the granularity. An interesting result




(a) Running time, GeoText (b) Running time, SNAP

**Figure 5: Running time under different user/feed ratio**

is that CompositeView with granularity 1000 has a CPU usage that is even lower than (or similar to) GridView with granularity 750. This further validates the effectiveness of CompositeView in finding beneficial views without over-compromising spatial accuracy.

*4.2.3 Impact of query range.* We also simulate our algorithm under different query ranges to examine the sensitivity to the number of feeds followed by a user. From Figure 3c, we can see all the algorithms' CPU usages are increased with a greater query range (and hence a greater number of feeds followed by each user). *CompositeView* achieves a higher performance enhancement with a greater query range due to the fact that larger composite views can be used to to reduce the query evaluation cost. Comparing with *GridView*, it achieves 17% and 28% less CPU usage with a query range as 3 cells and 9 cells respectively. The CPU usage of *GeoFeed* is more than 2 times of that of *GridView*. Similar results can be found in Figure 4c. Since we have heavier workload with the SNAP dataset and there are more feeds followed by each user, there is less performance improvement that can be achieved by *GridView* and *CompositeView* with larger query ranges. Based on the results in this and the previous subsection, with a larger query range, one should use a coarser granularity to achieve a better performance under the assumption that a larger query range can tolerate a higher spatial inaccuracy.

*4.2.4 Impact of user/feed ratio.* In this subsection, we examine the algorithms' sensitivity to different user/feed percentage. From each data set, we keep a certain percent of users and make the rest as feeds. We vary the user percentage to simulate the situations with different user/feed ratios.

The results of GeoText are presented in Figure 3d. In general, the CPU usages of all the algorithms become lower with a higher user percentage (and hence fewer feeds). This is because fewer feeds would result in fewer views to maintain and fewer updates to be processed. *GridView*'s CPU usage is half of *GeoFeed*'s under all cases and *CompositeView* further improves it by 20%. This verifies *CompositeView* can achieve similar performance when the user percentage is changed. Similar results can be found for the SNAP dataset in Figure 4d.

*4.2.5 Running time.* We also collect the algorithms' running time. Here we only present the results with various user percentages. The other results show similar trends. By using the location-centric query planning strategy, both *GridView* and *CompositeView* reduce the number of feeds and query plans in a feed-following system. The running time of *GridView* is less than half of that of the user-centric algorithm *GeoFeed*, as shown in both Figure 5a and Figure 5b. The running time of *CompositeView* is much higher than *GridView* because of a much larger search space. However the running time of *CompositeView* is comparable to that of *GeoFeed*.

## 4.3 Dynamic Scenario

Here, we present the experiments within a dynamic environment, where users move over time according to the location information stored in the datasets. The various parameters are set as in Table 2. When a user moves to a new location, *GridView* simply replaces his query plan with the one at his new cell and *GeoFeed* would re-generate his query plan. In addition, *CompositeView* can iteratively update the optimized plan over time by using the new statistics.

Besides the cost of query evaluation and view maintenance, the cost of updating the query plans is also counted in the total CPU usage. To compare the algorithms' performance under the dynamic and the static scenarios, we also run the same experiments under a static scenario by simply fixing the users to their initial locations.

*4.3.1 Dynamicity of the Datasets.* We collect the percentage of user movements in the two real datasets and present them in Figure 6. The check-in records within each dataset is loaded by using the grid granularity as 250. If a user's check-in cell is different from its last one, it is counted as one user movement. We collect the percentage of user check-ins which result in a user movements per time unit. We use box plot to report the median, first quantile, third quantile, maximum and minimum of the percentages. Recall that, in *CompositeView*, we use a threshold on the total moved distances of users to trigger the re-optimization. The threshold should be chosen to limit the cost of re-optimization, so in general, we would select a higher threshold for a more dynamic dataset. After some trial-and-error tuning, we set the threshold as 25,000 for GeoText and 250,000 for SNAP.

*4.3.2 Impact of Workload.* We first test the performance under different workloads by using different load levels on each dataset, similar to what we do for the static experiments.

