StreamRec: A Real-Time Recommender System

Badrish Chandramouli[1] Justin J. Levandoski[2§] Ahmed Eldawy[2§] Mohamed F. Mokbel[2§]

[1]Microsoft Research, Redmond, WA, badrishc@microsoft.com
[2]University of Minnesota, Minneapolis, MN, {justin,eldawy,mokbel}@cs.umn.edu

1. INTRODUCTION

Research and development of recommender systems has been a vibrant field for over a decade, having produced proven methods for "preference-aware" computing. Recommenders use community opinion histories to help users identify interesting items from a considerably large search space (e.g., inventory from Amazon [7], movies from Netflix [9]). Personalization, recommendation, and the "human side" of data-centric applications are also becoming important topics in the data management community [3].

A popular recommendation method used heavily in practice is *collaborative filtering*, consisting of two phases: (1) An *offline model-building* phase that uses community opinions of items (e.g., movie ratings, "Diggs" [6]) to build a model storing meaningful correlations between users and items. (2) An *on-demand recommendation* phase that uses the model to produce a set of recommended items when requested from a user or application.

To be effective, recommender systems must evolve with their content. In current update-intensive systems (e.g., social networks, online news sites), the restriction that a model be generated *offline* is a significant drawback, as it hinders the system's ability to evolve quickly. For instance, new users enter the system changing the collective opinions over items, or the system adds new items quickly (e.g., news posts, Facebook postings), which widens the recommendation pool. These updates affect the recommender model, that in turn affect the system's recommendation quality in terms of providing accurate answers to recommender queries. In such systems, a completely *real-time* recommendation process is paramount. Unfortunately, most traditional state-of-the-art recommenders are "hand-built", implemented as custom software *not* built for a real-time recommendation process [1]. Further, for some scenarios, a purely request-based recommendation model does not scale with the number of subscribers; this suggests the need for more "push-based" recommendation schemes.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems

General Terms

Algorithms, Design, Human Factors

§The research of these authors 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

Copyright is held by the author/owner(s).
SIGMOD'11, June 12–16, 2011, Athens, Greece.
ACM 978-1-4503-0661-4/11/06.

In this demonstration, we propose *StreamRec*, a recommender system architecture that leverages a stream processing system [4, 5]. *StreamRec* addresses the drawbacks of more traditional recommender systems through two salient features: (1) *Real-time incremental processing*. Streaming systems are architected for high-throughput processing, where query operators are tuned for incremental evaluation, meaning the recommendation process (model building and recommendation generation) can be performed in real-time. (2) *Push-based subscriptions*. Users can register long-running recommendation requests, updated only when their recommendation list changes; in some cases, this approach is more scalable than on-demand systems that regenerate whole recommendations from scratch for each query. Alternatively, requests can have short lifetimes, causing *StreamRec* to revert to an on-demand system.

The basic idea of *StreamRec* is to model a recommendation system as a complex event processing (CEP) application. We show that well-known collaborative filtering recommender models can be expressed using *only* native incremental streaming operators. *StreamRec* is scalable, as all the operations used in our solution are parallelizable. Moreover, the entire recommender can be expressed using a *single* stream query plan. *StreamRec* accepts two input stream types: (1) *Update events*: a stream of new user opinions (e.g., movie ratings), used to incrementally update the recommender model and (2) *Recommend events*: a stream of requests to produce recommendations (e.g., "recommend user Alice 10 movies"). Recommendations can be *pull-based* (e.g., on-demand) or *push-based*, which are registered with *StreamRec* over an extended period. Recommendations are produced by joining *recommend events* with the maintained recommender model. In addition, *StreamRec* can easily provide recommendation *freshness*; using event windows, older "stale" opinions (e.g., ratings) can decay within the recommender model over time, meaning recommendations rely on current opinion trends and remain "fresh". We implement *StreamRec* using the Microsoft StreamInsight stream processing system [2].

In the rest of this paper, we provide details of our *StreamRec* demo. Section 2 provides background information, while Section 3 provides the details of *StreamRec*. Finally, Section 4 describes our *StreamRec* demonstration and application scenario.

