LARS*: An Efficient and Scalable Location-Aware Recommender System

Mohamed Sarwat*, Justin J. Levandoski†, Ahmed Eldawy* and Mohamed F. Mokbel*
*Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455
†Microsoft Research, Redmond, WA 98052-6399

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.

Index Terms—Recommender System, Spatial, Location, Performance, Efficiency, Scalability, Social.

1 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 [1]). The technique used by many of these systems is collaborative filtering (CF) [2], 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 [2] and Facebook Places [3]) 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 [4] assume ratings are represented by the (*user*, *rating*, *item*) triple, thus are ill-equipped to produce location-aware recommendations.

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.

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.

1.1 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 [5] containing approximately 87K movie ratings associated with user zip codes (i.e., spatial ratings for non-spatial items) and (2) data from the Foursquare [6] location-based social network containing user visit data for 1M

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

1. Netflix: http://www.netflix.com
2. Foursquare: http://foursquare.com

users to 643K venues across the United States (i.e., spatial ratings for spatial items). 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 [7] 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).

1.2 Our Contribution: LARS* - A Location-Aware Recommender System

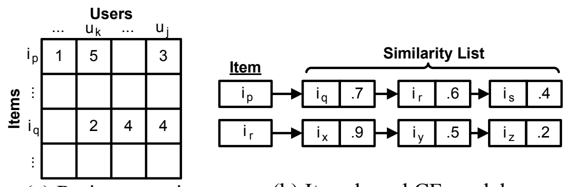
Like traditional recommender systems, LARS* suggests k items personalized for a querying user u. However, LARS* is distinct in its ability to produce location-aware recommendations using *each* of the three types of location-based rating within a *single* framework.

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 [6] and MovieLens [5], 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.
- LARS* distinguishes itself from LARS [8] in the following points: (1) LARS* achieves higher locality gain than LARS using a better user partitioning data structure and



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

Fig. 2. Item-based CF model generation.

algorithm. (2) LARS* exhibits a more flexible tradeoff between locality and scalability. (3) LARS* provides a more efficient way to maintain the user partitioning structure, as opposed to LARS expensive operations.

- 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 2 gives an overview of LARS*. Sections 4, 5, and 6 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 7 provides experimental analysis. Section 8 covers related work, while Section 9 concludes the paper.

2 LARS* Overview

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

2.1 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 4 to 6.

2.2 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), or unary (e.g., Facebook "check-ins" [3]). 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

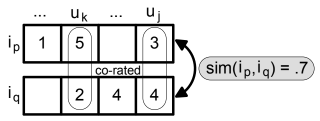


Fig. 3. Item-based similarity calculation.

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 [1], 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]:

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

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 $P_{(u,i)}$.

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:

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

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 [4], that uses correlations between users (instead of items).

3 Non-Spatial User Ratings for Non-Spatial Items

The traditional item-based collaborative filtering (CF) method is a special case of LARS*. CF takes as input the classical rating triplet (*user, rating, item*) such that neither the user location nor the item location are specified. In such case, LARS* directly employs the traditional model building phase (Phase-I in section 2) to calculate the similarity scores between

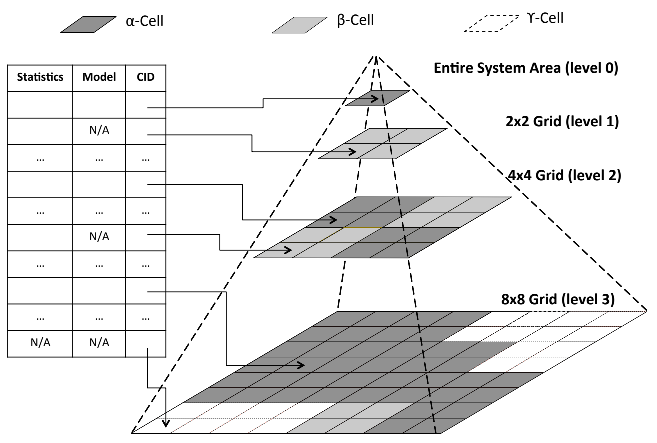


Fig. 4. Pyramid data structure

all items. Moreover, recommendations are produced to the users using the recommendation generation phase (Phase-II in section 2). During the rest of the paper, we explain how LARS* incorporates either the user spatial location or the item spatial location to serve location-aware recommendations to the system users.

4 Spatial User Ratings for Non-Spatial Items

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 1.1). 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 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 4.1, query processing in Section 4.2, and finally data structure maintenance in Section 4.3.

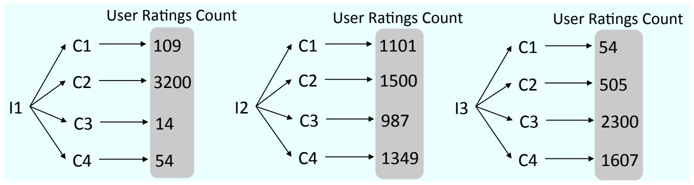


Fig. 5. Example of *Items Ratings Statistics Table*

4.1 Data Structure

LARS* employs a partial *in-memory* 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.

A rating may belong to up to H pyramid cells: one per each pyramid level starting from the lowest maintained grid cell containing the embedded user location up to the root level. To provide a tradeoff between recommendation locality and system scalability, the pyramid data structure maintains three types of cells (see figure 4): (1) Recommendation Model Cell (α-Cell), (2) Statistics Cell (β-Cell), and (3) Empty Cell (γ-Cell), explained as follows:

Recommendation Model Cell (α-Cell). Each α-Cell stores an item-based collaborative filtering model built using *only* the spatial ratings with user locations contained in the cell's spatial region. Note that the root cell (level 0) of the pyramid is an α-Cell and represents a "traditional" (i.e., non-spatial) item-based collaborative filtering model. Moreover, each α-Cell maintains statistics about all the ratings located within the spatial extents of the cell. Each α-Cell C_p maintains a hash table that indexes all items (by their IDs) that have been rated in this cell, named *Items Ratings Statistics Table*. For each indexed item i in the *Items Ratings Statistics Table*, we maintain four parameters; each parameter represent the *number of user ratings* to item i in each of the four children cells (i.e., C_1, C_2, C_3, and C_4) of cell C_p. An example of the maintained parameters is given in Figure 5. Assume that cell C_p contains ratings for three items I_1, I_2, and I_3. Figure 5 shows the maintained statistics for each item in cell C_p. For example, for item I_1, the number of user ratings located in child cell C_1, C_2, C_3, and C_4 is equal to 109, 3200, 14, and 54, respectively. Similarly, the number of user ratings is calculated for items I_2 and I_3.

Statistics Cell (β-Cell). Like an α-Cell, a β-Cell maintains statistics (i.e., *items ratings Statistics Table*) about the user/item ratings that are located within the spatial range of the cell. The only difference between an α-Cell and a β-Cell is that a β-Cell does not maintain a collaborative filtering (CF) model for the user/item ratings lying in its boundaries. In consequence, a β-Cell is a light weight cell such that it incurs less storage than an α-Cell. In favor of system scalability, LARS* prefers a β-Cell over an α-Cell to reduce the total system storage.