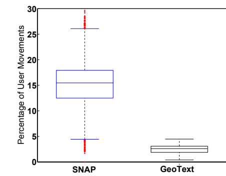



**Figure 6: Percentage of User Movements of Each Dataset**

We examine both the low (100) and high load level (1000) for both datasets and present the results in Figure 7. We use box plot to report the median, first quantile, third quantile, maximum and minimum CPU usage over the whole system running time. Box "D" and "S" represent the Dynamic and Static cases respectively.

We can see that, in comparing to the static case, *GeoFeed* has a much higher CPU usage in a dynamic case with moving users. This is mainly due to the extra overhead of frequent updates of user plans, which is the drawback of user-centric plans. On the other hand, by adopting location-centric query plans, *GridView* can achieve a CPU usage comparable to the static results. Furthermore, thanks to the use of composite views and the iterative optimizer, *CompositeView* further decreases the system workload, which is more significant with a higher load level (Figure 7b).

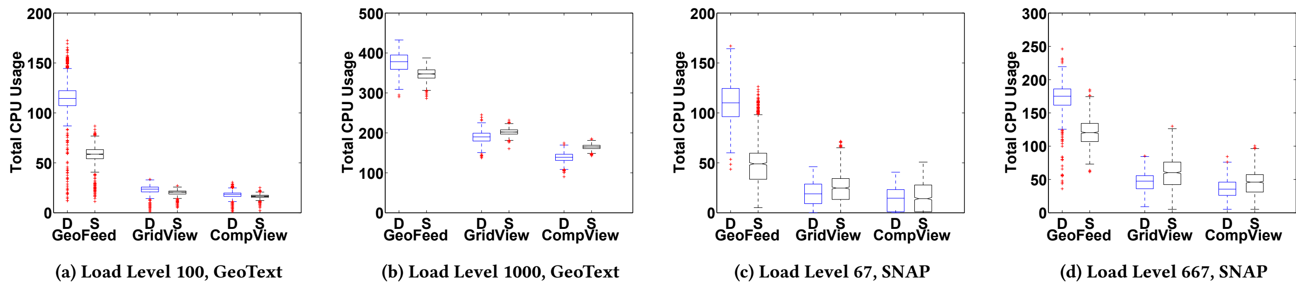
For the SNAP dataset, similar results can be found in Figure 7c and Figure 7d. We can conclude that location-centric plans significantly outperform user-centric plans when users are continuously moving as the two real datasets.

*4.3.3 Impact of Grid Granularity.* We also conduct experiments with different gird granularity using the same parameters as in the static experiments on granularities. The results are presented in Figure 8.

From Figure 8a and Figure 8b, we can find the CPU usages of both location-based algorithms are much lower than the user-centric *GeoFeed*, with *CompositeView* being the overall best performing algorithm. We can also see that the performance gaps among the algorithms are larger with a relatively coarse-grained grid (granularity 125) in comparing to the finer-grained grid (granularity 750). This is because, with a coarse-grained grid and location-centric query plans, users can share the same query plan within a greater range and the corresponding materialized views would have greater benefit.

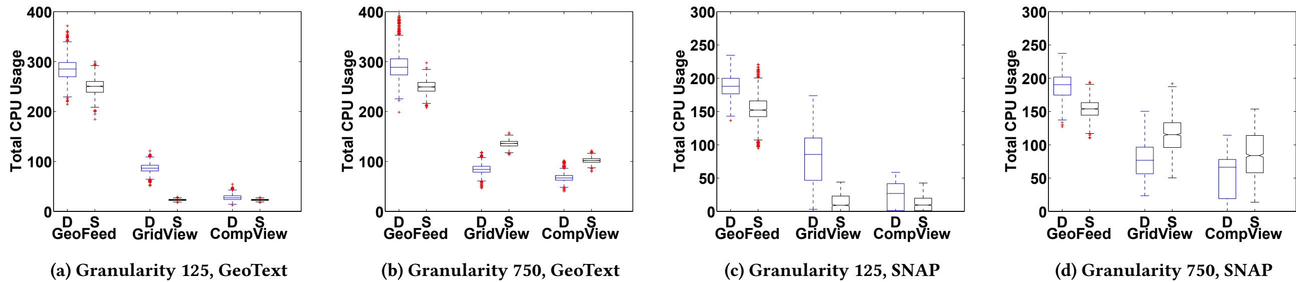
When comparing to the static results, *GeoFeed* has a continuous and consistent overhead of deploying new query plans for the moving users with different granularities. It is interesting to see that, with a fine-grained granularity (Figure 8b), both *GridView* and *CompositeView* can achieve lower CPU usages in the dynamic case than that in the static case. This is due to the fact that both algorithms can effectively capture the movements of users during runtime, and hence achieve a lower total cost when users move closer to each other and there are opportunities to share more views among the users.

