LARS: A Location-Aware Recommender System

Justin J. Levandoski [1§], Mohamed Sarwat [2], Ahmed Eldawy [3], Mohamed F. Mokbel [4]

[1]*Microsoft Research, Redmond, WA, USA*

[2−4]*Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN, USA*

[1]justin.levandoski@microsoft.com, [2]sarwat@cs.umn.edu, [3]eldawy@cs.umn.edu, [4]mokbel@cs.umn.edu

spatial property

위치기반 정보

***Abstract*—This paper proposes LARS, a location-aware recommender system that uses location-based ratings to produce recommendations. Traditional recommender systems do not consider spatial properties of users nor items; LARS, on the other hand, supports a taxonomy of three novel classes of location-based ratings, namely, *spatial ratings for non-spatial items*, *non-spatial ratings for spatial items*, and *spatial ratings for spatial items*. LARS exploits user rating locations through *user partitioning*, a technique that influences recommendations with ratings spatially close to querying users in a manner that maximizes system scalability while not sacrificing recommendation quality. LARS exploits item locations using *travel penalty*, a technique that favors recommendation candidates closer in travel distance to querying users in a way that avoids exhaustive access to all spatial items. LARS can apply these techniques separately, or together, depending on the type of location-based rating available. Experimental evidence using large-scale real-world data from both the Foursquare location-based social network and the MovieLens movie recommendation system reveals that LARS is efficient, scalable, and capable of producing recommendations twice as accurate compared to existing recommendation approaches.**

spatial ratings item. non-spatial

I. Introduction

Recommender systems make use of community opinions to help users identify useful items from a considerably large search space (e.g., Amazon inventory [1], Netflix movies [2]). The technique used by many of these systems is collaborative filtering (CF) [3], which analyzes past community opinions to find correlations of similar users and items to suggest k personalized items (e.g., movies) to a querying user u. Community opinions are expressed through explicit ratings represented by the triple (*user*, *rating*, *item*) that represents a *user* providing a numeric *rating* for an *item*.

Currently, myriad applications can produce *location-based ratings* that embed user and/or item locations. For example, location-based social networks (e.g., Foursquare [4] and Facebook Places [5]) allow users to "check-in" at spatial destinations (e.g., restaurants) and rate their visit, thus are capable of associating both user and item locations with ratings. Such ratings motivate an interesting new paradigm of *location-aware recommendations*, whereby the recommender system exploits the spatial aspect of ratings when producing recommendations. Existing recommendation techniques [6] assume ratings are represented by the (*user*, *rating*, *item*) triple, thus are ill-equipped to produce location-aware recommendations.

This work is supported in part by the National Science Foundation under Grants IIS-0811998, IIS-0811935, CNS-0708604, IIS-0952977 and by a Microsoft Research Gift

§Work done while at the University of Minnesota

U.S. State	Top Movie Genres	Avg. Rating
Minnesota	Film-Noir	3.8
	War	3.7
	Drama	3.6
	Documentary	3.6
Wisconsin	War	4.0
	Film-Noir	4.0
	Mystery	3.9
	Romance	3.8
Florida	Fantansy	4.3
	Animation	4.1
	War	4.0
	Musical	4.0

(a) Movielens preference locality

Users from:	Visited venues in:	% Visits
Edina, MN	Minneapolis , MN	37 %
	Edina , MN	59 %
	Eden Prarie , MN	5 %
Robbinsdale, MN	Brooklyn Park, MN	32 %
	Robbinsdale, MN	20 %
	Minneapolis, MN	15 %
Falcon Heights, MN	St. Paul, MN	17 %
	Minneapolis, MN	13 %
	Roseville, MN	10 %

(b) Foursquare preference locality

Fig. 1. Preference locality in location-based ratings.

In this paper, we propose LARS, a novel location-aware recommender system built specifically to produce high-quality location-based recommendations in an efficient manner. LARS produces recommendations using a taxonomy of *three* types of location-based ratings within a single framework: (1) Spatial ratings for non-spatial items, represented as a four-tuple (*user*, *ulocation*, *rating*, *item*), where *ulocation* represents a user location, for example, a user located at home rating a book; (2) non-spatial ratings for spatial items, represented as a four-tuple (*user*, *rating*, *item*, *ilocation*), where *ilocation* represents an item location, for example, a user with unknown location rating a restaurant; (3) spatial ratings for spatial items, represented as a five-tuple (*user*, *ulocation*, *rating*, *item*, *ilocation*), for example, a user at his/her office rating a restaurant visited for lunch. Traditional rating triples can be classified as non-spatial ratings for non-spatial items and do not fit this taxonomy.

A. Motivation: A Study of Location-Based Ratings

The motivation for our work comes from analysis of two real-world location-based rating datasets: (1) a subset of the well-known MovieLens dataset [7] containing approximately 87K movie ratings associated with user zip codes (i.e., spatial ratings for non-spatial items) and (2) data from the Foursquare [4] location-based social network containing user visit data for 1M users to 643K venues across the United States (i.e., spatial ratings for spatial items). Appendix B provides further details of both datasets. In our analysis we consistently observed two interesting properties that motivate the need for location-aware recommendation techniques.

Preference locality. Preference locality suggests users from a spatial region (e.g., neighborhood) prefer items (e.g., movies, destinations) that are manifestly different than items preferred by users from other, even adjacent, regions. Figure 1(a) lists the top-4 movie genres using average MovieLens ratings of users from different U.S. states. While each list is different, the top genres from Florida differ vastly from the others.

Florida's list contains three genres ("Fantasy", "Animation", "Musical") not in the other lists. This difference implies movie preferences are unique to specific spatial regions, and confirms previous work from the New York Times [8] that analyzed Netflix user queues across U.S. zip codes and found similar differences. Meanwhile, Figure 1(b) summarizes our observation of preference locality in Foursquare by depicting the visit destinations for users from three *adjacent* Minnesota cities. Each sample exhibits diverse behavior: users from Falcon Heights, MN favor venues in St. Paul, MN (17% of visits) Minneapolis (13%), and Roseville, MN (10%), while users from Robbinsdale, MN prefer venues in Brooklyn Park, MN (32%) and Robbinsdale (20%). Preference locality suggests that recommendations should be influenced by location-based ratings *spatially close* to the user. The intuition is that localization influences recommendation using the unique preferences found within the spatial region containing the user.

Travel locality. Our second observation is that, when recommended items are spatial, users tend to travel a limited distance when visiting these venues. We refer to this property as "travel locality." In our analysis of Foursquare data, we observed that 45% of users travel 10 miles or less, while 75% travel 50 miles or less. This observation suggests that spatial items closer in travel distance to a user should be given precedence as recommendation candidates. In other words, a recommendation loses efficacy the further a querying user must travel to visit the destination. Existing recommendation techniques do not consider travel locality, thus may recommend users destinations with burdensome travel distances (e.g., a user in Chicago receiving restaurant recommendations in Seattle).

B. Our Contribution: LARS - A Location-Aware Recommender

Like traditional recommender systems, LARS suggests k items personalized for a querying user u. However, LARS is distinct in its ability to produce location-aware recommendations using *each* of the three types of location-based rating within a *single* framework.

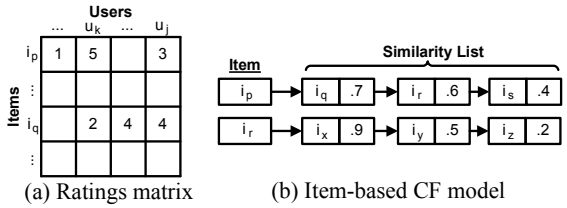
LARS produces recommendations using *spatial ratings for non-spatial items*, i.e., the tuple (*user*, *ulocation*, *rating*, *item*), by employing a *user partitioning* technique that exploits preference locality. This technique uses an adaptive pyramid structure to partition ratings by their *user location* attribute into spatial regions of varying sizes at different hierarchies. For a querying user located in a region R, we apply an existing collaborative filtering technique that utilizes only the ratings located in R. The challenge, however, is to determine whether all regions in the pyramid must be maintained in order to balance two contradicting factors: *scalability* and *locality*. Maintaining a large number of regions increases *locality* (i.e., recommendations unique to smaller spatial regions), yet adversely affects system *scalability* because each region requires storage and maintenance of a collaborative filtering data structure necessary to produce recommendations (i.e., the recommender model). The LARS pyramid dynamically adapts to find the right pyramid shape that balances scalability and recommendation locality.

