Dynamic Prefetching of Data Tiles for Interactive Visualization

ABSTRACT

In this paper, we present ForeCache, a general-purpose tool for exploratory browsing of large datasets. ForeCache utilizes a client-server architecture, where the user interacts with a lightweight client-side interface to browse datasets, and the data to be browsed is retrieved from a DBMS running on a back-end server. We assume a detail-on-demand browsing paradigm, and optimize the back-end support for this paradigm by inserting a separate middleware layer in front of the DBMS. To improve response times, the middleware layer fetches data ahead of the user as she explores a dataset.

We consider two different mechanisms for prefetching: (a) learning what to fetch from the user's recent movements, and (b) using data characteristics (*e.g.,* histograms) to find data similar to what the user has viewed in the past. We incorporate these mechanisms into a single prediction engine that adjusts its prediction strategies over time, based on changes in the user's behavior. We evaluated our prediction engine with a user study, and found that our dynamic prefetching strategy provides: (1) significant improvements in overall latency when compared with non-prefetching systems (430% improvement); and (2) substantial improvements in both prediction accuracy (25% improvement) and latency (88% improvement) relative to existing prefetching techniques.

1. INTRODUCTION

Exploratory browsing helps users analyze large amounts of data quickly by rendering the data at interactive speeds within a viewport of fixed size (*e.g.,* a laptop screen). This is of particular interest to data scientists, because they do not have the time or resources to analyze billions of datapoints by hand. One common interaction pattern we have observed in data scientists is that they analyze a small region within a larger dataset, and then move to a nearby region and repeat the same analysis. They initially aggregate or sample these regions when looking for a quick answer, and zoom into the raw data when an exact answer is needed. Thus, we focus on supporting a detail-on-demand browsing paradigm, where users can move to different regions within a single dataset, and zoom into these regions to see them in greater detail.

While users want to be able to drill down into specific regions of a dataset, they also want their actions within the browsing tool to be fluid and interactive. Even one second of delay after a pan or zoom can be frustrating for users, hindering their analyses and distracting them from what the data has to offer [16, 14]. Thus, the goal of this project is to make all user interactions extremely fast (*i.e.,* 500 ms or less), thereby providing a seamless exploration experience for users. However, though modern database management systems (DBMS's) allow users to perform complex scientific analyses over large datasets [19], DBMS's are not designed to respond to queries at interactive speeds, resulting in long interaction delays for browsing tools that must wait for answers from a backend DBMS [2]. Thus, new optimization techniques are needed to address the non-interactive performance of modern DBMS's, within the context of exploratory browsing.

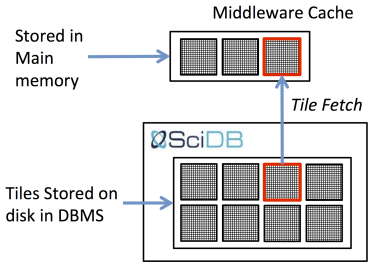


Figure 1: A diagram of ForeCache's tile storage scheme.

In this paper, we present ForeCache, a general-purpose tool for exploratory browsing of large datasets at interactive speeds. Given that data scientists routinely analyze datasets that do not fit in main memory, ForeCache utilizes a client-server architecture, where users interact with a lightweight client-side interface, and the data to be explored is retrieved from a back-end server running a DBMS. We use the array-based DBMS SciDB as the back-end [21], and insert a middleware layer in front of the DBMS, which utilizes prefetching techniques and a main-memory cache to speedup server-side performance.

When the user performs zooms in ForeCache, she expects to see more detail from the underlying data. To support multiple levels of detail, we apply aggregation queries to the raw data. However, these queries are slow, and may not execute at interactive speeds. To ensure that zooms are fast in ForeCache, we compute each level of detail, or *zoom level*, beforehand, and store them on disk. A separate materialized view is created for each zoom level, and we partition each zoom level into equal-size blocks, or *data tiles* [15].