The results of the SNAP dataset are shown in Figure 8c and Figure 8d. While we can draw similar conclusions as in the GeoText dataset, we can also observe that all the algorithms have higher



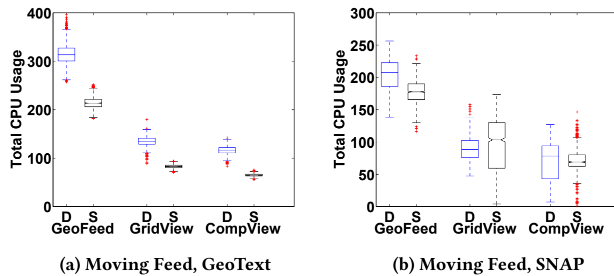

(a) Load Level 100, GeoText (b) Load Level 1000, GeoText (c) Load Level 67, SNAP (d) Load Level 667, SNAP

**Figure 7: Impact of Load Level, Dynamic Scenario**




(a) Granularity 125, GeoText (b) Granularity 750, GeoText (c) Granularity 125, SNAP (d) Granularity 750, SNAP

**Figure 8: Impact of Granularity, Dynamic Scenario**




(a) Moving Feed, GeoText (b) Moving Feed, SNAP

**Figure 9: Results of Moving Feed Experiments**

fluctuations in CPU usages due to the larger number of users and feeds in the SNAP dataset.

*4.3.4 Impact of Moving Feeds.* Finally, we conduct experiments to test the performance under the scenario of moving feeds. We use standard parameters presented in Table 2 and use the virtual feed abstraction for all the algorithms to make a fair comparison. The locations of users and feeds are dynamically changing overtime according to the two real datasets. Again we run the same experiment by fixing the feeds and users to their initial locations.

The results of the experiments are presented in Figure 9a and Figure 9b. Again *CompositeView* is the overall winner. Moreover, we observe that almost all the algorithms have higher CPU usages in the dynamic case than in the static case. This is because, by using the virtual feed abstraction, we introduce additional feeds into the system, because a virtual feed has to be created for each cell even there is currently no real feed in that cell. The larger number of virtual feeds introduces higher overhead to query planning, which also contributes to the CPU usages. Note that even though *CompositeView* and *GridView* use location-centric plans, we use a lazy query plan generation method, where a query plan is generated for a cell when the first user enters the cell. Hence there is still a run-time overhead of query planning for our algorithms even though it is much lower than that of location-centric plans.

## 5 RELATED WORK

An essential problem of query optimizations in feed-following systems is to select the materialized views. **View selection** is one of the most challenging problems in database systems and is known to be NP-complete [12]. It has also been proved that view selection is inapproximable for general partial orders. This area has been extensively surveyed in [5, 11, 15, 21].

View selection in **feed-following systems** without location information has been studied in Feeding-Frenzy [22], which provides a view selection solution creating views for each user-feed pair. They introduce a cost model to make query plan decisions on each user-feed pair without considering the location information. They consider a candidate view for each user-feed pair and each selected view only contains results from one feed to limit the number of candidate views. Moreover, user-centric query plans are used and hence it has the aforementioned drawbacks if it is applied to applications with moving users and feeds. In [3], the authors propose a view selection algorithm for feed-following systems, which considers sharing materialized views among different users. However, this approach does not support location-based feed-following services.