Empty Cell (γ-Cell). a γ-Cell is a cell that maintains neither the statistics nor the recommendation model for the ratings lying within its boundaries. a γ-Cell is the most light weight cell among all cell types as it almost incurs no storage overhead. Note that an α-Cell can have α-Cells, β-Cells, or γ-Cells children. Also, a β-Cell can have α-Cells, β-Cells, or γ-Cells children. However, a γ-Cell cannot have any children.

4.1.1 Pyramid structure intuition

An α-Cell requires the highest storage and maintenance overhead because it maintains a CF model as well as the user/item ratings statistics. On the other hand, an α-Cell (as opposed to β-Cell and γ-Cell) is the only cell that can be leveraged to answer recommendation queries. A pyramid structure that only contains α-Cells achieves the highest recommendation locality, and this is why an α-Cell is considered the highly ranked cell type in LARS*. a β-Cell is the secondly ranked cell type as it only maintains statistics about the user/item ratings. The storage and maintenance overhead incurred by a β-Cell is less expensive than an α-Cell. The statistics maintained at a β-Cell determines whether the children of that cell needs to be maintained as α-Cells to serve more localized recommendation. Finally, a γ-Cell (lowest ranked cell type) has the least maintenance cost, as neither a CF model nor statistics are maintained for that cell. Moreover, a γ-Cell is a leaf cell in the pyramid.

LARS* upgrades (downgrades) a cell to a higher (lower) cell rank, based on trade-offs between recommendation locality and system scalability (discussed in Section 4.3). If recommendation locality is preferred over scalability, more α-Cells are maintained in the pyramid. On the other hand, if scalability is favored over locality, more γ-Cells exist in the pyramid. β-Cells comes as an intermediary stage between α-Cells and γ-Cells to further increase the recommendation locality whereas the system scalability is not quite affected.

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 than a "space-partitioning" structure for two main reasons: (1) "data-partitioning" structures index data points, and hence covers only locations that are inserted in them. In other words, "data-partitioning" structures are not guaranteed to completely cover a given space, which is not suitable for queries issued in arbitrary spatial locations. (2) In contrast to "data-partitioning" structures (e.g., R-trees [13]), "space partitioning" structures show better performance for dynamic memory resident data [14], [15], [16].

4.1.2 LARS versus LARS*

Table 1 compares LARS* against LARS. Like LARS*, LARS [8] employs a partial pyramid data structure to support spatial user ratings for non-spatial items. LARS is different from LARS* in the following aspects: (1) As shown in Table 1, LARS* maintains α-Cells, β-Cells, and γ-Cells, whereas LARS only maintains α-Cells and γ-Cells. In other words, LARS either merges or splits a pyramid cell based on a tradeoff between scalability and recommendation locality. LARS* employs the same tradeoff and further increases the recommendation locality by allowing for more α-Cells to be maintained at lower pyramid levels. (2) As opposed to LARS, LARS* does not perform a speculative splitting operation to decide whether to maintain more localized CF models. However, LARS maintains extra statistics at each α-Cell and β-Cell that helps in quickly deciding wether a CF model needs to be maintained at a child cell. (3) As it turns out from Table 1, LARS* achieves higher recommendation locality than LARS. That is due to the fact that LARS maintains a CF recommendation model in a cell at pyramid level h if and only if a CF model, at its parent cell at level $h-1$, is also maintained. However, LARS* may maintain an α-Cell at level h even though its parent cell, at level $h-1$, does not maintain a CF model, i.e., the parent cell is a β-Cell. In LARS*, the role of a β-Cell is to keep the *user/item ratings statistics* that are used to quickly decide whether the child cells needs to be γ-Cells or α-Cells. (4) As given in Table 1, LARS* incurs more storage overhead than LARS which is explained by the fact that LARS* maintains additional type of cell, i.e., β-Cells, whereas LARS only maintains α-Cells and γ-Cells. In addition, LARS* may also maintain more α-Cells than LARS does in order to increase the recommendation locality. (5) Even LARS* may maintain more α-Cells than LARS besides the extra statistics maintained at β-Cells, nonetheless LARS* incurs less maintenance cost. That is due to the fact that LARS* also reduces the maintenance overhead by avoiding the expensive speculative splitting operation employed by LARS maintenance algorithm. Instead, LARS* employs the *user/item ratings statistics* maintained at either a β-Cell or an α-Cell to quickly decide whether the cell children need to maintain a CF model (i.e., upgraded to α-Cells), just needs to maintain the statistics (i.e., become β-Cells), or perhaps downgraded to γ-Cells.

	LARS	**LARS***
	Supported Features	
α-Cell	Yes	Yes
β-Cell	No	Yes
γ-Cell	Yes	Yes
Speculative Split	Yes	No
Rating Statistics	No	Yes
	Performance Factors	
Locality	-	≈26% higher than LARS
Storage	≈5% lower than LARS*	-
Maintenance	-	≈38% lower than LARS

TABLE 1
Comparison between LARS and LARS*. Detailed experimental evaluation results are provided in section 7.

4.2 Query Processing

Given a recommendation query (as described in Section 2.1) 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 α-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 an α-Cell is not maintained at the lowest level, we return the nearest maintained ancestor α-Cell. (2) The top-k recommended items are generated using the item-based collaborative filtering technique (covered in Section 2.2) 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 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 only if the new cell is an α-Cell. In case the new cell is an α-Cell, LARS* only sends incremental updates [16] 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 α-Cell. Note that since higher-level cells maintain larger spatial regions, the continuous query will cross spatial boundaries less often, reducing the amount of 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 α-Cell containing the querying user location at level I, instead of the lowest maintained grid α-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.

4.3 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 α-Cells and contain ratings statistics and 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 *cell type maintenance* step that scans all cells starting from the lowest level h and downgrades cell types to either (β-Cell or γ-Cell) if necessary (cell type switching is discussed in Section 4.5.2). We note that while the original partial pyramid [11] was concerned with spatial queries over static data, it did not address pyramid maintenance.

4.4 Main Idea

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 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 [17]. Thus, maintenance is needed less often.

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.

Algorithm 1 Pyramid maintenance algorithm

```
1: /* Called after cell $C$ receives $N$% new ratings */
2: Function PyramidMaintenance(Cell $C$, Level $h$)
3: /* Step I: Statistics Maintenance*/
4: Maintain cell $C$ statistics
5: /*Step II: Model Rebuild */
6: if (Cell $C$ is an $\alpha$-Cell) then
7:     Rebuild item-based collaborative filtering model for cell $C$
8: end if
9: /*Step III: Cell Child Quadrant Maintenance */
10: if ($C$ children quadrant $q$ cells are $\alpha$-Cells) then
11:     CheckDownGradeToSCells($q$,$C$) /* covered in Section 4.5.2 */
12: else if ($C$ children quadrant $q$ cells are $\gamma$-Cells) then
13:     CheckUpGradeToSCells($q$,$C$)
14: else
15:     isSwitchedToMcells ← CheckUpGradeToMCells($q$,$C$) /* covered in Section 4.5.3 */
16:     if (isSwitchedToMcells is False) then
17:         CheckDownGradeToECells($q$,$C$)
18:     end if
19: end if
20: return
```

4.5 Maintenance Algorithm

Algorithm 1 provides the pseudocode for the LARS* maintenance algorithm. The algorithm takes as input a pyramid cell C and level h, and includes three main steps: *Statistics Maintenance*, *Model Rebuild* and *Cell Child Quadrant Maintenance*, explained below.