The user cycles through the following steps when browsing data in ForeCache: (1) she analyzes the result of the previous request, (2) performs an action in the interface to update or refine the request (*e.g.,* zooms in), and then (3) waits for the result to be rendered on the screen. ForeCache eliminates step 3 by prefetching neighboring tiles and storing them in main memory while the user is still

in step 1, thereby providing the user with a seamless browsing experience. At the middleware level, we incorporate a main-memory cache for fetching computed tiles, shown in Figure 1. When tiles are prefetched, they are copied from SciDB to the cache. However, in a multi-user environment, there may be too little space on the server to cache all neighboring tiles for every user. Thus, we must rank the tiles first, and fetch only the most likely candidates.

While prefetching is known to be effective, ForeCache needs access to the user's past interactions with the interface to predict future data requests. We have observed that the client has extensive records of the user's past interactions, which we can leverage to improve our prefetching strategy. For example, the client knows what regions the user has visited in the past, and what actions she has recently performed. One straightforward optimization is to train a Markov model on the user's past actions, and to use this model to predict the user's future actions [6, 7]. We refer to these prediction techniques as *recommendation models* throughout this paper.

However, the user's actions are often too complex to be described by a single model (which we will show in Section 5). Thus, existing models only cover a fraction of possible browsing strategies, leading to longer user wait times due to prediction errors. A comprehensive approach is needed, such that we can consistently prefetch the right tiles over a diverse range of browsing strategies.

To address the limitations of existing techniques, we have designed a new two-level prediction engine for our middleware. At the top level, our prediction engine learns the user's current browsing strategy, given her most recent actions. Users frequently employ several low-level browsing behaviors as part of their browsing strategies (*e.g.,* panning right three times in a row). Therefore at the bottom level, our prediction engine runs multiple recommendation models in parallel, each designed to model a specific low-level browsing behavior. Using this two-level design, our prediction engine tracks shifts in the user's browsing strategy, and updates its prediction strategy accordingly. To do this, we increase or decrease the space allotted to each low-level recommendation model for predictions. Furthermore, this two-level design can be applied to other components within the ForeCache architecture. For example, we plan to extend our techniques to improve caching and prefetching performance within the DBMS.

Taking inspiration from other user analysis models [18], we have observed that the space of possible browsing strategies can be partitioned into three separate *analysis phases*: Foraging (analyzing individual tiles at a coarse zoom level to form a new hypothesis), Sensemaking (comparing neighboring tiles at a detailed zoom level to test the current hypothesis), and Navigation (moving between coarse and detailed zoom levels to transition between the previous two phases). The user's goal changes depending on which phase she is currently in. For example, in the Navigation phase, the user is shifting the focus of her analysis from one region in the dataset to another. In contrast, the user's goal in the Foraging phase is to find new regions that exhibit interesting data patterns. Thus, users will also employ a different browsing strategy for each phase.

We consider two separate mechanisms for our low-level recommendation models: (a) learning what to fetch based on the user's past movements (*e.g.,* given that the the user's last three moves were all to "pan right," what should be fetched?) [7]; and (b) using data-derived characteristics, or *signatures*, to identify neighboring tiles that are similar to what the user has seen in the past. We use a Markov chain to model the first mechanism, and a suite of signatures for the second mechanism, ranging from simple statistics (*e.g.,* histograms) to sophisticated machine vision features.

To evaluate ForeCache, we conducted a user study, where domain scientists explored satellite imagery data. Our results show

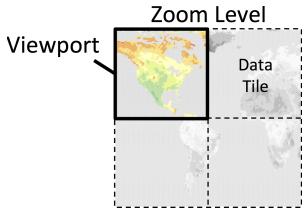


Figure 2: A tiling example with satellite imagery data.

that ForeCache achieves (near) interactive speeds for data exploration (*i.e.,* average latency within 500 ms). We also found that ForeCache achieves: (1) dramatic improvements in latency compared with traditional non-prefetching systems (430% improvement in latency); and (2) higher prediction accuracy (25% better accuracy) and significantly lower latency (88% improvement in latency), compared to existing prefetching techniques.