LARS produces recommendations using *non-spatial ratings for spatial items*, i.e., the tuple (*user*, *rating*, *item*, *ilocation*), by using *travel penalty*, a technique that exploits travel locality. This technique penalizes recommendation candidates the further they are in travel distance to a querying user. The challenge here is to avoid computing the travel distance for all spatial items to produce the list of k recommendations, as this will greatly consume system resources. LARS addresses this challenge by employing an efficient query processing framework capable of terminating early once it discovers that the list of k answers cannot be altered by processing more recommendation candidates. To produce recommendations using *spatial ratings for spatial items*, i.e., the tuple (*user*, *ulocation*, *rating*, *item*, *ilocation*) LARS employs both the *user partitioning* and *travel penalty* techniques to address the user and item locations associated with the ratings. This is a salient feature of LARS, as the two techniques can be used separately, or in concert, depending on the location-based rating type available in the system.

We experimentally evaluate LARS using real location-based ratings from Foursquare [4] and MovieLens [7], along with a generated user workload of both *snapshot* and *continuous* queries. Our experiments show LARS is scalable to real large-scale recommendation scenarios. Since we have access to real data, we also evaluate recommendation *quality* by building LARS with 80% of the spatial ratings and testing recommendation accuracy with the remaining 20% of the (withheld) ratings. We find LARS produces recommendations that are *twice* as accurate (i.e., able to better predict user preferences) compared to traditional collaborative filtering. In summary, the contributions of this paper are as follows:

- We provide a novel classification of three types of location-based ratings not supported by existing recommender systems: *spatial ratings for non-spatial items*, *non-spatial ratings for spatial items*, and *spatial ratings for spatial items*.
- We propose LARS, a novel location-aware recommender system capable of using three classes of location-based ratings. Within LARS, we propose: (a) a *user partitioning* technique that exploits user locations in a way that maximizes system scalability while not sacrificing recommendation locality and (b) a *travel penalty* technique that exploits item locations and avoids exhaustively processing all spatial recommendation candidates.
- We provide experimental evidence that LARS scales to large-scale recommendation scenarios and provides better quality recommendations than traditional approaches.

This paper is organized as follows: Section II gives an overview of LARS. Sections III, IV, and V cover LARS recommendation techniques using *spatial ratings for non-spatial items*, *non-spatial ratings for spatial items*, and *spatial ratings for spatial items*, respectively. Section VI provides experimental analysis. Section VII covers related work, while Section VIII concludes the paper.



(a) Ratings matrix (b) Item-based CF model

Fig. 2. Item-based CF model generation.

II. LARS Overview

This section provides an overview of LARS by discussing the query model and the collaborative filtering method.

A. LARS Query Model

Users (or applications) provide LARS with a user id U, numeric limit K, and location L; LARS then returns K recommended items to the user. LARS supports both *snapshot* (i.e., one-time) queries and *continuous* queries, whereby a user subscribes to LARS and receives recommendation updates as her location changes. The technique LARS uses to produce recommendations depends on the type of location-based rating available in the system. Query processing support for each type of location-based rating is discussed in Sections III to V.

B. Item-Based Collaborative Filtering

LARS uses item-based collaborative filtering (abbr. CF) as its primary recommendation technique, chosen due to its popularity and widespread adoption in commercial systems (e.g., Amazon [1]). Collaborative filtering (CF) assumes a set of n users $\mathcal{U} = \{u_1, ..., u_n\}$ and a set of m items $\mathcal{I} = \{i_1, ..., i_m\}$. Each user u_j expresses opinions about a set of items $\mathcal{I}_{u_j} \subseteq \mathcal{I}$. Opinions can be a numeric rating (e.g., the Netflix scale of one to five stars [2]), or unary (e.g., Facebook "check-ins" [5]). Conceptually, ratings are represented as a matrix with users and items as dimensions, as depicted in Figure 2(a). Given a querying user u, CF produces a set of k recommended items $\mathcal{I}_r \subset \mathcal{I}$ that u is predicted to like the most.

Phase I: Model Building. This phase computes a similarity score *sim*(i_p,i_q) for each pair of objects i_p and i_q that have at least one common rating by the same user (i.e., co-rated dimensions). Similarity computation is covered below. Using these scores, a model is built that stores for each item $i \in \mathcal{I}$, a list $\mathcal{L}$ of similar items ordered by a similarity score *sim*(i_p,i_q), as depicted in Figure 2(b). Building this model is an $O(\frac{R^2}{U})$ process, where R and U are the number of ratings and users, respectively. It is common to truncate the model by storing, for each list $\mathcal{L}$, only the n most similar items with the highest similarity scores [9]. The value of n is referred to as the *model size* and is usually much less than $|\mathcal{I}|$.

Phase II: Recommendation Generation. Given a querying user u, recommendations are produced by computing u's predicted rating $P_{(u,i)}$ for each item i not rated by u [9]:

$$P_{(u,i)} = \frac{\sum_{l \in \mathcal{L}} sim(i,l) * r_{u,l}}{\sum_{l \in \mathcal{L}} |sim(i,l)|} \quad (1)$$

Before this computation, we reduce each similarity list $\mathcal{L}$ to contain only items *rated* by user u. The prediction is the sum of $r_{u,l}$, a user u's rating for a related item $l \in \mathcal{L}$ weighted by *sim(i,l)*, the similarity of l to candidate item i, then normalized by the sum of similarity scores between i and l. The user receives as recommendations the top-k items ranked by $P_{(u,i)}$.

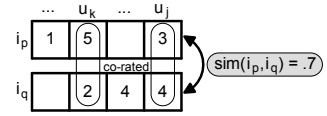


Fig. 3. Item-based similarity calculation.

Computing Similarity. To compute *sim*(i_p, i_q), we represent each item as a vector in the user-rating space of the rating matrix. For instance, Figure 3 depicts vectors for items i_p and i_q from the matrix in Figure 2(a). Many similarity functions have been proposed (e.g., Pearson Correlation, Cosine); we use the Cosine similarity in *LARS* due to its popularity:

$$sim(i_p, i_q) = \frac{\vec{i_p} \cdot \vec{i_q}}{\|\vec{i_p}\| \|\vec{i_q}\|} \quad (2)$$

This score is calculated using the vectors' co-rated dimensions, e.g., the Cosine similarity between i_p and i_q in Figure 3 is .7 calculated using the circled co-rated dimensions. Cosine distance is useful for numeric ratings (e.g., on a scale [1,5]). For unary ratings, other similarity functions are used (e.g., absolute sum [10]).

While we opt to use item-based CF in this paper, no factors disqualify us from employing other recommendation techniques. For instance, we could easily employ user-based CF [6], that uses correlations between users (instead of items).

III. Spatial User Ratings for Non-Spatial Items

어떤 측정단위에 대해 선호도가 다름. 지역 ← 선호도

This section describes how LARS produces recommendations using spatial ratings for non-spatial items represented by the tuple (*user*, *ulocation*, *rating*, *item*). The idea is to exploit *preference locality*, i.e., the observation that user opinions are spatially unique (based on analysis in Section I-A). We identify three requirements for producing recommendations using spatial ratings for non-spatial items: (1) *Locality*: recommendations should be influenced by those ratings with user locations spatially close to the querying user location (i.e., in a spatial neighborhood); (2) *Scalability*: the recommendation procedure and data structure should scale up to large number of users; (3) *Influence*: system users should have the ability to control the size of the spatial neighborhood (e.g., city block, zip code, or county) that influences their recommendations.

LARS achieves its requirements by employing a *user partitioning* technique that maintains an adaptive pyramid structure, where the shape of the adaptive pyramid is driven by the three goals of *locality*, *scalability*, and *influence*. The idea is to adaptively partition the rating tuples (*user*, *ulocation*, *rating*, *item*) into spatial regions based on the *ulocation* attribute. Then, LARS produces recommendations using any existing collaborative filtering method (we use item-based CF) over the remaining three attributes (*user*, *rating*, *item*) of *only* the ratings within the spatial region containing the querying user. We note that ratings can come from users with

ulocation을 rating tuples → spatial regions로 나눈다.
그다음. LARS CF로 recommendation.
item based

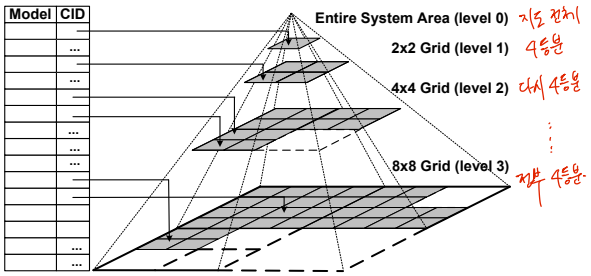


Fig. 4. Partial pyramid data structure.

varying tastes, and that our method only forces collaborative filtering to produce personalized user recommendations based only on ratings restricted to a specific spatial region. In this section, we describe the pyramid structure in Section III-A, query processing in Section III-B, and finally data structure maintenance in Section III-C.