GeoFeed [1] studies the view selections in a location-based feed-following system. However, GeoFeed's feed following model is very different from ours. GeoFeed considers feed followings in a social network with a static following relation graph and moving users. Each user has a number of location-based queries, one for each friend of the user, which retrieves a friend's $k$ most recent messages whose locations are within a specified range from the userâĂŹs location. The optimizer is very similar to that of Feeding-Frenzy, which considers views containing a single feed and optimizes a

query plan for each user queries. As analyzed earlier, such user-centric query plans have limited adaptivity to the movement of users. Furthermore, in terms of optimization, the Grid-based algorithm is essentially equivalent to GeoFeed's optimizer, but generates location-centric plans instead of user-centric ones.

MobiFeed [25] extends GeoFeed's solution by using user location prediction to schedule the aggregated news periodically to mobile users. They do not consider the problem of user query optimization and assume aggregated news is available at all the predicted positions of a user.

Query optimization in feed-following systems is also related to **partial indexing** and **partial materialized views**. Luo [14] proposes a partial materialization method to only maintain frequently accessed results to minimize the response time of popular queries. Wu et al [24] present a partial indexing technique to support efficient content search in structured peer-to-peer networks. They only made indices for frequently accessed tuples while keeping others to be pulled from the sources. Aristides et al [9] present an approach where users pull social contents from some chosen users, acting as *hub* nodes, to reduce the maintenance cost and to improve event dissemination efficiencies in social networks. Our composite views can also be seen as *virtual hub* nodes which can reduce multiple users' query evaluation cost.

Another related research area is **multi-query optimizations**. Mistry et al [16] attempt to make use of multi-query optimization techniques in view selections. They find that common subexpressions among multiple views could be shared by multiple queries to significantly reduce 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 relationship among multiple users in our cost model to examine the sharing of common subexpressions and address the challenges specific to feed-following systems.

Feed-following can be considered as a special type of **pub/sub** service [26–29]. Handling location-based queries and moving subscriber are also important challenges in this area [4, 10, 17]. While these efforts consider the sharing of processing and communication among different pub/sub queries, we mostly focus on the sharing of the maintenance cost of materialized views. Our *Composite-view* algorithm can also be applied in a location-based pub/sub context by adopting a new cost model and using our *GreedySetCover* query planner.

## 6 CONCLUSION

This paper formulates the query optimization problem in a location-based feed-following system. We observe that in a highly dynamic system, where a substantial portion of users are moving, using location-centric query plans is more efficient than using user-centric ones. We propose a grid space model and transform the feed-following model into the grid space. Besides the native views at each grid cell, we also propose the concept of composite views that may contain feeds located at multiple cells. Such composite views can be potentially used by multiple users to further enhance the system performance. We then develop an iterative algorithm, which can re-optimize query plans progressively to handle user movements and keep up the high system performance over time. We conduct a comprehensive experimental evaluation on our algorithms by comparing them to the state-of-the-art solution. The results show that our *Composite-view* algorithm can achieve the best performance under most tested cases.