In this paper, we make the following contributions:

1. We propose a new three-phase analysis model to describe how users generally explore array-based data.
2. We present a tile-based data model for arrays, and architecture for supporting interactive browsing of tiles in SciDB,
3. We present our two-level prediction engine, with an SVM classifier at the top level to predict the user's current analysis phase, and recommendation models at the bottom to predict low-level user interactions.
4. We present the results from our user study. Our results show that our approach provides higher prediction accuracy and significantly lower latency, compared to existing techniques.

2. DATA MODEL

In this section, we describe the kinds of datasets supported by ForeCache, and our process for building zoom levels and data tiles.

2.1 Datasets Supported by ForeCache

Our tiling structure supports browsing of any array-based dataset that can be stored in SciDB, such as time series datasets (*e.g.,* stock prices) and geospatial datasets (*e.g.,* satellite imagery). Given its extensive support for complex analytics over multidimensional datasets, we use SciDB as the back-end DBMS in ForeCache. Consider Figure 2, where the user happens to be exploring an array of satellite imagery, and each array cell has been mapped to a pixel. We have partitioned the current zoom level along the array's two dimensions (latitude and longitude), resulting in four data tiles. The user's current viewport is located at the top left data tile, and the user can move to other data tiles by performing actions in the client-side interface (*e.g.,* panning). The user can also zoom in or out to explore different zoom levels.

Building zoom levels over this array will produce high-resolution (finer) or lower-resolution (coarser) views of the underlying data. For example, we can make a zoom level with one quarter of the original resolution quality by computing the average pixel value of every 2x2 window of array cells.

2.2 Building Data Tiles

In this section, we explain how ForeCache builds zoom levels (and data tiles) for array-based data stored in SciDB. We conducted our experiments in Section 5 using two-dimensional data, so we focus on 2D arrays here. However, it is straightforward to extend our 2D tiling scheme to work for three or more dimensions.

The ForeCache tile building process is divided into three steps: (1) building a separate materialized view for each zoom level; (2) partitioning each zoom level into non-overlapping blocks of fixed size (*i.e.,* data tiles); and (3) computing any necessary metadata

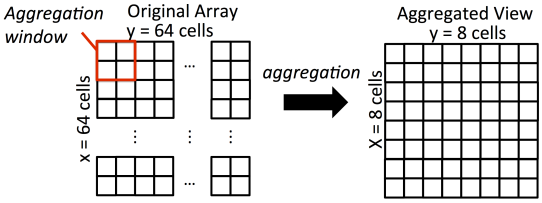


Figure 3: A 64x64 array being aggregated down to an 8x8 array with aggregation parameters $(8,8)$.

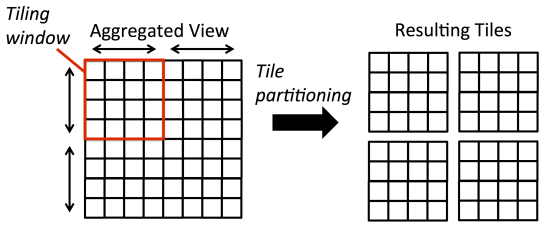


Figure 4: A zoom level being partitioned into four tiles, with tiling parameters $(4,4)$.

(*e.g.,* data statistics) for each data tile. The most detailed zoom level (*i.e.,* highest resolution) is just the raw data without any aggregation.

To build a materialized view, we apply an aggregation query to the raw data, where the aggregation parameters dictate how detailed the resulting zoom level will be. These parameters form a tuple $(j_1,j_2,...,j_d)$, where d is the number of dimensions. Each parameter j specifies an aggregation interval over the corresponding dimension, where every j array cells along this dimension are aggregated into a single cell. Consider Figure 3, where we have a 64x64 array (on the left), with two dimensions labeled x and y, respectively. Aggregation parameters of $(8,8)$ correspond to every 8 cells being aggregated along dimension x, and every 8 cells along dimension y. This aggregation window is denoted by the red box in Figure 3. If we compute the average cell value for each window in the 64x64 array, the resulting array will have dimensions 8x8 (right side of Figure 3).