A. Data Structure

LARS employs a partial pyramid structure [11] (equivalent to a partial quad-tree [12]) as depicted in Figure 4. The pyramid decomposes the space into H levels (i.e., pyramid height). For a given level h, the space is partitioned into 4^h equal area grid cells. For example, at the pyramid root (level 0), one grid cell represents the entire geographic area, level 1 partitions space into four equi-area cells, and so forth. We represent each cell with a unique identifier cid. In each cell, we store an item-based collaborative filtering model built using *only* the spatial ratings with user locations contained in the cell's spatial region. A rating may contribute to up to H collaborative filtering models: one per each pyramid level starting from the lowest maintained grid cell containing the embedded user location up to the root level. Note that the root cell (level 0) of the pyramid represents a "traditional" (i.e., non-spatial) item-based collaborative filtering model. Levels in the pyramid can be incomplete, as LARS will periodically merge or split cells based on trade-offs of locality and scalability (discussed in Section III-C). For example, in Figure 4, the four cells in the upper right corner of level 3 are not maintained (depicted as blank white squares).

We chose to employ a pyramid as it is a "space-partitioning" structure that is guaranteed to completely cover a given space. For our purposes, "data-partitioning" structures (e.g., R-trees) are less ideal, as they index data points and are not guaranteed to completely cover a given space.

B. Query Processing

Given a recommendation query (as described in Section II-A) with user location L and a limit K, LARS performs two query processing steps: (1) The user location L is used to find the lowest maintained cell C in the adaptive pyramid that contains L. This is done by hashing the user location to retrieve the cell at the lowest level of the pyramid. If this cell is not maintained, we return the nearest maintained ancestor cell. (2) The top-k recommended items are generated using the item-based collaborative filtering technique (covered in Section II-B) using the model stored at C. As mentioned earlier, the model in C is built using *only* the spatial ratings associated with user locations within C.

In addition to traditional recommendation queries (i.e., snapshot queries), LARS also supports continuous queries and can account for the *influence* requirement for each user as follows.

Continuous queries. LARS evaluates a continuous query in full once it is issued, and sends recommendations back to a user U as an initial answer. LARS then monitors the movement of U using her location updates. As long as U does not cross the boundary of her current grid cell, LARS does nothing as the initial answer is still valid. Once U crosses a cell boundary, LARS reevaluates the recommendation query for the new cell and only sends incremental updates [13] to the last reported answer. Like snapshot queries, if a cell at level h is not maintained, the query is temporarily transferred higher in the pyramid to the nearest maintained ancestor cell. Note that since higher-level cells maintain larger spatial regions, the continuous query will cross spatial boundaries less often, reducing the amount of required recommendation updates.

Influence level. LARS addresses the *influence* requirement by allowing querying users to specify an optional *influence level* (in addition to location L and limit K) that controls the size of the spatial neighborhood used to influence their recommendations. An influence level I maps to a pyramid level and acts much like a "zoom" level in Google or Bing maps (e.g., city block, neighborhood, entire city). The level I instructs LARS to process the recommendation query starting from the grid cell containing the querying user location at level I, instead of the lowest maintained grid cell (the default). An influence level of zero forces LARS to use the root cell of the pyramid, and thus act as a traditional (non-spatial) collaborative filtering recommender system.

C. Data Structure Maintenance

This section describes building and maintaining the pyramid data structure. Initially, to build the pyramid, all location-based ratings currently in the system are used to build a *complete pyramid* of height H, such that all cells in all H levels are present and contain a collaborative filtering model. The initial height H is chosen according to the level of *locality* desired, where the cells in the lowest pyramid level represent the most localized regions. After this initial build, we invoke a *merging* step that scans all cells starting from the lowest level h and merges quadrants (i.e., four cells with a common parent) into their parent at level $h-1$ if it is determined that a tolerated amount of locality will not be lost (merging is discussed in Section III-C1). We note that while the original partial pyramid [11] was concerned with spatial queries over static data, it did not address pyramid maintenance.

As time goes by, new users, ratings, and items will be added to the system. This new data will both increase the size of the collaborative filtering models maintained in the pyramid cells, as well as alter recommendations produced from each cell. To account for these changes, LARS performs maintenance on a cell-by-cell basis. Maintenance is triggered for a cell C

Algorithm 1 Pyramid maintenance algorithm

```
1:  /* Called after cell C receives N% new ratings */
2:  Function PyramidMaintenance(Cell C, Level h)
3:  /*Step I: Model Rebuild */
4:  Rebuild item-based collaborative filtering model for cell C
5:  /*Step II: Merge/Split Maintenance */
6:  if (C has children quadrant q maintained at level h + 1) then
7:      if (All cells in q have no maintained children) then
8:          CheckDoMerge(q,C) /* Merge covered in Section III-C1 */
9:      end if
10: else
11:     CheckDoSplit(C) /* Split covered in Section III-C2 */
12: end if
13: return
```

once it receives $N\%$ new ratings; the percentage is computed from the number of existing ratings in C. We do this because an appealing quality of collaborative filtering is that as a model matures (i.e., more data is used to build the model), more updates are needed to significantly change the top-k recommendations produced from it [14]. Thus, maintenance is needed less often. Algorithm 1 provides the pseudocode for the LARS maintenance algorithm. The algorithm takes as input a pyramid cell C and level h, and includes two main steps: *model rebuild* and *merge/split maintenance*.

Step I: Model Rebuild. The first step is to rebuild the item-based collaborative filtering (CF) model for a cell C, as described in Section II-B (line 4). Rebuilding the CF model is necessary to allow the model to "evolve" as new location-based ratings enter the system (e.g., accounting for new items, ratings, or users). Given the cost of building the CF model is $O(\frac{R^2}{U})$ (per Section II-B), the cost of the model rebuild for a cell C at level h is $\frac{(R/4^h)^2}{(U/4^h)}=\frac{R^2}{4^h U}$, assuming ratings and users are uniformly distributed.

Step II: Merging/Split Maintenance. After rebuilding the CF model for cell C, LARS invokes a merge/split maintenance step that may decide to merge or split cells based on trade-offs in *scalability* and *locality*. The algorithm first checks if C has a child quadrant q maintained at level $h+1$ (line 6), and that none of the four cells in q have maintained children of their own (line 7). If both cases hold, LARS considers quadrant q as a candidate to merge into its parent cell C (calling function *CheckDoMerge* on line 8). We provide details of merging in Section III-C1. On the other hand, if C does *not* have a child quadrant maintained at level $h+1$ (line 10), LARS considers splitting C into four child cells at level $h+1$ (calling function *CheckDoSplit* on line 11). The split operation is covered in Section III-C2. Merging and splitting are performed completely in quadrants (i.e., four equi-area cells with the same parent). We made this decision for simplicity in maintaining the partial pyramid. However, we also discuss (in Section III-D) relaxing this constraint by merging and splitting at a finer granularity than a quadrant.

We note the following features of pyramid maintenance: (1) Maintenance can be performed completely offline, i.e., LARS can continue to produce recommendations using the "old" pyramid cells while part of the pyramid is being updated; (2) maintenance does not entail rebuilding the whole pyramid at once, instead, only one cell is rebuilt at a time; (3) maintenance is performed only after $N\%$ new ratings are added to a pyramid cell, meaning maintenance will be amortized over many operations.

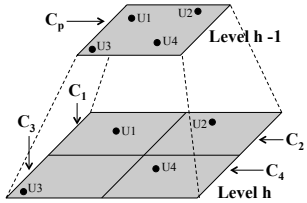
1) Cell Merging: Merging entails discarding an entire quadrant of cells at level h with a common parent at level $h-1$. Merging improves scalability (i.e., storage and computational overhead) of LARS, as it reduces storage by discarding the item-based collaborative filtering (CF) models of the merged cells. Furthermore, merging improves computational overhead in two ways: (a) *less maintenance computation*, since less CF models are periodically rebuilt, and (b) *less continuous query processing computation*, as merged cells represent a larger spatial region, hence, users will cross cell boundaries less often triggering less recommendation updates. Merging hurts locality, since merged cells capture community opinions from a wider spatial region, causing less unique (i.e., "local") recommendations than smaller cells.