2. BACKGROUND

Collaborative Filtering. Our demo uses collaborative filtering as its recommendation approach, a popular method used in real-world systems [7]. 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 ranking (e.g., one to five stars in Netflix [9]), or unary (e.g., a "Digg" [6]). Given a querying user u_q,

CF produces a set of k recommended items $\mathcal{I}_r \subset \mathcal{I}$ that u_q is predicted to like the most. There are many CF paradigms (see [1] for a comprehensive survey). Each follows a similar two-phase model-building then recommendation generation approach, described below for the popular item-based CF method used this demo.

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 (represented as vectors in the user-rating space) that have at least one co-rated dimensions. In this demo we use Cosine similarity as our measure due to its popularity [7], computed as:

$$sim(i_p, i_q) = k \frac{\vec{i_p} \cdot \vec{i_q}}{\|\vec{i_p}\| \|\vec{i_q}\|} \quad (1)$$

A model is built that stores for each item item $i \in \mathcal{I}$, a list $\mathcal{L}$ of similar items ordered by a similarity score *sim*(i_p,i_q).

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

$$P_{(u_t,i)} = \frac{\sum_{l \in \mathcal{L}} sim(i,l) * r_{u_q,l}}{\sum_{l \in \mathcal{L}} |sim(i,l)|} \quad (2)$$

Before this computation, we reduce each similarity list $\mathcal{L}$ to contain only items *rated* by user u_q. The prediction is the sum of $r_{u_q,l}$, the user'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 $P_{(u_t,i)}$.

Stream Processing Systems. A *stream* is a sequence $e_1, e_2, \ldots, e_n$ of events. An *event* $e_i = \langle p, c \rangle$ is an outside notification (e.g., user rating) that consists of a *payload* $p = \langle p_1, \ldots, p_k \rangle$ (e.g., rating value), and event *metada* c. While metadata varies across systems [5, 10], two common notions are: (1) an event generation time, and (2) a time window, which indicates the period of time over which an event can influence output. We capture these by defining $c = \langle \mathsf{LE}, \mathsf{RE} \rangle$, where the time interval $[\mathsf{LE}, \mathsf{RE})$ specifies the period (or *lifetime*) over which the event contributes to output. The left endpoint (LE) of this interval is the application time of event generation, also called the event *timestamp*.

3. StreamRec DESCRIPTION

Figure 1 depicts part of StreamRec's fully incremental continuous query plan for end-to-end item-based collaborative filtering. The plan covers model generation as well as similarity scoring and recommendation. The input to our recommender consists of two streaming events: (1) *Update events*, which are user ratings for items, and (2) *Recommend events*, which are requests for recommendations for a target user. We model the input using a common schema (Timestamp, StreamId, UserId, ItemId, Rating). An event with StreamId=0 denotes a new rating for an item, while StreamId=1 denotes a request for recommendation by a user (in the latter case, ItemId and Rating are null). Recommendation requests can be registered with *StreamRec* by using: (1) *Edge events*, which sets the event lifetime end as $\mathsf{RE} = \infty$. *Edge events* allow users to "subscribe" to their recommendation list over a period of time. This approach is geared toward *push-based* applications, where *StreamRec* only sends updates (changes) to the user recommendation list. (2) *Point events*, which are"instantaneous" events with no lifetime, where RE is set to $\mathsf{LE} + \delta$ where δ is the smallest possible time-unit. *Point events* allow the user to receive a one-time on-demand recommendation list. This approach is geared toward *pull-based* applications

Model Building. Model building works as follows:

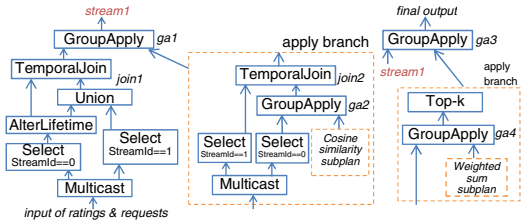

Figure 1: Part of StreamRec's recommender plan

- We first perform an AlterLifetime operation on ratings events to control the window of historical ratings that are used for building the model. Applying this windows allows the system to decay older "stale" user opinions over time, allowing the model to rely on newer "fresh" user opinions (an infinite window implies ratings will never decay).
- For each rating event for item *Item1* by user *User1*, we perform a temporal self-equi-join on UserId using the TemporalJoin operator (*join1*) to produce events (UserId, Item1, Rating1, Item2, Rating2) for every pair of items rated by *User1*. The TemporalJoin operator allows correlation between two streams. It outputs the relational join between its left and right input events. Streaming systems typically implement TemporalJoin as a symmetric hash join, where events along each input are stored in a separate internal *join synopsis*. Further, each join output has a lifetime consisting of the intersection of the joining event lifetimes.
- Events from *join1* are fed into the GroupApply operator (*ga1*) with grouping key *Item1*, followed by a second GroupApply (*ga2*) with grouping key *Item2*. The GroupApply operator allows us to specify a grouping key, and a query sub-plan (called *apply branch*) to be "applied" to each group (depicted as a dotted box in Figure 1). Within each group in *ga1*, we produce an aggregate cosine similarity across all users for each item that *Item1* pairs with, producing a stream of (Item2, SimScore) events. A streaming aggregation operator (e.g., Average, Top-k) reports a result each time the active event set changes. The similarity metric is computed in an *incremental manner*, by using one built-in sum operator for each term in Equation 1, followed by Project to compute the similarity score. For unary ratings (e.g., Diggs [6]), we substitute the vector similarity for cosine similarity, which, interestingly, can still be expressed using compositions of native streaming operators (details omitted for brevity).
- These events serve as input to a TemporalJoin *join2*—still within *ga1*—that effectively holds the model in memory (in the right join synopsis) for scoring in the future.

Recommendation Generation. Recommendation generation works as follows:

- When a new recommendation request event arrives for user *User1*, we first send the event to *join1*, in order "look up" the previously rated items (*Item1*) for *User1*.
- The join results are grouped by *Item1*, by re-using GroupApply *ga1*. Within the group, for each item *Item1*, we "look up" similarity scores of related items using *join2*, and produce a stream (as output of *ga1*) that contains, for every item *Item1* rated by *User1*, an event for every other item (say *Item2*) that *Item1* is similar to.
- These events are re-grouped by UserId using GroupApply *ga3*. For each user, we group by the second item (*Item2*) using another GroupApply *ga4*, and an aggregate (weighted sum) is used to compute the predicted recommendation score of *Item2* for *User1* (i.e., Equation 2).

(a) MSRNews (b) MSRFlix

Figure 2: Demonstration Applications

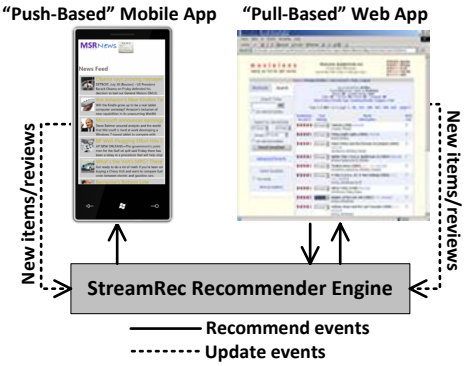

Figure 3: Demonstration scenario overview

- We eliminate items that have been previously rated by the user, using a *LeftAntiSemiJoin* operator (details omitted). Finally, we use a Top-k aggregation operator to report the final recommendation for *User1*.

In case a request arrives as an edge event, the corresponding events gets lodged in the right synopsis of join1 and the left synopsis of join2. As a consequence, any change to either items rated by the user or to the model itself causes the query to produce an update to the top-k result that is then pushed to the user.

We implement the entire recommendation process using *only* native stream operators, ensuring high performance. Every operator is either stateless, within a GroupApply, or is an equijoin, which implies that the computation can easily *scale out* on a cluster.