Step I: Statistics Maintenance. The first step (line 4) is to maintain the *Items Ratings Statistics Table*. The maintained statistics are necessary for cell type switching decision, especially when new location-based ratings enter the system. As the *items ratings statistics table* is implemented using a hash table, then it can be queried and maintained in $O(1))$ time, requiring $O(|I_C|)$ space such that I_C is the set of all items rated at cell C and $|I_C|$ is the total number of items in I_C.

Step II: Model Rebuild. The second step is to rebuild the item-based collaborative filtering (CF) model for a cell C, as described in Section 2.2 (line 7). The model is rebuilt at cell C only if cell C is an α-Cell, otherwise (β-Cell or γ-Cell) no CF recommendation model is maintained, and hence the model rebuild step does not apply 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 item-based CF model is $O(\frac{R^2}{U})$ (per Section 2.2), 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 III: Cell Child Quadrant Maintenance. LARS* invokes a maintenance step that may decide whether cell C child quadrant need to be switched to a different cell type based on trade-offs between *scalability* and *locality*. The algorithm first checks if cell C child quadrant q at level $h+1$ is of type α-Cell (line 10). If that case holds, LARS* considers quadrant q cells as candidates to be downgraded to β-Cells (calling function *CheckDownGradeToSCells* on line 11). We provide details of the *Downgrade α-Cells to β-Cells* operation in Section 4.5.2. On the other hand, if C have a child quadrant of type γ-Cells at level $h+1$ (line 12), LARS* considers upgrading cell C four children cells at level $h+1$ to β-Cells (calling function *CheckUpGradeToSCells* on line 13). The *Updgrade From E to β-Cells* operation is covered in Section 4.5.4. However, if C have a child quadrant of type β-Cells at level $h+1$ (line 12), LARS* first considers upgrading cell C four children cells at level $h+1$ from β-Cells to α-Cells (calling function *CheckUpGradeToMCells* on line 15). If the children cells are not switched to α-Cells, LARS* then considers downgrading them to γ-Cells (calling function *CheckDownGradeToECells* on line 17). Cell Type switching operations 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.

4.5.1 Recommendation Locality

In this section, we explain the notion of locality in recommendation that is essential to understand the cell type switching (upgrade/downgrade) operations highlighted in the Pyramid-Maintenance algorithm (algorithm 1). We use the following example to give the intuition behind recommendation locality.

Running Example. Figure 6 depicts a two-levels pyramid in which C_p is the root cell and its children cells are C_1, C_2, C_3, and C_4. In the example, we assume eight users (U_1, U_2, ..., and U_8) have rated eight different items (I_1, I_2, ..., and I_8). Figure 6(b) gives the spatial distributions of users U_1, U_2, U_3, U_4, U_5, U_6, U_7, and U_8 as well as the items that each user rated.

Intuition. Consider the example given in Figure 6. In cell C_p, users U_2 and U_5 that belongs to the child cell C_2 have both rated items I_2 and I_5. In that case, the similarity score between items I_2 and I_5 in the item-based collaborative filtering CF model built at cell C_2 is exactly the same as the one in the CF model built at cell C_p. The last phenomenon happened because items (i.e., I_2 and I_5) have been rated by mostly users located in the same child cell, and hence the recommendation model at the parent cell will not be different from the model at the children cells. In this case, if the CF model at C_2 is not maintained, LARS* does not lose recommendation locality at all.

The opposite case happens when an item is rated by users located in different pyramid cells (spatially skewed). For example, item I_4 is rated by users U_2, U_4, and U_7 in three

Parameter	Description
$RP_{c,i}$	The set of user pairs that co-rated item i in cell c
$RS_{c,i}$	The set of user pairs that co-rated item i in cell c such that each pair of users $\langle u_1, u_2\rangle \in S_{c,i}$ are not located in the same child cell of c
$LG_{c,i}$	The degree of locality lost for item i from downgrading the four children of cell c to β-Cells, such that $0 \leq LG_{c,i} \leq 1$
LG_c	The amount of locality lost by downgrading cell c four children cells to β-Cells ($0 \leq LG_c \leq 1$)

TABLE 2
Summary of Mathematical Notations.

different cells (C_2, C_3, and C_4). In this case, U_2, U_4, and U_7 are spatially skewed. Hence, the similarity score between item I_4 and other items at the children cells is different from the similarity score calculated at the parent cell C_p because not all users that have rated item I_4 exist in the same child cell. Based on that, we observe the following:

Observation 1: The more the user/item ratings in a parent cell C are geographically skewed, the higher the locality gained from building the item-based CF model at the four children cells.

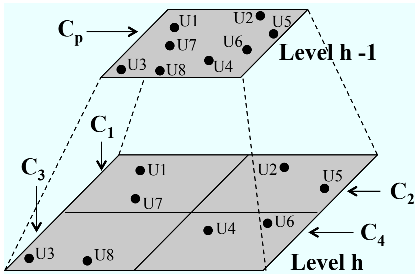
The amount of locality gained/lost by maintaining the child cells of a given pyramid cell depends on whether the CF models at the child cells are similar to the CF model built at the parent cell. In other words, LARS* loses locality if the child cells are not maintained even though the CF model at these cells produce different recommendations than the CF model at the parent cell. LARS* leverages Observation 1 to determine the amount of locality gained/lost due to maintaining an item-based CF model at the four children. LARS* calculates the locality loss/gain as follows:

Locality Loss/Gain. Table 2 gives the main mathematical notions used in calculating the recommendation locality loss/gain. First, the *Item Ratings Pairs Set* ($RP_{c,i}$) is defined as the set of all possible pairs of users that rated item i in cell c. For example, in figure 6(c) the item ratings pairs set for item I_7 in cell C_p (RP_{C_p,I_7}) has three elements (i.e., RP_{C_p,I_7}=$\{\langle U_3,U_6\rangle$,$\langle U_3,U_7\rangle$,$\langle U_6,U_7\rangle\}$) as only users U_1 and U_7 have rated item I_1. Similarly, RP_{C_p,I_2} is equal to $\{\langle U_6,U_7\rangle\}$ (i.e., Users U_2 and U_5 have rated item I_2).