To determine whether to merge a quadrant q into its parent cell C_P (i.e., function *CheckDoMerge* on line 8 in Algorithm 1), we calculate two percentage values: (1) *locality_loss*, the amount of locality lost by (potentially) merging, and (2) *scalability_gain*, the amount of scalability gained by (potentially) merging. Details of calculating these percentages are covered next. When deciding to merge, we define a system parameter $\mathcal{M}$, a real number in the range [0,1] that defines a tradeoff between scalability gain and locality loss. LARS merges (i.e., discards quadrant q) if:

$$(1-\mathcal{M}) * scalability_gain > \mathcal{M} * locality_loss \quad (3)$$

A smaller $\mathcal{M}$ value implies gaining scalability is important and the system is willing to lose a large amount of locality for small gains in scalability. Conversely, a larger $\mathcal{M}$ value implies scalability is not a concern, and the amount of locality lost must be small in order to merge. At the extremes, setting $\mathcal{M}$=0 (i.e., always merge) implies LARS will function as a traditional CF recommender system, while setting $\mathcal{M}$=1 causes LARS to never merge, i.e., LARS will employ a complete pyramid structure maintaining all cells at all levels.

Calculating Locality Loss. We calculate locality loss by observing the loss of recommendation uniqueness when discarding a cell quadrant q and using its parent cell C_P to produce recommendations in its place. We perform this calculation in three steps. (1) *Sample*. We take a sample of diverse system users $\mathcal{U}$ that have at least one rating within C_P (and by definition one of the more localized cells $C_u \in q$). Due to space, we do not discuss user sampling in detail, however, the intuition is to select a set of users with *diverse* tastes by comparing each user's rating history. We measure diversity using the Cosine distance between users in the same manner as Equation 2, except we employ user vectors in the calculation (instead of item vectors). (2) *Compare*. For each user $u \in \mathcal{U}$, we measure the potential loss of recommendation uniqueness by comparing the list of top-k recommendations R_P produced from the merged cell C_P (i.e., the parent) with the list of recommendations R_u that the user receives from the more localized cell $C_u \in q$. Formally, the loss of uniqueness



User	Recommendation		Locality Loss
	C_u	C_p	
U_1	I_1, I_2, I_5, I_6	I_1, I_2, I_5, I_7	25%
U_2	I_1, I_2, I_3, I_4	I_1, I_2, I_3, I_5	25%
U_3	I_3, I_4, I_5, I_6	I_3, I_4, I_5, I_6	0%
U_4	I_3, I_4, I_6, I_8	I_3, I_4, I_5, I_7	50%
Average Locality Loss			25%

Fig. 5. Merge and split example.

can be computed as the ratio $\frac{|R_u - R_P|}{k}$, which indicates the number of recommended items that appear in R_u but not in the parent recommendation R_P, normalized to the total number of recommended objects k. (3) *Average*. We calculate the average loss of uniqueness over all users in $\mathcal{U}$ to produce a single percentage value, termed *locality_loss*.

Calculating scalability gain. Scalability gain is measured in storage and computation savings. We measure scalability gain by summing the model sizes for each of the merged (i.e., child) cells (abbr. $size_m$), and divide this value by the sum of $size_m$ and the size of the parent cell. We refer to this percentage as the *storage_gain*. We also quantify *computation* savings using storage gain as a surrogate measurement, as computation is considered a direct function of the amount of data in the system.

Cost. The cost of *CheckDoMerge* is $|\mathcal{U}|(2(\frac{n|\mathcal{I}|}{4^h})+k)$, where $|\mathcal{U}|$ is the size of the user sample, $|\mathcal{I}|$ is the number of items in the model stored within a cell, n is the model size (Section II-B), and k is the cost of comparing two recommendation lists. We note that this cost is less than the model re-build step.

Example. Figure 5 depicts four merge candidate cells C_1 to C_4 at level h merging into their parent C_P at level $h-1$, along with four sampled users u_1 to u_4. Each user location is shown twice: once within one of the cells C_u and then at the parent cell C_P. The recommendations produced for each user from cell C_u and C_P are provided in the table in Figure 5, along with the locality loss for each user. For example, for user u_1, cell C_1 produces recommendations R_{u_1}={I_1, I_2, I_5, I_6}, while C_P produces recommendations R_P={I_1, I_2, I_5, I_7}. Thus, the loss of locality for u_1 is 25% as only one item out of four (I_6) will be lost if merging occurs. Given locality loss for the four users u_1 to u_4 as 25%, 25%, 0%, and 50%, the final *locality loss* value is the average 25%. To calculate scalability gain, assume the sum of the model sizes for cells C_1 to C_4 and C_P is 4GB, and the sum of the model sizes for cells C_1 to C_4 is 2GB. Then, the *scalability gain* is $\frac{2}{4}$=50%. Assuming $\mathcal{M}$=0.7, then (0.3 * 50) $<$ (0.7 * 25), meaning that LARS will not merge cells C_1, C_2, C_3, C_4 into C_P.

2) Splitting: Splitting entails creating a new cell quadrant at pyramid level h under a cell at level $h-1$. Splitting improves locality in LARS, as newly split cells represent more granular spatial regions capable of producing recommendations unique to the smaller, more "local", spatial regions. On the other hand, splitting hurts scalability by requiring storage and maintenance of more item-based collaborative filtering models. Splitting also negatively affects continuous query processing, since it creates more granular cells causing user locations to cross cell boundaries more often, triggering recommendation updates.

To determine whether to split a cell C_P into four child cells (i.e., function *CheckDoSplit* on line 11 of Algorithm 1), we perform a *speculative split* that creates a temporary child quadrant q_s for C_P. Using C_P and q_s, two percentages are calculated: *locality_gain* and *scalability_loss*. These values are the opposite of those calculated for the merge operation. LARS splits C_P only if the following condition holds:

$$\mathcal{M} * locality_gain > (1-\mathcal{M}) * scalability_loss \quad (4)$$

This equation represents the opposite criteria of that presented for merging in Equation 3. We will next describe how to perform speculative splitting, followed by a description of how to calculate *locality gain* and *scalability loss*.

Speculative splitting. In order to evaluate *locality gain* and *scalability loss*, we must build, from scratch, the collaborative filtering (CF) models of the four cells that potentially result from the split, as they do not exist in the partial pyramid. As building CF models is non-trivial due to its high cost (Section II-B), we perform a cheaper *speculative split* that builds each model using a random sample of only 50% of the ratings from the spatial region of each potentially split cell. LARS uses these models to measure *locality_gain* and *scalability_loss*. If LARS decides to split, it builds the *complete* model for the newly split cells using all of the ratings. Speculative splitting is sufficient for calculating *locality_gain* and *scalability_loss* using the item-based CF technique, as experiments on real data and workloads have shown that using 50% of the ratings for model-building results in loss of only 3% of recommendation accuracy [9], assuming sufficiently high number of ratings (i.e., order of thousands). Thus, we *only* speculatively split if we have more than 1,000 ratings for the potentially split cell, otherwise, the model for the cell is built using all of R.

Calculating locality gain. After speculatively splitting a cell at level h into four child cells at level $h+1$, evaluating locality gain is performed exactly the same as for merging, where we compute the ratio of recommendations that will appear in R_u but not in R_P, where R_u and R_P are the list of top-k recommendations generated by the speculatively split cells C_1 to C_4 and the existing parent cell C_P, respectively. Like the merging case, we average locality gain over all sampled users. One caveat here is that if *any* of the speculatively split cells do not contain ratings for enough unique items (say less than ten unique items), we immediately set the locality gain to 0, which disqualifies splitting. We do this to prevent *recommendation starvation*, i.e., not having enough diverse items to produce meaningful recommendations.

Calculating scalability loss. We calculate *scalability loss* by estimating the storage necessary to maintain the newly split cells. Recall from Section II-B that the maximum size of an item-based CF model is approximately $n|I|$, where n is the model size. We can multiply $n|I|$ by the number of bytes needed to store an item in a CF model to find an upper-bound storage size of each potentially split cell. The sum of these four

estimated sizes (abbr. $size_s$) divided by the sum of the size of the existing parent cell and $size_s$ represents the *scalability loss* metric.