Next, we partition each computed zoom level into data tiles. To do this, we assign a tiling interval to each dimension, which dictates number of aggregated cells contained in each tile along this dimension. For example, consider our aggregated view with dimensions 8x8, shown in Figure 4. If we specify a tiling window of $(4,4)$, ForeCache will partition this view into four separate data tiles, each with the dimensions we specified in our tiling parameters (4x4).

We choose the aggregation and tiling parameters such that a single tile at zoom level i corresponds to four higher-resolution data tiles at zoom level $i+1$. To do this, we calculated our zoom levels bottom-up (*i.e.,* starting at the raw data level), multiplying our aggregation intervals by 2 for each coarser zoom level going upward. We then applied the same tiling intervals to every zoom level.

Last, ForeCache computes any necessary metadata for each data tile. For example, some of our recommendation models rely on data characteristics, or signatures, to be computed for each tile, such as histograms or machine vision features (see Section 4 for more detail). As ForeCache processes each tile and zoom level, this metadata is computed and stored in a shared data structure for later use by our prediction engine.

3. ARCHITECTURE

ForeCache has four primary components in its client-server architecture: a web-based visualization interface on the client; a prediction engine; a tile cache manager; and the back-end DBMS. Figure 5 illustrates the relationship between these components. The visualizer sends tile requests to the tile prediction engine, and the prediction engine then sends these requests to the cache manager for retrieval. To anticipate the user's future requests, the prediction engine sends predictions to the cache manager to be fetched ahead of time. The cache manager stores predicted tiles in the middleware tile cache, and retrieves tiles requested by the client from either the middleware cache or SciDB. We elaborate on the first three components of the ForeCache architecture in the rest of this section.

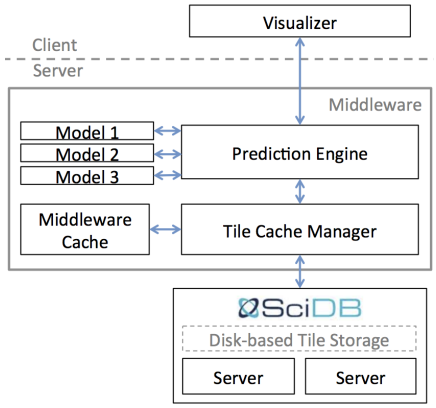


Figure 5: Overview of the ForeCache architecture.

3.1 Front-End Visualizer

ForeCache is agnostic to the front-end visualizer, which can be implemented in any language or platform on the client. The only requirement for the visualizer is that it must interact with the back-end through tile requests. As the user interacts with the client-side interface, requests are sent to the back-end to retrieve the corresponding tiles. For our experiments, we implemented a lightweight, front-end visualization interface that runs in a web browser on the client machine (see Section 5 for more details).

3.2 Prediction Engine

The purpose of the prediction engine is to manage ForeCache's tile requests, which consists of two major tasks: responding to tile requests from the client; and managing both the high-level (analysis phase classifier) and low-level (recommendation models) predictions used to infer the user's future requests. To fulfill existing tile requests, the prediction engine retrieves requested tiles from the cache manager, and sends them back to the client for visualization. This component also manages ForeCache's predictions. At the topmost level, this component runs an SVM classifier that predicts the user's current analysis phase. At the lower level, the prediction engine manages several tile recommendation models running in parallel. These recommendation models make suggestions for what data tiles to fetch, and the actual fetching of the tiles is left to the cache manager. After each user request, the prediction engine retrieves a separate list of predictions from each recommender, and forwards these predictions to the cache manager for further processing. We describe our SVM classifier and recommendation models in detail in Section 4.