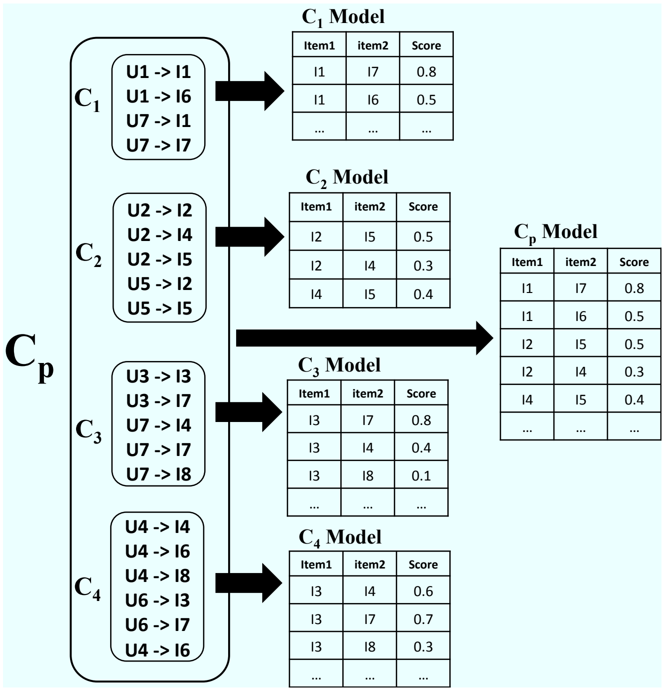
For each item, we define the *Skewed Item Ratings Set ($RS_{c,i}$)* as the total number of user pairs in cell c that rated item i such that each pair of users $\in RS_{c,i}$ do not exist in the same child cell of c. For example, in Figure 6(c), the skewed item ratings set for item I_2 in cell C_p (RS_{C_P,I_2}) is $\emptyset$ as all users that rated I_2, i.e., U_2 and U_5 are collocated in the same child cell C_2. For I_4, the skewed item ratings set RS_{C_P,I_2}=$\{\langle U_2,U_7\rangle, \langle U_2,U_4\rangle, \langle U_4,U_7\rangle\}$ as all users that rated item I_2 are located in different child cells,i.e., U_2 at C_2, U_4 at C_4, and U_7 at C_3.

Given the aforementioned parameters, we calculate *Item Locality Loss ($LG_{c,i}$)* for each item, as follows:

Definition 1: **Item Locality Loss ($LG_{c,i}$)**
$LG_{c,i}$ is defined as the degree of locality lost for item i from downgrading the four children of cell c to β-Cells, such that $0 \leq LG_{c,i} \leq 1$.



(a) Two-Levels Pyramid



(b) Ratings Distribution and Recommendation Models

Item	$\|RP_{c,i}\|$	$\|RS_{c,i}\|$	$LG_{c,i}$
I1	1	0	0.0
I2	1	0	0.0
I3	1	1	1.0
I4	3	3	1.0
I5	1	0	0.0
I6	3	2	0.666
I7	6	2	0.166
I8	1	1	1.0
Cell Locality Gain ($LG_{c,i}$)			0.48

(c) Locality Loss/Gain at C_p

Fig. 6. Item Ratings Spatial Distribution Example

$$LG_{c,i} = \frac{|RS_{c,i}|}{|RP_{c,i}|} \qquad (3)$$

The value of both $|RS_{c,i}|$ and $|RP_{c,i}|$ can be easily extracted using the *items ratings statistics table*. Then, we use the $LG_{c,i}$ values calculated for all items in cell c in order to calculate the overall *Cell Locality loss (LG_c)* from downgrading the children cells of c to α-Cells.

Definition 2: **Locality Loss (LG_c)**
LG_c is defined as the total locality lost by downgrading cell c four children cells to β-Cells ($0 \leq LG_c \leq 1$). It is calculated as the the sum of all items locality loss normalized by the total number of items $|I_c|$ in cell c.

$$LG_c = \frac{\sum_{i \in I_c} LG_{c,i}}{|I_c|} \qquad (4)$$

The cell locality loss (or gain) is harnessed by LARS* to determine whether the cell children need to be downgraded from α-Cell to β-Cell rank, upgraded from the γ-Cell to β-Cell rank, or downgraded from β-Cell to γ-Cell rank. During the rest of section 4, we explain the cell rank upgrade/downgrade operations.

4.5.2 Downgrade α-Cells to β-Cells

That operation entails downgrading an entire quadrant of cells from M-Cells to β-Cells at level h with a common parent at level $h-1$. Downgrading α-Cells to β-Cells 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 the four children cells. Furthermore, downgrading α-Cells to β-Cells leads to the following performance improvements: (a) *less maintenance cost*, since less CF models are periodically rebuilt, and (b) *less continuous query processing computation*, as β-Cells does not maintain a CF model and if many β-Cells cover a large spatial region, hence, for users crossing β-Cells boundaries, we do not need to update the recommendation query answer. Downgrading children cells from α-Cells to β-Cells might hurt recommendation locality, since no CF models are maintained at the granularity of the child cells anymore.

At cell C_p, in order to determine whether to downgrade a quadrant q cells to β-Cells (i.e., function *CheckDownGradeToSCells* on line 11 in Algorithm 1), we calculate two percentage values: (1) *locality_loss* (see equation 4), the amount of locality lost by (potentially) downgrading the children cells to β-Cells, and (2) *scalability_gain*, the amount of scalability gained by (potentially) downgrading the children cells to β-Cells. Details of calculating these percentages are covered next. When deciding to downgrade cells to β-Cells, 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* downgrades a quadrant q cells to β-Cells (i.e., discards quadrant q) if:

$$(1-\mathcal{M}) * scalability\_gain > \mathcal{M} * locality\_loss \qquad (5)$$

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 allow for β-Cells downgrade. At the extremes, setting $\mathcal{M}$=0 (i.e., always switch to β-Cell) implies LARS* will function as a traditional CF recommender system, while setting $\mathcal{M}$=1 causes LARS* pyramid cells to all be α-Cells, i.e., LARS* will employ a complete pyramid structure maintaining a recommendation model at all cells at all levels.

Calculating Locality Loss. To calculate the locality loss at a cell C_p, LARS* leverages the *Item Ratings Statistics Table* maintained in that cell. First, LARS* calculates the item locality loss $LG_{C_p,i}$ for each item i in the cell C_p. Therefore, LARS* aggregates the item locality loss values calculated for each item $i \in C_p$, to finally deduce the global cell locality loss LG_{C_p}.

Calculating scalability gain. Scalability gain is measured in storage and computation savings. We measure scalability gain by summing the recommendation model sizes for each of the downgraded (i.e., child) cells (abbr. $size_m$), and divide this value by the sum of $size_m$ and the recommendation model 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. using the *Items Ratings Statistics Table* maintained at cell C_p, the locality loss at cell C_p can be calculated in $O(|I_{C_p}|)$ time such that $|I_{C_p}|$ represents the total number of items in C_p. As scalability gain can be calculated in $O(1)$ time, then the total time cost of the *Downgrade To β-Cells* operation is $O(|I_{C_p}|)$.