Cost. The cost of *CheckDoSplit* is the sum of two operations (1) the cost of speculatively building four CF models at level $h+1$ using 50% of the rating, which is $4\frac{(0.5R)^2}{4^{(h+1)}U}$ (per Section II-B) and (2) the cost of calculating locality gain and scalability loss, which is the same cost as *CheckDoMerge*.

Example. Consider the example used for merging in Figure 5, but now assume we have *only* a cell C_P, and are trying to determine whether to split C_P into four new cells C_1 to C_4. *Locality gain* will be computed as in the table in Figure 5 to be 25%. Further, assume that we estimate the extra storage overhead for splitting (i.e., *storage loss*) to be 50%. Assuming $\mathcal{M}$=0.7, then (0.7 * 25) $>$ (0.3 * 50), meaning that LARS will decide to split C_P into four cells as *locality gain* is significantly higher than *scalability loss*.

D. Partial Merging and Splitting

So far, we have assumed cells are merged and split in complete quadrants. We now relax this constraint by discussing the changes to LARS necessary to support *partial* merging and splitting of pyramid cells.

1) Partial Merging: It may be beneficial to *partially* merge at a more granular level in order to sacrifice *less* locality while still gaining scalability. For example, in Figure 5 we may only want to merge cells C_1 and C_2 while leaving cells C_3 and C_4 intact, meaning three child cells would be maintained under the example parent C_P. To support partial merging, all techniques described in Section III-C1 remain the same, with two exceptions: (1) The resulting merged candidate cell (e.g., C_1 merged with C_2, abbreviated C_{12}) plays the role of the "parent" cell in evaluating locality loss; (2) When calculating storage gain, we must subtract the size of the resulting merge candidate cell (e.g., C_{12}) from the sum of the sizes of cells that will merge (e.g., C_1 and C_2), since we no longer discard the merged cells completely, i.e., the resulting merged cell now replaces the individual cells.

Partial merging involves extra overhead (compared to merging complete quadrants) since we must build, from scratch, the CF model for the candidate merge result cell (e.g., C_{12}) in order to calculate locality loss. In order to perform the build efficiently, we perform a *speculative merge* that builds the CF model using only 50% of the rating data. This is the *same* method used in *speculative splitting* (Section III-C2), except applied to the case of merging.

2) Partial Splitting: To support *partial splitting*, all techniques discussed in Section III-C2 remain the same. There are, however, two distinguishable cases of partial splitting: (1) A "parent" at level h splitting into less than four cells at level $h+1$. This case requires speculative splitting to be aware of which "partial" child cells to create. (2) A cell at level h is split into two or three separate cells that remain at level h, i.e., cells at level $h+1$ are not created. This case requires that a previous partial merge took place that originally reduced a cell quadrant to two or three cells.

IV. Non-Spatial User Ratings for Spatial Items

This section describes how LARS produces recommendations using non-spatial ratings for spatial items represented by the tuple (*user*, *rating*, *item*, *ilocation*). The idea is to exploit *travel locality*, i.e., the observation that users limit their choice of spatial venues based on travel distance (based on analysis in Section I-A). Traditional (non-spatial) recommendation techniques may produces recommendations with burdensome travel distances (e.g., hundreds of miles away). LARS produces recommendations within reasonable travel distances by using *travel penalty*, a technique that penalizes the recommendation rank of items the further in travel distance they are from a querying user. *Travel penalty* may incur expensive computational overhead by calculating travel distance to each item. Thus, LARS employs an efficient query processing technique capable of *early termination* to produce the recommendations without calculating the travel distance to all items. Section IV-A describes the query processing framework while Section IV-B describes travel distance computation.

A. Query Processing

Query processing for spatial items using the *travel penalty* technique employs a single system-wide item-based collaborative filtering model to generate the top-k recommendations by ranking each spatial item i for a querying user u based on $RecScore(u,i)$, computed as:

이동해야하는 거리로 rating scale

$$RecScore(u,i) = P(u,i) - TravelPenalty(u,i) \quad (5)$$

$P(u,i)$ is the standard item-based CF predicted rating of item i for user u (see Section II-B). $TravelPenalty(u,i)$ is the road network travel distance between u and i normalized to the same value range as the rating scale (e.g., [0, 5]).

When processing recommendations, we aim to avoid calculating Equation 5 for *all* candidate items to find the top-k recommendations, which can become quite expensive given the need to compute travel distances. To avoid such computation, we evaluate items in monotonically increasing order of travel penalty (i.e., travel distance), enabling us to use early termination principles from top-k query processing [15], [16], [17]. We now present the main idea of our query processing algorithm and in the next section discuss how to compute travel penalties in an increasing order of travel distance.

Algorithm 2 provides the pseudo code of our query processing algorithm that takes a querying user id U, a location L, and a limit K as input, and returns the list R of top-k recommended items. The algorithm starts by running a k-nearest-neighbor algorithm to populate the list R with k items with lowest travel penalty; R is sorted by the recommendation score computed using Equation 5. This initial part is concluded by setting the lowest recommendation score value (*LowestRecScore*) as the *RecScore* of the kth item in R (Lines 3 to 8). Then, the algorithm starts to retrieve items one by one in the order of their penalty score. This can be done using an *incremental* k-nearest-neighbor algorithm, as will be described in the next section. For each item i, we calculate the *maximum*

Algorithm 2 Travel Penalty Algorithm for Spatial Items

```
1:  Function LARS_SpatialItems(User U, Location L, Limit K)
2:  /* Populate a list R with a set of K items*/
3:  R ← φ
4:  for (K iterations) do
5:      i ← Retrieve the item with the next lowest travel penalty (Section IV-B)
6:      Insert i into R ordered by RecScore(U, i) computed by Equation 5
7:  end for
8:  LowestRecScore ← RecScore of the k^th object in R
9:  /*Retrieve items one by one in order of their penalty value */
10: while there are more items to process do
11:     i ← Retrieve the next item in order of penalty score (Section IV-B)
12:     MaxPossibleScore ← MAX_RATING - i.penalty
13:     if MaxPossibleScore ≤ LowestRecScore then
14:         return R /* early termination - end query processing */
15:     end if
16:     RecScore(U, i) ← P(U, i) - i.penalty /* Equation 5 */
17:     if RecScore(U, i) > LowestRecScore then
18:         Insert i into R ordered by RecScore(U, i)
19:         LowestRecScore ← RecScore of the k^th object in R
20:     end if
21: end while
22: return R
```

possible recommendation score that i can have by subtracting the travel penalty of i from *MAX_RATING*, the maximum possible rating value in the system, e.g., 5 (Line 12). If i cannot make it into the list of top-k recommended items with this maximum possible score, we immediately terminate the algorithm by returning R as the top-k recommendations without computing the recommendation score (and travel distance) for more items (Lines 13 to 15). The rationale here is that since we are retrieving items in increasing order of their penalty and calculating the maximum score that any remaining item can have, then there is no chance that any unprocessed item can beat the lowest recommendation score in R. If the early termination case does not arise, we continue to compute the score for each item i using Equation 5, insert i into R sorted by its score (removing the kth item if necessary), and adjust the lowest recommendation value accordingly (Lines 16 to 20).

Travel penalty requires very little maintenance. The only maintenance necessary is to occasionally rebuild the single system-wide item-based collaborative filtering model in order to account for new location-based ratings that enter the system. Following the reasoning discussed in Section III-C, we rebuild the model after receiving $N\%$ new location-based ratings.

B. Incremental Travel Penalty Computation

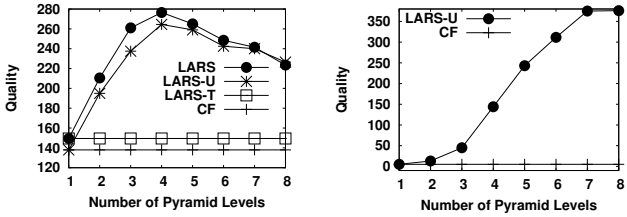
This section gives an overview of two methods we implemented in LARS to incrementally retrieve items one by one ordered by their travel penalty. The two methods exhibit a tradeoff between query processing efficiency and penalty accuracy: (1) an *online* method that provides exact travel penalties but is expensive to compute, and (2) an *offline* heuristic method that is less exact but efficient in penalty retrieval. Both methods can be employed interchangeably in Line 11 of Algorithm 2.