## REFERENCES

[1] Jie Bao, Mohamed F. Mokbel, and Chi-Yin Chow. 2012. GeoFeed: A Location Aware News Feed System. In *ICDE 2012*. 54–65.
[2] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. 2013. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. Morgan & Claypool Publishers.
[3] Kaiji Chen and Yongluan Zhou. 2016. Materialized View Selection in Feed Following Systems. In *BigData 2016*. 442–451.
[4] Lisi Chen, Gao Cong, Xin Cao, and Kian-Lee Tan. 2015. Temporal Spatial-Keyword Top-k Publish/Subscribe. In *ICDE 2015*. 255–266.
[5] Chandrashekhar A. Dhote and M. S. Ali. 2007. Materialized View Selection in Data Warehousing. In *ITNG 2007*. 843–847.
[6] Jacob Eisenstein, Brendan O'Connor, Noah A Smith, and Eric P Xing. 2010. A Latent Variable Model for Geographic Lexical Variation. In *EMNLP 2010*. 1277–1287.
[7] ETSI. 2019. TC Intelligent Transport Systems. http://portal.etsi.org/.
[8] Uriel Feige. 1998. A Threshold of ln *n* for Approximating Set Cover. *Journal of ACM* 45, 4 (1998), 634–652.
[9] Aristides Gionis, Flavio Junqueira, Vincent Leroy, Marco Serafini, and Ingmar Weber. 2013. Piggybacking on Social Networks. *PVLDB* 6, 6 (2013), 409–420.
[10] Long Guo, Dongxiang Zhang, Guoliang Li, Kian-Lee Tan, and Zhifeng Bao. 2015. Location-Aware Pub/Sub System: When Continuous Moving Queries Meet Dynamic Event Streams. In *SIGMOD 2015*. 843–857.
[11] Alon Y. Halevy. 2001. Answering Queries Using Views: A Survey. *The VLDB Journal* 10, 4 (2001), 270–294.
[12] Howard J. Karloff and Milena Mihail. 1999. On the Complexity of the View-Selection Problem. In *PODS 1999*. 167–173.
[13] Jure Leskovec and Andrej Krevl. 2017. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.
[14] Gang Luo. 2007. Partial Materialized Views. In *ICDE 2007*. 756–765.
[15] Imene Mami and Zohra Bellahsene. 2012. A Survey of View Selection Methods. *SIGMOD Record* 41, 1 (2012), 20–29.
[16] Hoshi Mistry, Prasan Roy, S. Sudarshan, and Krithi Ramamritham. 2001. Materialized View Selection and Maintenance Using Multi-Query Optimization. In *SIGMOD 2001*. 307–318.
[17] Jayanta Mondal and Amol Deshpande. 2014. EAGr: Supporting Continuous Ego-centric Aggregate Queries Over Large Dynamic Graphs. In *SIGMOD 2014*. 1335–1346.
[18] Inc. Niantic. 2019. Ingress,. http://www.ingress.com/.
[19] Nintendo. 2019. Pokemon Go,. http:/www.pokemon.com/.
[20] Salvatore Sanfilippo and Pieter Noordhuis. 2010. Redis. http://redis.io.
[21] Adam Silberstein, Ashwin Machanavajjhala, and Raghu Ramakrishnan. 2011. Feed Following: The Big Data Challenge in Social Applications. In *DBSocial 2011*. 1–6.
[22] Adam Silberstein, Jeff Terrace, Brian F. Cooper, and Raghu Ramakrishnan. 2010. Feeding Frenzy: Selectively Materializing Users' Event Feeds. In *SIGMOD 2010*. 831–842.
[23] 3GPP TR 36.885 V14.0.0. 2016. Study on LTE-based V2X Services.
[24] Sai Wu, Jianzhong Li, Beng Chin Ooi, and Kian-Lee Tan. 2008. Just-In-Time Query Retrieval Over Partially Indexed Data on Structured P2P Overlays. In *SIGMOD 2008*. 279–290.
[25] Wenjian Xu, Chi-Yin Chow, Man Lung Yiu, Qing Li, and Chung Keung Poon. 2015. MobiFeed: A location-aware news feed framework for moving users. *GeoInformatica* 19, 3 (2015), 633–669.
[26] Yongluan Zhou, Beng Chin Ooi, and Kian-Lee Tan. 2008. Disseminating Streaming Data in a Dynamic Environment: an Adaptive and Cost-Based Approach. *The VLDB Journal* 17, 6 (2008), 1465–1483.
[27] Yongluan Zhou, Beng Chin Ooi, Kian-Lee Tan, and Feng Yu. 2006. Adaptive Reorganization of Coherency-Preserving Dissemination Tree for Streaming Data. In *ICDE 2006*. 55:1 – 55:12.
[28] Yongluan Zhou, Ali Salehi, and Karl Aberer. 2009. Scalable Delivery of Stream Query Result. *PVLDB* 2, 1 (2009), 49–60.
[29] Yongluan Zhou, Zografoula Vagena, and Jonas Haustad. 2011. Dissemination of models over time-varying data. *PVLDB* 4, 11 (2011), 864–875.