3.3 Tile Cache Manager

The cache manager is in charge of managing tiles in the middleware cache. At the middleware level, the cache manager decides which tiles will be stored in our main memory cache. Each recommendation model is allotted a limited amount of space in this cache to make predictions. The remaining space in the middleware

cache is used to store the last n tiles requested by the interface. We refer to the allocations made to our recommendation models as the cache manager's current *allocation strategy*. This allocation strategy is reevaluated after each request to ensure that space is allocated efficiently across all models. We explain how we update these allocations in Section 4. The cache manger then uses the tile recommendations from the prediction engine to fill the cache once the allocations have been updated.

4. PREDICTION ENGINE DESIGN

The goal of our prediction engine is to identify changes in the user's browsing behavior, and update its prediction strategy accordingly. In this way, our prediction engine ensures that the most relevant prediction algorithms are being used to prefetch data tiles. To do this, our prediction engine makes predictions at two separate levels. At the top level, it learns the user's current browsing strategy. At the bottom level, it models the observed strategy with a suite of recommendation models.

We chose a two-level design because we have found that users frequently switch between browsing strategies over time. In contrast, recommendation models make strict assumptions about the user's browsing strategy, and thus ignore changes in the user's behavior. For example, Markov chains assume that the user's past movements will always be good indicators of her future actions. However, when the user has visually identified a new region to explore, the user's past actions are likely poor indicators for the future actions she will take to navigate towards this new area. As a result, we have found that recommendation models only work well in specific cases, making any individual model a poor choice for predicting the user's entire browsing session.

However, if we can learn what a user is trying to do, we can identify the most likely browsing strategy she will employ to accomplish her goals, and apply the corresponding recommendation model(s) to make predictions. To do this, we first outline the three general *analysis phases* that users alternate between when browsing array-based data. Each *analysis phase* represents a particular kind of browsing strategy, which we explain in Section 4.1.1.

Using these analysis phases, we build our two-level prediction engine. To build the top level of our prediction engine, we trained a classifier to predict the user's current analysis phase, given her past tile requests. To build the bottom level of our prediction engine, we developed a suite of recommendation models to capture the different browsing behaviors exhibited in our analysis phases. To combine the top and bottom levels, we developed three separate allocation strategies for our middleware cache, one for each analysis phase.

In the rest of this section, we explain the top and bottom level designs for our prediction engine, and how we combine the two levels using our allocation strategies. In Section 5, we explain how we used trace data from our user study to improve our allocation strategies and evaluate our prediction engine.

4.1 Top-Level Design

In this section, we explain the three analysis phases that users alternate between while browsing array-based data, and how we use this information to predict the user's current analysis phase.

4.1.1 Learning Analysis Phases

We informally observed several users browsing array-based data in SciDB, searching for common interaction patterns. We used these observed patterns to define a user analysis model, or a general-purpose template for user interactions in ForeCache. Our user analysis model was inspired in part by the well-known Sensemaking model [18]. However, we found that the Sensemaking Model did not accurately represent the behaviors we observed. For example, the Sensemaking model does not explicitly model navigation, which is an important aspect of browsing array data. Thus, we extend existing analysis models to generalize to array browsing behaviors. We found that our users alternated between three high-level analysis phases, each representing different user goals: Foraging, Sensemaking, and Navigation. Within each phase, users employed a number of low-level behaviors to achieve their goals. We model the low-level behaviors separately in the bottom half of our prediction engine, which we describe in detail in Section 4.2.

In the Foraging phase, the user is looking for visually interesting patterns in the data and forming hypotheses. The user will tend to stay at coarser zoom levels during this phase, because these levels allow the user to scan large sections of the dataset for visual patterns that she may want to investigate further. In the Sensemaking phase, the user has identified a region of interest (or ROI), and is looking to confirm an initial hypothesis. During this phase, the user stays at more detailed zoom levels, and analyzes neighboring tiles to determine if the pattern in the data supports or refutes her hypothesis. Finally, during the Navigation phase, the user performs zooming operations to move between the coarse zoom levels of the Foraging phase and detailed zoom levels of the Sensemaking phase.