1) Incremental KNN: An Exact Online Method: To calculate an exact travel penalty for a user u to item i, we employ an incremental k-nearest-neighbor (KNN) technique [18], [19], [20]. Given a user location l, incremental KNN algorithms return, on each invocation, the next item i nearest to u with regard to travel distance d. In our case, we normalize distance d to the ratings scale to get the travel penalty in Equation 5. Incremental KNN techniques exist for both Euclidean distance [19] and (road) network distance [18], [20]. The advantage of using Incremental KNN techniques is that they provide an *exact* travel distances between a querying user's location and each recommendation candidate item. The disadvantage is that distances must be computed *online* at query runtime, which can be expensive. For instance, the runtime complexity of retrieving a single item using incremental KNN in Euclidean space is [19]: $O(k+logN)$, where N and k are the number of total items and items retrieved so far, respectively.

2) Penalty Grid: A Heuristic Offline Method: A more efficient, yet less accurate method to retrieve travel penalties incrementally is to use a pre-computed *penalty grid*. The idea is to partition space using an $n \times n$ grid. Each grid cell c is of equal size and contains all items whose location falls within the spatial region defined by c. Each cell c contains a *penalty list* that stores the pre-computed penalty values for traveling from anywhere within c to all other n^2-1 destination cells in the grid; this means all items within a destination grid cell share the *same* penalty value. The penalty list for c is sorted by penalty value and always stores c (itself) as the first item with a penalty of zero. To retrieve items incrementally, all items within the cell containing the querying user are returned one-by-one (in any order) since they have no penalty. After these items are exhausted, items contained in the next cell in the penalty list are returned, and so forth until Algorithm 2 terminates early or processes all items.

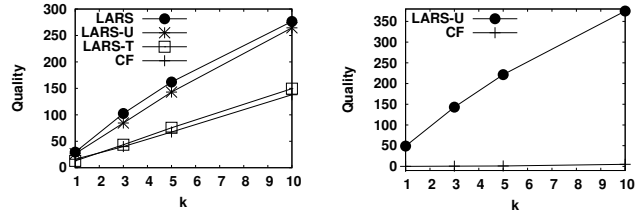
To populate the penalty grid, we must calculate the penalty value for traveling from each cell to every other cell in the grid. We assume items and users are constrained to a road network, however, we can also use Euclidean space without consequence. To calculate the penalty from a single source cell c to a destination cell d, we first find the average distance to travel from anywhere within c to all item destinations within d. To do this, we generate an *anchor point* p within c that both (1) lies on the road network segment within c and (2) lies as close as possible to the center of c. With these criteria, p serves as an approximate average "starting point" for traveling from c to d. We then calculate the shortest path distance from p to *all* items contained in d on the road network (any shortest path algorithm can be used). Finally, we average all calculated shortest path distances from c to d. As a final step, we normalize the average distance from c to d to fall within the rating value range. Normalization is necessary as the rating domain is usually small (e.g., zero to five), while distance is measured in miles or kilometers and can have large values that heavily influence Equation 5. We repeat this entire process for each cell to all other cells to populate the entire penalty grid.

When new items are added to the system, their presence in a cell d can alter the average distance value used in penalty calculation for each source cell c. Thus, we recalculate penalty scores in the penalty grid after N new items enter the system. We assume spatial items are relatively static, e.g., restaurants do not change location often. Thus, it is unlikely *existing* items will change cell locations and in turn alter penalty scores.



(a) Foursquare data (b) MovieLens data

Fig. 6. Quality experiments for varying locality



(a) Foursquare data (b) MovieLens data

Fig. 7. Quality experiments for varying answer sizes

V. Spatial User Ratings for Spatial Items

This section describes how LARS produces recommendations using spatial ratings for spatial items represented by the tuple (*user*, *ulocation*, *rating*, *item*, *ilocation*). A salient feature of LARS is that both the *user partitioning* and *travel penalty* techniques can be used together with very little change to produce recommendations using spatial user ratings for spatial items. The data structures and maintenance techniques remain *exactly* the same as discussed in Sections III and IV; only the query processing framework requires a slight modification. Query processing uses Algorithm 2 to produce recommendations. However, the only difference is that the item-based collaborative filtering prediction score $P(u, i)$ used in the recommendation score calculation (Line 16 in Algorithm 2) is generated using the (localized) collaborative filtering model from the partial pyramid cell that contains the querying user, instead of the system-wide collaborative filtering model as was used in Section IV.

VI. Experiments

This section provides experimental evaluation of LARS based on an actual system implementation. We compare LARS with the standard item-based collaborative filtering technique along with several variations of LARS. Experiments are based on three data sets: (1) *Foursquare*: a real data set consisting of *spatial user ratings for spatial items* derived from Foursquare user histories. (2) *MovieLens*: a real data set consisting of *spatial user ratings for non-spatial items* taken from the popular MovieLens recommender system [7]. The Foursquare and MovieLens data are used to test recommendation quality. (3) *Synthetic*: a synthetically generated data set consisting spatial user ratings for spatial items for venues in the state of Minnesota, USA; we use this data to test scalability and query efficiency. Details of all data sets are found in Appendix B.

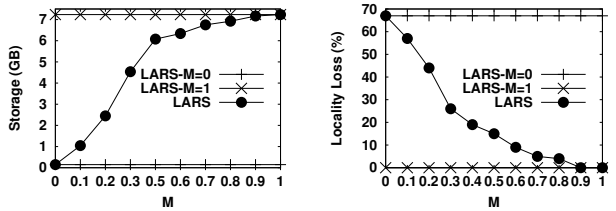
Unless mentioned otherwise, the default value of $\mathcal{M}$ is 0.3, k is 10, the number of pyramid levels is 8, and the influence level is the lowest pyramid level. The rest of this section evaluates LARS recommendation quality (Section VI-A), trade-offs between storage and locality (Section VI-C), scalability (Section VI-D), and query processing efficiency (Section VI-E).

A. Recommendation Quality for Varying Pyramid Levels

These experiments test the recommendation quality of LARS against the standard (non-spatial) item-based collaborative filtering method (denoted as CF) using both the Fourquare and MovieLens data. To test the effectiveness of our proposed techniques, we test the quality of LARS with only travel penalty enabled (abbr. LARS-T), LARS with only user partitioning enabled (abbr. LARS-U), and LARS with both techniques enabled (abbr. LARS). To measure quality, we build each recommendation method using 80% of the ratings from each data set. Each rating in the withheld 20% represents a Foursquare venue or MovieLens movie a user is known to like (i.e., rated highly). For each rating t in this 20%, we request a set of k recommendations $\mathcal{R}$ by submitting the *user* and *ulocation* associated with t. The quality measure is the count of how many times $\mathcal{R}$ contains the *item* associated with t (the higher the better). The rationale for this metric is that since each withheld rating represents a real visit to a venue (or movie a user liked), the technique that produces a large number of answers that contain venues (or movies) a user is known to like is considered of higher quality.

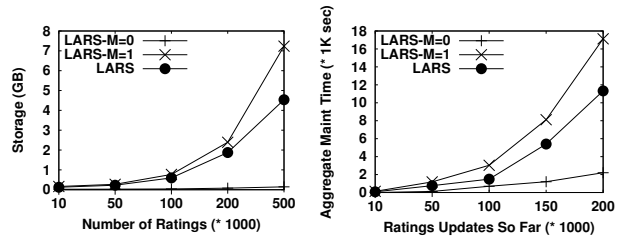
Figure 6(a) compares the quality of each technique for varying locality (i.e., different levels of the adaptive pyramid) using the Foursquare data. Both CF and LARS-T do not use the adaptive pyramid, thus have constant quality values. The gap between CF and LARS-T highlights the benefit of using the *travel penalty* technique that recommends items within a feasible distance. Meanwhile, the quality of LARS and LARS-U increases as more localized pyramid cells are used to produce recommendation, which verifies that *user partitioning* is indeed beneficial and necessary for location-based ratings. Ultimately, LARS has superior performance due to the additional use of *travel penalty*. While *travel penalty* produces moderate quality gain, it also enables more efficient query processing, which we observe later in Section VI-E).

Figure 6(b) compares the quality of LARS-U and CF for varying locality using the MovieLens data (LARS and LARS-T do not apply since movies are not spatial). While CF quality is constant, the quality of LARS-U increases when it produces movie recommendations from more localized pyramid cells. This behavior further verifies that *user partitioning* is beneficial in providing quality recommendations localized to a querying user location, even when items are not spatial. Quality decreases (or levels off for MovieLens) for both LARS-U and/or LARS for lower levels of the adaptive pyramid. This is due to *recommendation starvation*, i.e., not having enough ratings to produce meaningful recommendations.