Example. For the example given in Figure 6(c), the locality loss of downgrading cell C_p four children cells $\{C_1, C_2, C_3, C_4\}$ to β-Cells is calculated as follows: First, we retrieve the locality loss $LG_{C_p,i}$ for each item $i \in \{I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8\}$, from the maintained statistics at cell C_p. As given in figure 6(c), LG_{C_p,I_1}, LG_{C_p,I_2}, LG_{C_p,I_3}, LG_{C_p,I_4}, LG_{C_p,I_5}, LG_{C_p,I_6}, LG_{C_p,I_7}, and LG_{C_p,I_8} are equal to 0.0, 0.0, 1.0, 1.0, 0.0, 0.666, 0.166, and 1.0, respectively. Then, we calculate the overall locality loss at C_p (using equation 4), LG_{C_p} by summing all the locality loss values of all items and dividing the sum by the total number of items. Hence, the scalability loss is equal to $(\frac{0.0+0.0+1.0+1.0+0.0+1.0+0.666+1.0}{8})$ = 0.48 = 48%. 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 $(0.3 \times 50) < (0.7 \times 48)$, meaning that LARS* will not downgrade cells C_1, C_2, C_3, C_4 to β-Cells.

4.5.3 Upgrade β-Cells to α-Cells

Upgrading β-Cells to α-Cells operation entails upgrading the cell type of a cell child quadrant at pyramid level h under a cell at level $h-1$, to α-Cells. *Upgrading β-Cells to α-Cells* operation improves locality in LARS*, as it leads to maintaining a CF model at the children cells that represent more granular spatial regions capable of producing recommendations unique to the smaller, more "local", spatial regions. On the other hand, upgrading cells to α-Cells hurts scalability by requiring storage and maintenance of more item-based collaborative filtering models. The upgrade to α-Cells operation also negatively affects continuous query processing, since it creates more granular α-Cells causing user locations to cross α-Cell boundaries more often, triggering recommendation updates.

To determine whether to upgrade a cell C_P (quadrant q) four children cells to α-Cells (i.e., function *CheckUpGradeToMCells* on line 15 of Algorithm 1). Two percentages are calculated: *locality_gain* and *scalability_loss*. These values are the opposite of those calculated for the *Upgrade to β-Cells* operation. LARS* change cell C_P child quadrant q to α-Cells only if the following condition holds:

$$\mathcal{M} * locality\_gain > (1-\mathcal{M}) * scalability\_loss \qquad (6)$$

This equation represents the opposite criteria of that presented for *Upgrade to β-Cells* operation in Equation 5.

Calculating locality gain. To calculate the locality gain, LARS* does not need to speculatively build the CF model at the four children cells. The locality gain is calculated the same way the locality loss is calculated in equation 4.

Calculating scalability loss. We calculate *scalability loss* by estimating the storage necessary to maintain the children cells. Recall from Section 2.2 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 *Upgradeded to α-Cell* 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. Similar to the *CheckDownGradeToSCells* operation, scalability loss is calculate in $O(1)$ and locality gain can be calculated in $O(|I_{C_p}|)$ time. Then, the total time cost of the *CheckUpGradeToMCells* operation is $O(|I_{C_p}|)$.

Example. Consider the example given in Figure 6(c). Assume the cell C_p is an α-Cell and its four children C_1, C_2, C_3, and C_4 are β-Cells. The *locality gain* (LG_{C_p}) is calculated using equation 4 to be 0.48 (i.e., 48%) as depicted in the table in Figure 6(c). Further, assume that we estimate the extra storage overhead for upgradinging the children cells to α-Cells (i.e., *storage loss*) to be 50%. Assuming $\mathcal{M}$=0.7, then $(0.7 \times 48) > (0.3 \times 50)$, meaning that LARS* will decide to upgrade C_P four children cells to α-Cells as *locality gain* is significantly higher than *scalability loss*.

4.5.4 Downgrade β-Cells to γ-Cells and Vice Versa

Downgrading β-Cells to γ-Cells operation entails downgrading the cell type of a cell child quadrant at pyramid level h under a cell at level $h-1$, to γ-Cells (i.e., empty cells). Downgrading the child quadrant type to γ-Cells means that the maintained statistics are no more maintained in the children cell, which definitely reduces the overhead of maintaining the *Item Ratings Statistics Table* at these cells. Even though γ-Cells incurs no maintenance overhead, however they reduce the amount of recommendation locality that LARS* provides.

The decision of downgrading from β-Cells to γ-Cells is taken based on a system parameter, named *MAX_SLEVELS*. It is defined as the maximum number of consecutive pyramid levels in which descendant cells can be β-Cells. *MAX_SLEVELS* can take any value between zero and the total height of the pyramid. A high value of *MAX_SLEVELS* results in maintaining more β-Cells and less γ-Cells in the pyramid. For example, in Figure 4, *MAX_SLEVELS* is set to two, and this is why if two consecutive pyramid levels are β-Cells, the third level β-Cells are autotmatically downgraded to γ-Cells. For each β-Cell C, a counter, called *S-Levels Counter*, is maintained. The S-Levels Counter stores of the total number of consecutive levels in the direct ancestry of cell C such that all these levels contains β-Cells.

At a β-Cell C, if the cell children are β-Cells, then we compare the *S-Levels Counter* at the child cells with the *MAX_SLEVELS* parameter. Note that the counter counts only the consecutive S-Levels, so if some levels in the chain are α-Cells the counter is reset to zero at the α-Cells levels. If *S-Levels Counter* is greater than or equal to *MAX_SLEVELS*, then the children cells of C are downgraded to γ-Cells. Otherwise, cell C children cells are not downgraded to γ-Cells. Similarly, LARS* also makes use of the same *S-Levels Counter* to decide whether to upgrade γ-Cells to β-Cells.

5 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 1.1). Traditional (non-spatial) recommendation techniques may produce 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 5.1 describes the query processing framework while Section 5.2 describes travel distance computation.

5.1 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:

$$RecScore(u, i) = P(u, i) - TravelPenalty(u, i) \qquad (7)$$

$P(u, i)$ is the standard item-based CF predicted rating of item i for user u (see Section 2.2). $TravelPenalty(u, i)$ 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 7 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 [18], [19], [20]. 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 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:  $R \leftarrow \phi$
4:  for ($K$ iterations) do
5:      $i \leftarrow$ Retrieve the item with the next lowest travel penalty (Section 5.2)
6:      Insert $i$ into $R$ ordered by $RecScore(U, i)$ computed by Equation 7
7:  end for
8:  $LowestRecScore \leftarrow RecScore$ 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:     $i \leftarrow$ Retrieve the next item in order of penalty score (Section 5.2)
12:     $MaxPossibleScore \leftarrow MAX\_RATING$ - $i.penalty$
13:     if $MaxPossibleScore \leq LowestRecScore$ then
14:         return $R$ /* early termination - end query processing */
15:     end if
16:     $RecScore(U, i) \leftarrow P(U, i)$ - $i.penalty$ /* Equation 7 */
17:     if $RecScore(U, i) > LowestRecScore$ then
18:         Insert $i$ into $R$ ordered by $RecScore(U, i)$
19:         $LowestRecScore \leftarrow RecScore$ of the $k^{th}$ object in $R$
20:     end if
21: end while
22: return $R$
```

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 7. This initial part is concluded by setting the lowest recommendation score value (*LowestRecScore*) as the *RecScore* of the k^{th} 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 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 7, insert i into R sorted by its score (removing the k^{th} 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 4.3, we rebuild the model after receiving $N\%$ new ratings.

5.2 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 trade-off 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.

5.2.1 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 [21], [22], [23]. 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 7. Incremental KNN techniques exist for both Euclidean distance [22] and (road) network distance [21], [23]. 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 [22]: $O(k + logN)$, where N and k are the number of total items and items retrieved so far, respectively.

5.2.2 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 7. 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.

6 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 4 and 5; 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 5.

7 EXPERIMENTS

This section provides experimental evaluation of LARS* based on an actual system implementation using C++ and STL. We compare LARS* with the standard item-based collaborative filtering technique along with several variations of LARS*. We also compare LARS* to LARS [8]. Experiments are based on three data sets:

Foursquare: a real data set consisting of *spatial user ratings for spatial items* derived from Foursquare user histories. 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 [17], since use of *sparse* data (i.e., having users or items with very few ratings) tends to cause inaccuracies in recommendation techniques.

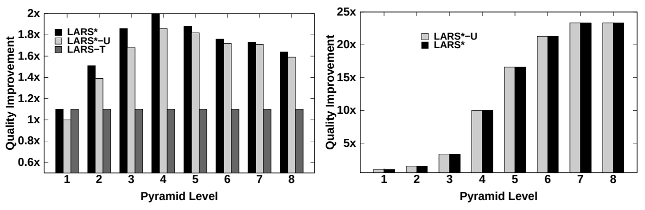
MovieLens: a real data set consisting of *spatial user ratings for non-spatial items* taken from the popular MovieLens recommender system [5]. The Foursquare and MovieLens data are used to test recommendation quality. The MovieLens data used in our experiments was real movie rating data taken from the popular MovieLens recommendation system at the University of Minnesota [5]. 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.

Synthetic: a synthetically generated data set consisting of spatial user ratings for spatial items for venues in the state of Minnesota, USA. The synthetic data set we use in our experiments is 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. As this data set contains a number of ratings that is about twenty five times and five times larger than the foursquare data set and the Movilens data set, we use such synthetic data set to test scalability and query efficiency.

Unless mentioned otherwise, the default value of $\mathcal{M}$ is 0.3, k is 10, the number of pyramid levels is 8, the influence level is the lowest pyramid level, and MAX_SLEVELS is set to two. The rest of this section evaluates LARS* recommendation quality (Section 7.1), trade-offs between storage and locality (Section 7.4), scalability (Section 7.5), and query processing efficiency (Section 7.6). As the system stores its data structures in main memory, all reported time measurements represent the CPU time.

7.1 Recommendation Quality for Varying Pyramid Levels

These experiments test the recommendation quality improvement that LARS* achieves over the standard (non-spatial) item-based collaborative filtering method using both the Foursquare and MovieLens data. To test the effectiveness of our proposed techniques, we test the quality improvement of LARS* with only travel penalty enabled (abbr. LARS*-T), LARS* with only user partitioning enabled and M set to one (abbr. LARS*-U), and LARS* with both techniques enabled and M set to one (abbr. LARS*). Notice that LARS*-T represents the traditional item-based collaborative filtering augmented with the travel penalty technique (section 5) to take the distance between the querying user and the recommended items into account. We do not plot LARS with LARS* as both give the same result for M=1, and the quality experiments are meant to show how locality increases the recommendation quality.



(a) Foursquare data (b) MovieLens data

Fig. 7. Quality experiments for varying locality

Quality Metric. 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 ranked recommendations $\mathcal{S}$ by submitting the *user* and *ulocation* associated with t. We first calculate the quality as the weighted sum of the number of occurrences of the *item* associated with t (the higher the better) in $\mathcal{S}$. The weight of an item is a value between zero and one that determines how close the rank of this item from its real rank. The quality of each recommendation method is calculated and compared against the baseline, i.e., traditional item-based collaborative filtering. We finally report the ratio of improvement in quality each recommendation method achieves over the baseline. 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 correctly ranked answers that contain venues (or movies) a user is known to like is considered of higher quality.

Figure 7(a) compares the quality improvement of each technique (over traditional collaborative filtering) for varying locality (i.e., different levels of the adaptive pyramid) using the Foursquare data. LARS*-T does not use the adaptive pyramid, thus has constant quality improvement. However, LARS*-T shows some quality improvement over traditional collaborative filtering. This quality boost is due to that fact that LARS*-T uses a *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 7.6.

Figure 7(b) compares the quality improvement of LARS*-U over CF (traditional collaborative filtering) for varying locality

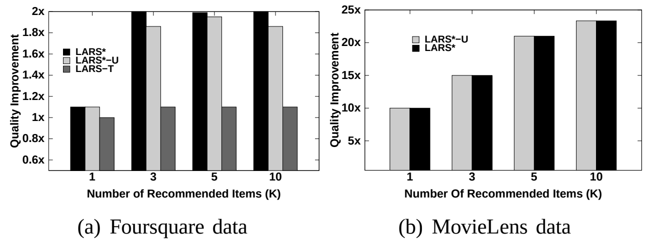


Fig. 8. Quality experiments for varying answer sizes

using the MovieLens data. Notice that LARS* gives the same quality improvement as LARS*-U because LARS*-T do not apply for this dataset since movies are not spatial. Compared to CF, the quality improvement achieved by LARS*-U (and LARS*) 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.

7.2 Recommendation Quality for Varying k