4.1.2 Predicting the Current Analysis Phase

The top half of our two-level scheme is responsible for predicting the user's current analysis phase. To do this, we apply a Support Vector Machine (SVM) classifier using the LibSVM Java Library[1], similar to the work by Brown *et al.* [3]. SVM's are a group of supervised learning techniques that are frequently used for classification and regression tasks. We used a multi-class SVM classifier with a non-linear kernel (RBF). To format data for our SVM classifier, we generate a feature vector from incoming user requests, which contain the location of the requested tile (XY-coordinate and zoom level) and whether the previous request was a zoom in, zoom out, or pan. Because this SVM classifier only learns from interaction data and relative tile positions, we can apply our classification techniques to any dataset that is amenable to a tile-based format. To build a training dataset for the classifier, we hand-labeled tile requests from our user study with their corresponding analysis phase. We evaluate the accuracy of this classifier in Section 5.

4.2 Bottom-Level Design

Once the user's current analysis phase has been identified, ForeCache employs the corresponding recommendation model(s) to predict specific tiles. At the bottom level of our prediction engine, we run multiple recommendation models in parallel, where each model is designed to predict specific browsing behaviors. Each of these models requires access to the user's tile request history, and some additional metadata.

These low-level recommendation models can be categorized into two types of predictions: (a) learning what to predict from the user's previous moves (*e.g.,* pans and zooms), and (b) learning what to predict by using data characteristics, or *signatures*, from the tiles that the user has recently analyzed (*e.g.,* histograms).

4.2.1 Additional Inputs for Recommendation Models

All of our models require additional inputs in order to make predictions. Here we provide an overview of the inputs that we pass to our models, and the output produced by our models.

Each model outputs a ranked list of tiles, where the ranking signifies the model's prediction of how likely the user will request each

[1]https://github.com/cjlin1/libsvm

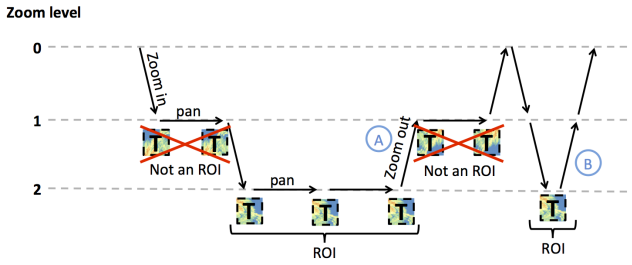


Figure 6: An example of how we retrieve the user's last ROI.

tile. Each model also requires two additional information sources, or inputs, to make predictions. The first input is a subset of tiles from the user's request history; the second input is a list of candidate tiles for the models to rank and return as output.

We use the first input as a reference point for what the user's current interests, which we refer to as a *region of interest* (ROI). We use a simple heuristic to identify the user's last ROI. We use zooming in as a cue for the user preparing to analyze a region, and zooming out as a sign that the user is finished analyzing a region. Thus, our heuristic simply tracks when the user zooms out, and records the list of tiles the user analyzed *before* zooming out. Figure 6 shows an example of how to use this heuristic. When the user first zooms out from zoom level two to zoom level one (A), the first ROI contains three tiles from zoom level two: the tile the user zoomed out from, and two other tiles the user explored after zooming in. When the user zooms out from zoom level one to zoom level zero, she is still leaving the previous ROI. Thus, we ignore this zoom out. The user then explores a new ROI by zooming into level two again. When the user zooms out (B), we find that one tile was explored since the last zoom in. Thus this ROI only has one tile. Note that the first two moves (a zoom in to level zero, and directional pan) are not considered for ROI's, as the user is still moving towards her next ROI at zoom level two.

For the second input to our models, we compile a list of candidate data tiles by finding all tiles that are within d actions from the user's current focal point. We call d the user's current "neighborhood." For example, a neighborhood of size one represents all tiles that are exactly one action away from the user's last tile request. All tiles reachable by one panning action would be part of this neighborhood. Note that we use Manhattan distance and not Euclidean distance, which is equivalent to counting the number of moves required to go from one tile to another.