(a) Storage (b) Locality

Fig. 8. Effect of $\mathcal{M}$ on storage and locality



(a) Storage (b) Maintenance

Fig. 9. Scalability of the adaptive pyramid

B. Recommendation Quality for Varying Values of k

These experiments test recommendation quality of LARS, LARS-U, LARS-T, and CF for different values of k (i.e., recommendation answer sizes). We perform experiments using both the Foursquare and MovieLens data. Our quality metric is exactly the same as presented previously in Section VI-A.

Figure 7(a) depicts the effect of the recommendation list size k on the quality of each technique using the Foursquare data set. We report quality numbers using the pyramid height of four (i.e., the level exhibiting the best quality from Section VI-A in Figure 6(a)). For all sizes of k from one to ten, LARS and LARS-U consistently exhibit better quality. In fact, LARS is consistently twice as accurate as CF for all k. LARS-T exhibits similar quality to CF for smaller k values, but does better for k values of three and larger.

Figure 7(b) depicts the effect of the recommendation list size k on the quality of LARS-U and CF using the MovieLens data (LARS and LARS-T do not apply in this experiment since movies are not spatial). This experiment was run using a pyramid hight of seven (i.e., the level exhibiting the best quality in Figure 6(b)). Again, LARS-U consistently exhibits better quality than CF for sizes of K from one to ten. In fact, the quality of CF increases by just a fraction as k increases. Meanwhile, the quality of LARS-U increases by a factor of seven as k increases from one to ten.

C. Storage Vs. Locality

Figure 8 depicts the impact of varying $\mathcal{M}$ on both the storage and locality in LARS. We plot LARS-M=0 and LARS-M=1 as constants to delineate the extreme values of $\mathcal{M}$, i.e., $\mathcal{M}$=0 mirrors traditional collaborative filtering, while $\mathcal{M}$=1 forces LARS to employ a complete pyramid. Our metric for locality is *locality loss* (defined in Section III-C1) when compared to a complete pyramid (i.e., $\mathcal{M}$=1). LARS-M=0 requires the lowest storage overhead, but exhibits the highest locality loss, while LARS-M=1 exhibits no locality loss but requires the most storage. For LARS, increasing $\mathcal{M}$ results in increased storage overhead since LARS favors splitting, requiring the maintenance of more pyramid cells each with its own collaborative filtering model. Meanwhile, increasing $\mathcal{M}$ results in smaller locality loss as LARS merges less and maintains more localized cells. The most drastic drop in locality loss is between 0 and 0.3, which is why we chose $\mathcal{M}$=0.3 as a default.

D. Scalability

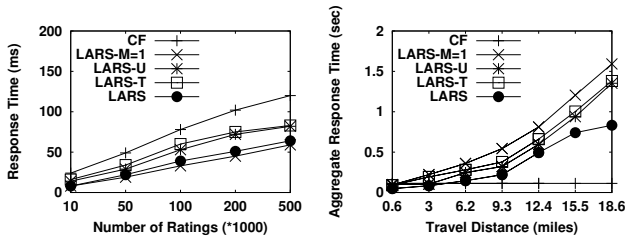
Figure 9 depicts the storage and aggregate maintenance overhead required for an increasing number of ratings. We again plot LARS-M=0 and LARS-M=1 to indicate the extreme cases for LARS. Figure 9(a) depicts the impact of increasing the number of ratings from 10K to 500K on storage overhead. LARS-M=0 requires the lowest amount of storage since it only maintains a single collaborative filtering model. LARS-M=1 requires the highest amount of storage since it requires storage of a collaborative filtering model for all cells (in all levels) of a complete pyramid. The storage requirement of LARS is in between the two extremes since it merges cells to save storage. Figure 9(b) depicts the cumulative computational overhead necessary to maintain the adaptive pyramid initially populated with 100K ratings, then updated with 200K ratings (increments of 50K reported). The trend is similar to the storage experiment, where LARS exhibits better performance than LARS-M=1 due to merging. Though LARS-M=0 has the best performance in terms of maintenance and storage overhead, previous experiments show that it has unacceptable drawbacks in quality/locality.

E. Query Processing Performance

Figure 10 depicts snapshot and continuous query processing performance of LARS, LARS-U (LARS with only *user partitioning*), LARS-T (LARS with only *travel penalty*), CF (traditional collaborative filtering), and LARS-M=1 (LARS with a complete pyramid).

Snapshot queries. Figure 10(a) gives the effect of various number of ratings (10K to 500K) on the average snapshot query performance averaged over 500 queries posed at random locations. LARS and LARS-M=1 consistently outperform all other techniques; LARS-M=1 is slightly better due to recommendations always being produced from the smallest (i.e., most localized) CF models. The performance gap between LARS and LARS-U (and CF and LARS-T) shows that employing the *travel penalty* technique with early termination leads to better query response time. Similarly, the performance gap between LARS and LARS-T shows that employing *user partitioning* technique with its localized (i.e., smaller) collaborative filtering model also benefits query processing.

Continuous queries. Figure 10(b) provides the continuous query processing performance of the LARS variants by reporting the aggregate response time of 500 continuous queries. A continuous query is issued once by a user u to get an initial



(a) Snapshot Queries (b) Continuous Queries

Fig. 10. Query Processing Performance.

answer, then the answer is continuously updated as u moves. We report the aggregate response time when varying the travel distance of u from 1 to 30 miles using a random walk over the spatial area covered by the pyramid. CF has a constant query response time for all travel distances, as it requires no updates since only a single cell is present. However, since CF is unaware of user location change, the consequence is poor recommendation quality (per experiments from Section VI-A). LARS-M=1 exhibits the worse performance, as it maintains all cells on all levels and updates the continuous query whenever the user crosses pyramid cell boundaries. LARS-U has a lower response time than LARS-M=1 due to merging: when a cell is not present on a given influence level, the query is transferred to its next highest ancestor in the pyramid. Since cells higher in the pyramid cover larger spatial regions, query updates occur less often. LARS-T exhibits slightly higher query processing overhead compared to LARS-U: even though LARS-T employs the early termination algorithm, it uses a large (system-wide) collaborative filtering model to (re)generate recommendations once users cross boundaries in the penalty grid. LARS exhibits a better aggregate response time since it employs the early termination algorithm using a localized (i.e., smaller) collaborative filtering model to produce results while also merging cells to reduce update frequency.

VII. Related Work

Location-based services. Current location-based services employ two main methods to provide interesting destinations to users. (1) KNN techniques [19] and variants (e.g., aggregate KNN [21]) simply retrieve the k objects nearest to a user and are completely removed from any notion of user *personalization*. (2) Preference methods such as skylines [22] (and spatial variants [23]) and location-based top-k methods [24] require users to express *explicit* preference constraints. Conversely, *LARS* is the first location-based service to consider *implicit* preferences by using location-based ratings to help users discover new and interesting items.

Recent research has proposed the problem of hyper-local place ranking [25]. Given a user location and query string (e.g., "French restaurant"), hyper-local ranking provides a list of top-k points of interest influenced by previously logged directional queries (e.g., map direction searches from point A to point B). While similar in spirit to LARS, hyper-local ranking is fundamentally different from our work as it does *not personalize* answers to the querying user, i.e., two users issuing the same search term from the same location will receive exactly the same ranked answer set.

Traditional recommenders. A wide array of techniques are capable of producing recommendations using non-spatial ratings for non-spatial items represented as the triple (*user*, *rating*, *item*) (see [6] for a comprehensive survey). We refer to these as "traditional" recommendation techniques. The closest these approaches come to considering location is by incorporating contextual attributes into statistical recommendation models (e.g., weather, traffic to a destination) [26]. However, no traditional approach has studied explicit location-based ratings as done in LARS. Some existing commercial applications make cursory use of location when proposing interesting items to users. For instance, Netflix [2] displays a "local favorites" list containing popular movies for a user's given city. However, these movies are *not* personalized to each user (e.g., using recommendation techniques); rather, this list is built using aggregate rental data for a particular city [27]. LARS, on the other hand, produces personalized recommendations influenced by location-based ratings and a querying user location.

Location-aware recommenders. The CityVoyager system [28] mines a user's personal GPS trajectory data to determine her preferred shopping sites, and provides recommendation based on where the system predicts the user is likely to go in the future. *LARS*, conversely, does not attempt to predict future user movement, as it produces recommendations influenced by user and/or item locations embedded in community ratings.