These experiments test recommendation quality improvement of LARS*, LARS*-U, and LARS*-T for different values of k (i.e., recommendation answer sizes). We do not plot LARS with LARS* as both gives the same result for M=1, and the quality experiments are meant to show how the degree of locality increases the recommendation quality. We perform experiments using both the Foursquare and MovieLens data. Our quality metric is exactly the same as presented previously in Section 7.1.

Figure 8(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 7.1 in Figure 7(a)). For all sizes of k from one to ten, LARS* and LARS*-U consistently exhibit better quality. In fact, LARS* consistently achieves better quality over 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 8(b) depicts the effect of the recommendation list size k on the quality of improvement of LARS*-U (and LARS*) over CF using the MovieLens data. Notice that LARS* gives the same quality improvement as LARS*-U because LARS*-T do not apply for this dataset 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 7(b)). Again, LARS*-U (and LARS*) consistently exhibits better quality than CF for sizes of K from one to ten.

7.3 Recommendation Quality for Varying $\mathcal{M}$

These experiments compares the quality improvement achieved by both LARS and LARS* for different values of

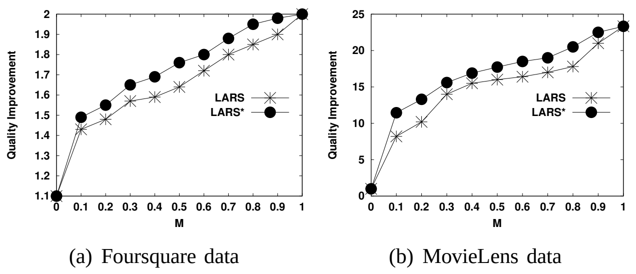


Fig. 9. Quality experiments for varying value of $\mathcal{M}$

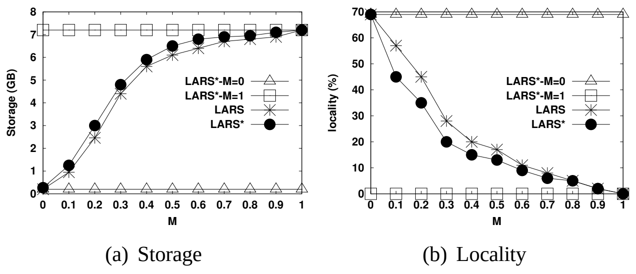


Fig. 10. Effect of $\mathcal{M}$ on storage and locality (Synthetic data)

$\mathcal{M}$. We perform experiments using both the Foursquare and MovieLens data. Our quality metric is exactly the same as presented previously in Section 7.1.

Figure 9(a) depicts the effect of $\mathcal{M}$ on the quality of both LARS and LARS* using the Foursquare data set. Notice that we enable both the user partitioning and travel penalty techniques for both LARS and LARS*. We report quality numbers using the pyramid height of four and the number of recommended items of ten. When $\mathcal{M}$ is equal to zero, both LARS and LARS* exhibit the same quality improvement as $\mathcal{M} = 0$ represents a traditional collaborative filtering with the travel penalty technique applied. Also, when $\mathcal{M}$ is set to one, both LARS and LARS* achieve the same quality improvement as a fully maintained pyramid is maintained in both cases. For $\mathcal{M}$ values between zero and one, the quality improvement of both LARS and LARS* increases for higher values of $\mathcal{M}$ due to the increase in recommendation locality. LARS* achieves better quality improvement over LARS because LARS* maintains α-Cells at lower levels of the pyramid.

Figure 9(b) depicts the effect of $\mathcal{M}$ on the quality of both LARS and LARS* using the Movilens data set. We report quality improvement over traditional collaborative filtering using the pyramid height of seven and the number of recommended items set to ten. Similar to Foursquare data set, the quality improvement of both LARS and LARS* increases for higher values of $\mathcal{M}$ due to the increase in recommendation locality. For $\mathcal{M}$ values between zero and one, LARS* consistently achieves higher quality improvement over LARS as LARS* maintains more α-Cells at more granular levels of the pyramid structure.

7.4 Storage Vs. Locality

Figure 10 depicts the impact of varying $\mathcal{M}$ on both the storage and locality in LARS* using the synthetic data set. We plot

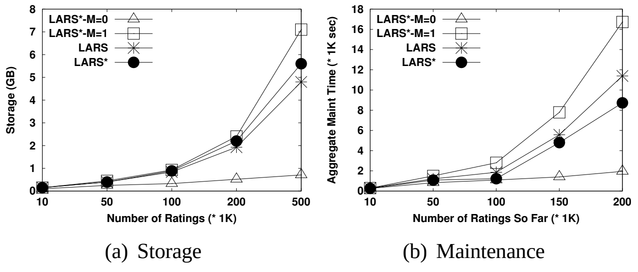
LARS*-M=0 and LARS*-M=1 as constants to delineate the extreme values of $\mathcal{M}$, i.e., $\mathcal{M}$=0 mirrors traditional collaborative filtering, while $\mathcal{M}$=1 forces LARS* to employ a complete pyramid. Our metric for locality is *locality loss* (defined in Section 4.5.2) when compared 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 switching cells to α-Cells, requiring the maintenance of more pyramid cells each with its own collaborative filtering model. Each additional α-Cell incurs a high storage overhead over the original data size as an additional collaborative filtering model needs to be maintained. 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. LARS* leads to smaller locality loss ($\approx$26% less) than LARS because LARS* maintains α-Cells below β-Cells which result in higher locality gain. On the other hand, LARS* exhibits slightly higher storage cost ($\approx$5% more storage) than LARS due to the fact that LARS* stores the *Item Ratings Statistics Table* per each α-Cell and β-Cell.

7.5 Scalability

Figure 11 depicts the storage and aggregate maintenance overhead required for an increasing number of ratings using the synthetic data set. We again plot LARS*-M=0 and LARS*-M=1 to indicate the extreme cases for LARS*. Figure 11(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 11(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 switching some cells from α-Cells to β-Cells. 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. Compare to LARS, LARS* has less maintenance overhead ($\approx$38% less) due to the fact that the maintenance algorithm in LARS* avoids the expensive speculative splitting used by LARS.

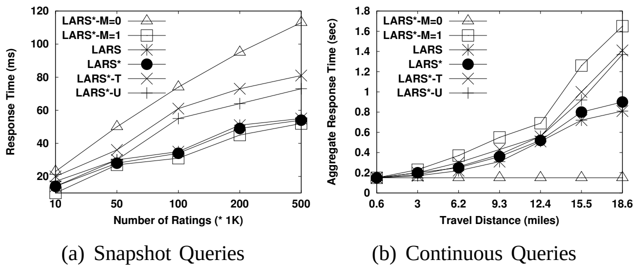
7.6 Query Processing Performance

Figure 12 depicts snapshot and continuous query processing performance of LARS, 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), using the synthetic data set.



(a) Storage (b) Maintenance

Fig. 11. Scalability of the adaptive pyramid (Synthetic data)



(a) Snapshot Queries (b) Continuous Queries

Fig. 12. Query Processing Performance (Synthetic data).

Snapshot queries. Figure 12(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. LARS* performance is slightly better than LARS as LARS* sometimes maintains more localized CF models than LARS which incurs less query processing time.

Continuous queries. Figure 12(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 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 7.1). 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 switching cells from α-Cells to β-Cells: 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 switching cells to β-Cells to reduce update frequency. LARS has a slightly better performance than LARS* as LARS tends to merge more cells at higher levels in the pyramid structure.

8 Related Work

Location-based services. Current location-based services employ two main methods to provide interesting destinations to users. (1) KNN techniques [22] and variants (e.g., aggregate KNN [24]) 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 [25] (and spatial variants [26]) and location-based top-k methods [27] 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 items.

Recent research has proposed the problem of hyper-local place ranking [28]. 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.

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 [4] 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) [29]. 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 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 [30]. LARS*, on the other hand, produces personalized recommendations influenced by location-based ratings and a query location.

Location-aware recommenders. The CityVoyager system [31] 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 [32] 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 [33] 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., partial pyramid structure, early query termination) necessary for deployment in actual large-scale applications.