4.2.2 Actions-Based (AB) Recommender

As the user moves to or from ROI's, she is likely to consistently zoom or pan in a predictable way (*e.g.,* zoom out three times). Doshi *et al.* leverage this assumption in their Momentum model, which predicts that the user's next move will match her previous move [7]. Similarly, Lee *et al.* [11] apply Markov chains to map tiles. We expand on this idea with our actions-based (AB) recommender, which builds a Markov chain from users' past actions. This differs significantly from our signature-based mechanism, because we do not need to compute any data-specific features. We use the user's last $n-1$ moves as a reference point.

A basic Markov chain contains a single state for each move the user can make (*e.g.,* zoom out, pan up, pan down, etc.). These states represent the last move the user has made. Each state also has an outgoing transition to all other states. These transitions represent how likely the user is to make the corresponding move, given the previous move. Figure 7 is an example of a Markov chain that predicts whether the user will zoom in or zoom out. Initially, zooming in and out are equally likely, because they both have probability

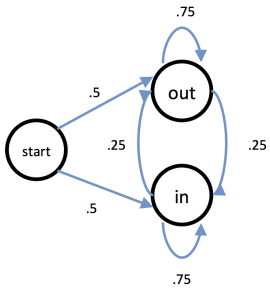


Figure 7: An example Markov chain for predicting zooms.

Table 1: Features computed in ForeCache to compare data tiles for visual similarity.

Signature	Measures used for Comparison
Normal Distribution	Mean, standard deviation
1-D histogram	histogram bins
SIFT	histogram built from clustered SIFT descriptors
DenseSIFT	same as SIFT

0.5. However, once the user has moved, she is more likely to repeat the same move than to change direction, shown by the self-loops in Figure 7 with probability 0.75.

We build more powerful n^{th}-order Markov chains in our AB recommender by embedding more of the user's movement history into each state in the Markov model. Specifically, we create a state in the Markov model for each possible sequence of the user's last n moves.

We learn transition probabilities using tile request logs recorded from our user study, which we describe in more detail in Section 5. However, if our AB recommender incorporates too much information from past logs, we will over-fit the model to our training data. To assess the tradeoff between flexibility and accuracy, we tested building n^{th}-order Markov chains with history lengths ranging from the user's last request to the user's last ten requests. We observed that setting $n = 3$ provided the best prediction accuracy. We discuss the accuracy of our recommendation models in Section 5.

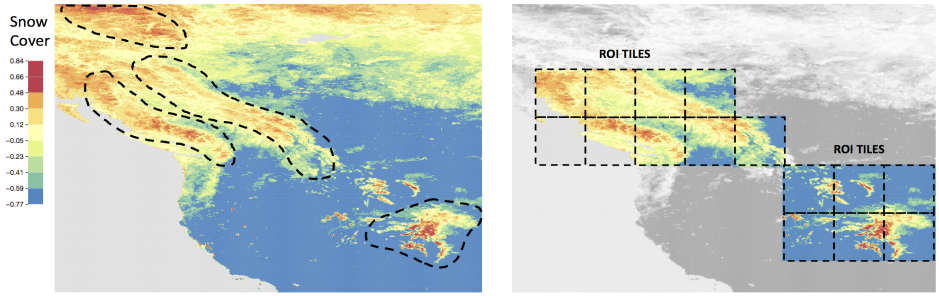
We used the BerkeleyLM [17] Java library to implement our Markov chains.

4.2.3 Signature-Based (SB) Recommender

The goal of our signature-based (SB) recommender is to identify neighboring tiles that are visually similar to what the user has seen in the past. For example, in the Foraging phase, the user is using a coarse view of the data to find new ROI's to explore. When the user finds a new ROI, she zooms into this area until she reaches her desired zoom level. Each tile along her zooming path will share the same visual features, which the user depends on to navigate to her destination. In the Sensemaking phase, the user is analyzing visually similar data tiles at the same zoom level. One such example is when the user is exploring satellite imagery of the earth, and panning to tiles within the same mountain range.