The spatial activity recommendation system [29] mines GPS trajectory data with embedded user-provided tags in order to detect interesting activities located in a city (e.g., art exhibits and dining near downtown). It uses this data to answer two query types: (a) given an activity type, return where in the city this activity is happening, and (b) given an explicit spatial region, provide the activities available in this region. This is a vastly different problem than we study in this paper. LARS does not mine activities from GPS data for use as suggestions for a given spatial region. Rather, we apply LARS to a more traditional recommendation problem that uses community opinion histories to produce recommendations.

Geo-measured friend-based collaborative filtering [30] produces recommendations by using only ratings that are from a querying user's social-network friends that live in the same city. This technique only addresses user location embedded in ratings. LARS, on the other hand, addresses three possible types of location-based ratings. More importantly, LARS is a complete system (not just a recommendation technique) that employs efficiency and scalability techniques (e.g., merging, splitting, early query termination) necessary for deployment in actual large-scale applications.

VIII. Conclusion

LARS, our proposed location-aware recommender system, tackles a problem untouched by traditional recommender systems by dealing with three types of location-based ratings: *spatial ratings for non-spatial items*, *non-spatial ratings for spatial items*, and *spatial ratings for spatial items*. LARS employs *user partitioning* and *travel penalty* techniques to support spatial ratings and spatial items, respectively. Both tech-

niques can be applied separately or in concert to support the various types of location-based ratings. Experimental analysis using real and synthetic data sets show that LARS is efficient, scalable, and provides better quality recommendations than techniques used in traditional recommender systems.

REFERENCES

[1] G. Linden et al, "Amazon.com Recommendations: Item-to-Item Collaborative Filtering," *IEEE Internet Computing*, vol. 7, no. 1, pp. 76–80, 2003.

[2] "Netflix: http://www.netflix.com."

[3] P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl, "GroupLens: An Open Architecture for Collaborative Filtering of Netnews," in *CSWC*, 1994.

[4] "Foursquare: http://foursquare.com."

[5] "The Facebook Blog, "Facebook Places": http://tinyurl.com/3aetfs3."

[6] G. Adomavicius and A. Tuzhilin, "Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions," *TKDE*, vol. 17, no. 6, pp. 734–749, 2005.

[7] "MovieLens: http://www.movielens.org/."

[8] "New York Times - A Peek Into Netflix Queues: http://www.nytimes.com/interactive/2010/01/10/nyregion/20100110-netflix-map.html."

[9] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, "Item-Based Collaborative Filtering Recommendation Algorithms," in *WWW*, 2001.

[10] J. S. Breese, D. Heckerman, and C. Kadie, "Empirical Analysis of Predictive Algorithms for Collaborative Filtering," in *UAI*, 1998.

[11] W. G. Aref and H. Samet, "Efficient Processing of Window Queries in the Pyramid Data Structure," in *PODS*, 1990.

[12] R. A. Finkel and J. L. Bentley, "Quad trees: A data structure for retrieval on composite keys," *Acta Inf.*, vol. 4, pp. 1–9, 1974.

[13] M. F. Mokbel et al, "SINA: Scalable Incremental Processing of Continuous Queries in Spatiotemporal Databases," in *SIGMOD*, 2008.

[14] J. L. Herlocker, J. A. Konstan, L. G. Terveen, and J. T. Riedl, "Evaluating Collaborative Filtering Recommender Systems," *TOIS*, vol. 22, no. 1, pp. 5–53, 2004.

[15] M. J. Carey et al, "On saying "Enough Already!" in SQL," in *SIGMOD*, 1997.

[16] S. Chaudhuri et al, "Evaluating Top-K Selection Queries," in *VLDB*, 1999.

[17] R. Fagin, A. Lotem, and M. Naor, "Optimal Aggregation Algorithms for Middleware," in *PODS*, 2001.

[18] J. Bao, C.-Y. Chow, M. F. Mokbel, and W.-S. Ku, "Efficient evaluation of k-range nearest neighbor queries in road networks," in *MDM*, 2010.

[19] G. R. Hjaltason and H. Samet, "Distance Browsing in Spatial Databases," *TODS*, vol. 24, no. 2, pp. 265–318, 1999.

[20] K. Mouratidis, M. L. Yiu, D. Papadias, and N. Mamoulis, "Continuous nearest neighbor monitoring in road networks," in *VLDB*, 2006.

[21] D. Papadias, Y. Tao, K. Mouratidis, and C. K. Hui, "Aggregate Nearest Neighbor Queries in Spatial Databases," *TODS*, vol. 30, no. 2, pp. 529–576, 2005.

[22] S. Börzsönyi et al, "The Skyline Operator," in *ICDE*, 2001.

[23] M. Sharifzadeh and C. Shahabi, "The Spatial Skyline Queries," in *VLDB*, 2006.

[24] N. Bruno, L. Gravano, and A. Marian, "Evaluating Top-k Queries over Web-Accessible Databases," in *ICDE*, 2002.

[25] P. Venetis, H. Gonzalez, C. S. Jensen, and A. Y. Halevy, "Hyper-Local, Directions-Based Ranking of Places," *PVLDB*, vol. 4, no. 5, pp. 290–301, 2011.

[26] M.-H. Park et al, "Location-Based Recommendation System Using Bayesian User's Preference Model in Mobile Devices," in *UIC*, 2007.

[27] "Netflix News and Info - Local Favorites: http://tinyurl.com/4qt8ujo."

[28] Y. Takeuchi and M. Sugimoto, "An Outdoor Recommendation System based on User Location History," in *UIC*, 2006.

[29] V. W. Zheng, Y. Zheng, X. Xie, and Q. Yang, "Collaborative Location and Activity Recommendations with GPS History Data," in *WWW*, 2010.

[30] M. Ye, P. Yin, and W.-C. Lee, "Location Recommendation for Location-based Social Networks," in *ACM SIGSPATIAL GIS*, 2010.

APPENDIX

A. Foursquare Description

Foursquare [4] is a mobile location-based social network application. Users are associated with a home city, and alert friends when visiting a venue (e.g., restaurant) by **"checking-in"** on their mobile phones. During a "check-in", users can also leave **"tips"**, which are free text notes describing what that they liked about the venue. Any other user can add the "tip" to her **"to-do list"** if interested in visiting the venue. Once a user visits a venue in the "to-do list" , she marks it as "done". Also, users who check into a venue the most are considered the **"mayor"** of that venue.

B. Experimental Data Details

1) Foursquare Data: We crawled Foursquare and collected data for 1,010,192 users and 642,990 venues across the United States. Foursquare does not publish each "check-in" for a user, however, we were able to collect the following pieces of data: (1) user tips for a venue, (2) the venues for which the user is the mayor, and (3) the completed to-do list items for a user. In addition, we extracted each user's friend list.

Extracting location-based ratings. To extract spatial user ratings for spatial items from the Foursquare data (i.e., the five-tuple (*user*, *ulocation*, *rating*, *item*, *ilocation*)), we map each user visit to a single location-based rating. The *user* and *item* attributes are represented by the unique Foursquare user and venue identifier, respectively. We employ the user's home city in Foursquare as the *ulocation* attribute. Meanwhile, the *ilocation* attribute is the item's inherent location. We use a numeric *rating* value range of [1, 3], translated as follows: (a) 3 represents the user is the "mayor" of the venue, (b) 2 represents that the user left a "tip" at the venue, and (c) 1 represents the user visited the venue as a completed "to-do" list item. Using this scheme, a user may have multiple ratings for a venue, in this case we use the highest rating value.

Data properties. Our experimental data consisted of 22,390 location-based ratings for 4K users for 2K venues all from the state of Minnesota, USA. We used this reduced data set in order to focus our quality experiments on a *dense* rating sample. Use of *dense* ratings data has been shown to be a very important factor when testing and comparing recommendation quality [14], since use of *sparse* data (i.e., having users or items with very few ratings) tends to cause inaccuracies in recommendation techniques.

2) MovieLens Data: The MovieLens data used in our experiments was real movie rating data taken from the popular MovieLens recommendation system at the University of Minnesota [7]. This data consisted of 87,025 ratings for 1,668 movies from 814 users. Each rating was associated with the zip code of the user who rated the movie, thus giving us a real data set of spatial user ratings for non-spatial items.

3) Synthetic Data: The synthetic data set we use in our experiments was generated to contain 2000 users and 1000 items, and 500,000 ratings. Users and items locations are randomly generated over the state of Minnesota, USA. Users' ratings to items are assigned random values between zero and five.