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

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

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

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

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

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

[8] J. J. Levandoski, M. Sarwat, A. Eldawy, and M. F. Mokbel, "LARS: A Location-Aware Recommender System," in *Proceedings of the International Conference on Data Engineering, ICDE*, 2012.

[9] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, “Item-Based Collaborative Filtering Recommendation Algorithms,” in *Proceedings of the International World Wide Web Conference, WWW*, 2001.

[10] J. S. Breese, D. Heckerman, and C. Kadie, “Empirical Analysis of Predictive Algorithms for Collaborative Filtering,” in *Proceedings of the Conference on Uncertainty in Artificial Intelligence, UAI*, 1998.

[11] W. G. Aref and H. Samet, “Efficient Processing of Window Queries in the Pyramid Data Structure,” in *Proceedings of the ACM Symposium on Principles of Database Systems, 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] A. Guttman, “R-trees: A dynamic index structure for spatial searching,” in *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 1984.

[14] K. Mouratidis, S. Bakiras, and D. Papadias, “Continuous monitoring of spatial queries in wireless broadcast environments,” *IEEE Transactions on Mobile Computing, TMC*, vol. 8, no. 10, pp. 1297–1311, 2009.

[15] K. Mouratidis and D. Papadias, “Continuous nearest neighbor queries over sliding windows,” *IEEE Transactions on Knowledge and Data Engineering, TKDE*, vol. 19, no. 6, pp. 789–803, 2007.

[16] M. F. Mokbel et al, “SINA: Scalable Incremental Processing of Continuous Queries in Spatiotemporal Databases,” in *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2004.

[17] J. L. Herlocker, J. A. Konstan, L. G. Terveen, and J. T. Riedl, “Evaluating Collaborative Filtering Recommender Systems,” *ACM Transactions on Information Systems, TOIS*, vol. 22, no. 1, pp. 5–53, 2004.

[18] M. J. Carey et al, “On saying ”Enough Already!” in SQL,” in *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 1997.

[19] S. Chaudhuri et al, “Evaluating Top-K Selection Queries,” in *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 1999.

[20] R. Fagin, A. Lotem, and M. Naor, “Optimal Aggregation Algorithms for Middleware,” in *Proceedings of the ACM Symposium on Principles of Database Systems, PODS*, 2001.

[21] J. Bao, C.-Y. Chow, M. F. Mokbel, and W.-S. Ku, “Efficient evaluation of k-range nearest neighbor queries in road networks,” in *Proceedings of the International Conference on Mobile Data Management, MDM*, 2010.

[22] G. R. Hjaltason and H. Samet, “Distance Browsing in Spatial Databases,” *ACM Transactions on Database Systems, TODS*, vol. 24, no. 2, pp. 265–318, 1999.

[23] K. Mouratidis, M. L. Yiu, D. Papadias, and N. Mamoulis, “Continuous nearest neighbor monitoring in road networks,” in *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2006.

[24] D. Papadias, Y. Tao, K. Mouratidis, and C. K. Hui, “Aggregate Nearest Neighbor Queries in Spatial Databases,” *ACM Transactions on Database Systems, TODS*, vol. 30, no. 2, pp. 529–576, 2005.

[25] S. Börzsönyi et al, “The Skyline Operator,” in *Proceedings of the International Conference on Data Engineering, ICDE*, 2001.

[26] M. Sharifzadeh and C. Shahabi, “The Spatial Skyline Queries,” in *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2006.

[27] N. Bruno, L. Gravano, and A. Marian, “Evaluating Top-k Queries over Web-Accessible Databases,” in *Proceedings of the International Conference on Data Engineering, ICDE*, 2002.

[28] 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.

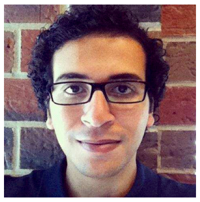
[29] M.-H. Park et al, “Location-Based Recommendation System Using Bayesian User’s Preference Model in Mobile Devices,” in *Proceedings of the International Conference on Ubiquitous Intelligence and Computing, UIC*, 2007.

[30] “Netflix News and Info - Local Favorites: http://tinyurl.com/4qt8ujo.”

[31] Y. Takeuchi and M. Sugimoto, “An Outdoor Recommendation System based on User Location History,” in *Proceedings of the International Conference on Ubiquitous Intelligence and Computing, UIC*, 2006.

[32] V. W. Zheng, Y. Zheng, X. Xie, and Q. Yang, “Collaborative Location and Activity Recommendations with GPS History Data,” in *Proceedings of the International World Wide Web Conference, WWW*, 2010.

[33] M. Ye, P. Yin, and W.-C. Lee, “Location Recommendation for Location-based Social Networks,” in *Proceedings of the ACM Symposium on Advances in Geographic Information Systems, ACM GIS*, 2010.



Mohamed Sarwat is a doctoral candidate at the Department of Computer Science and Engineering, University of Minnesota. He obtained his Bachelor′s degree in computer engineering from Cairo University in 2007 and his Master′s degree in computer science from University of Minnesota in 2011. His research interest lies in the broad area of data management systems. More specifically, some of his interests include database systems (i.e., query processing and optimization, data indexing), database support for recommender systems, personalized databases, database support for location-based services, database support for social networking applications, distributed graph databases, and large scale data management. Mohamed has been awarded the University of Minnesota Doctoral Dissertation Fellowship in 2012. His research work has been recognized by the Best Research Paper Award in the 12th international symposium on spatial and temporal databases 2011.

Justin J. Levandoski is a researcher in the database group at Microsoft Research. Justin received his Bachelor′s degree at Carleton College, MN, USA in 2003, and his Master′s and PhD degrees at the University of Minnesota, MN, USA in 2008 and 2011, repecitvely. His research lies in a broad range of topics dealing with large-scale data management systems. More specifically, some of his interests include cloud computing, database support for new hardware paradigms, transaction processing, query processing, and support for new data-intensive applications such as social/recommender systems.

Ahmed Eldawy is a PhD student at the Department of Computer Science and Engineering, the University of Minnesota. His main research interests are spatial data management, social networks and cloud computing. More specifically, his research focuses on building scalable spatial data management systems over cloud computing platforms. Ahmed received his Bachelor′s and Master′s degrees in Computer Science from Alexandria University in 2005 and 2010, respectively.

Mohamed F. Mokbel (Ph.D., Purdue University, 2005, MS, B.Sc., Alexandria University, 1999, 1996) is an associate professor in the Department of Computer Science and Engineering, University of Minnesota. His main research interests focus on advancing the state of the art in the design and implementation of database engines to cope with the requirements of emerging applications (e.g., location-based applications and sensor networks). His research work has been recognized by three best paper awards at IEEE MASS 2008, MDM 2009, and SSTD 2011. Mohamed is a recipient of the NSF CAREER award 2010. Mohamed has actively participated in several program committees for major conferences including ICDE, SIGMOD, VLDB, SSTD, and ACM GIS. He is/was a program co-chair for ACM SIGSPATIAL GIS 2008, 2009, and 2010. Mohamed is an ACM and IEEE member and a founding member of ACM SIGSPATIAL.