4. DEMONSTRATION SCENARIO

4.1 Application

We provide two applications built specifically for this demo. (1) *MSRNews*, a social news application (similar to Digg). *MSRNews* uses *StreamRec* to provide personalized news feeds using collective user feedback (e.g., "likes") for popular news postings. (2) *MSRFlix*, a movie recommendation application. *MSRFlix* uses *StreamRec* to provide movie recommendations using collective user ratings for movies. Both applications come in two versions: (a) *Mobile-based*, implemented as a Microsoft Windows Phone 7 application, depicted in Figure 2, and (b) *Web-based*, displayed in a standard web browser (screenshot omitted for space). In both applications users perform three basic tasks: (1) *Get recommendations* from the system (e.g., news feeds, movies), (2) *Give opinions* (e.g., "like" news posts, rate movies), (3) *Add* new items to the system (e.g., news posts, movies). Each action triggers an event in the underlying *StreamRec* recommender system, implemented in Microsoft StreamInsight, as depicted in Figure 3. Specifically, the *get recommendations* task triggers a *recommend event* (depicted as a solid line in Figure 3), while the *give opinion* and *add* tasks trigger *update events* (depicted as dashed lines in Figure 3).

4.2 Data, Applications, and Queries

Data. To build the initial recommendation model in *MSRFlix*, we use the popular MovieLens open-source movie ratings data [8]. The rating domain is a user-specified value in the range of [1-5]. For *MSRNews*, we use publicly available data from the Digg [6] social news website with a unary rating domain where users can only "like" a news item. For both applications, attendees will generate new data as they use the system. This new data will instantly integrate into *StreamRec*, producing fresh relevant recommendations.

Push and Pull Applications. We use two application types to showcase *subscription* and *on-demand* recommendations in *StreamRec* (depicted in Figure 3). (1) *Push-based*. The mobile version of our applications receive push-based recommendations, meaning updates to user recommendation lists are pushed to the device when *StreamRec* detects a change in a user's top-n recommendations. (2) *Pull-based*. The web version of our applications receive recommendations for a user only after explicitly submitting a recommend request to the underlying *StreamRec* engine.

4.3 Walkthrough

Our walkthrough showcases the features of *StreamRec* in three phases : (1) *Initial recommendations*. The attendee will register with *MSRNews* or *MSRFlix*, and give opinions on a number of existing news items or movies in the system. This action will cause *StreamRec* to provide the attendee with an initial set of top-n recommendations. (2) *Witness instant updates*. Using the mobile version of *MSRNews* or *MSRFlix*, the attendee will open their recommendation list, and witness it changing even though they have not rated any new items. This behavior is the result of *StreamRec* pushing updates to the recommendation list as *other users* add or rate new items in the system (thereby changing the recommender model). (3) *Witness freshness*. Without *any* other user activity, attendees will see their recommendation list change, as *StreamRec* decays older ratings in the system in favor of newer ratings (described in Section 3), thereby keeping recommendations *fresh*.

As *StreamRec* is a fully developed systems, the attendee will also be free to use our demo applications in an ad-hoc manner. Access to StreamRec's backend (implemented using StreamInsight as the underlying engine) will also be available through Visual Studio 2010, where attendees can visualize query plans, analyze the event flow graphically to see how/why particular ratings were generated, and examine performance statistics on a per-operator basis.

5. REFERENCES

[1] G. Adomavicius and A. Tuzhilin. Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions. *TKDE*, 17(6), 2005.
[2] M. Ali et al. Microsoft CEP Server and Online Behavioral Targeting. In *VLDB*, 2009 (demonstration).
[3] S. Amer-Yahia et al. Crowds, Clouds, and Algorithms: Exploring the Human Side of Big Data Applications. In *SIGMOD*, 2010.
[4] B. Babcock et al. Models and issues in data stream systems. In *PODS*, 2002.
[5] R. Barga et al. Consistent streaming through time: A vision for event stream processing. In *CIDR*, 2007.
[6] Digg: http://digg.com.
[7] G. Linden et al. Amazon.com Recommendations: Item-to-Item Collaborative Filtering. *IEEE Internet Computing*, 7(1), 2003.
[8] MovieLens: http://www.movielens.org.
[9] Netflix: http://www.netflix.com.
[10] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *PODS*, 2004.