Consider Figure 8a, where the user is exploring snow cover data derived from a satellite imagery dataset. Snow is colored red, and regions without snow are blue. Thus, the user will search for ROI's that contain large clusters of red pixels, which are circled in Figure 8a. These ROI's correspond to mountain ranges.

Given the user's last ROI (*i.e.,* the last mountain range the user visited), we can look for neighboring tiles that look similar (*i.e.,* find more mountains). Figure 8b is an example of some tiles that may be in the user's history if she has recently explored some of



(a) **Potential ROI's (*e.g.*, mountain ranges) for snow cover data in the US and Canada. Snow is in red.**

(b) **Example tiles that may be in the user's request history after visiting some ROI's from (a).**

Figure 8: Example ROI's in the US and Canada for snow cover data. Note that (a) and (b) span the same latitude-longitude range.

Algorithm 1 Computes the visual distance of each candidate tile, with respect to a given ROI.

Input: Signatures S_1-S_4, candidate tiles, ROI tiles
Output: A set of distance values D

1: **for** Signature S_i, $i = 1-4$ **do**
2: $d_{i,MAX} \leftarrow 1$
3: **for** each candidate tile T_A **do**
4: Retrieve signature $S_i(T_A)$
5: **for** each ROI tile T_B **do**
6: Retrieve signature $S_i(T_B)$
7: **for** each candidate/ROI pair (T_A,T_B) **do**
8: $d_{i,A,B} \leftarrow 2^{d_{manh}(T_A,T_B)-1}[dist_{S_i}(S_i(T_A),S_i(T_B))]$
9: $d_{i,MAX} \leftarrow max(d_{i,MAX},d_{i,A,B})$
10: **for** each candidate/ROI pair (T_A,T_B) **do**
11: $d_{i,A,B} \leftarrow \frac{d_{i,A,B}}{d_{i,MAX}}$
12: **for** each candidate/ROI pair (T_A,T_B) **do**
13: $d_{A,B} \leftarrow \frac{\sqrt{\sum_{S_i} w_i(d_{i,A,B})^2}}{d_{physical}(A,B)}$
14: **for** each candidate tile T_A **do**
15: $d_A \leftarrow \sum_B d_{A,B}$
return $D = \{d_1, d_2, ..., d_A, ...\}$

these ROI's, which we can use for reference to find new ROI's.

We measure visual similarity by computing a diverse set of tile *signatures*. A signature is a compact, numerical representation of a data tile, and is stored as a vector of double-precision values. Table 1 lists the four signatures we compute in ForeCache. All of our signatures are calculated over a single SciDB array attribute. The first signature in Table 1 calculates the average and standard deviation of all values stored within a single data tile. The second signature builds a histogram over these array values, using a fixed number of bins.

Given that users are interacting with rendered images of the data, we also tested two machine vision techniques as signatures: the scale-invariant feature transform (SIFT), and a variant called denseSIFT (signatures 3 and 4 in Table 1). SIFT is a computer vision algorithm that is used to detect and compare local features in an image. We used the open source machine vision library OpenCV to compute our SIFT and denseSIFT signatures[2].

Algorithm 1 outlines how we compare candidate tiles to the user's last ROI using these signatures. We first retrieve all four signatures for each candidate tile (lines 3-4). We also retrieve these four signatures for each ROI tile on lines 5-6. Then we compute how much each candidate tile (T_A) deviates from each ROI tile (T_B), with respect to each signature (lines 7-8). To do this, a distance function for the signature is applied to the candidate tile and ROI tile (denoted as $dist_{S_i}$ in Algorithm 1). Since our signatures do not automatically account for the physical distance between T_A and T_B, we apply a penalty to our signature distances based on the Manhattan distance between the tiles. Since all four of our current signatures produce histograms as output, we use the Chi-Squared distance metric as the distance function for all signatures. We then normalize the computed distance values (lines 10-11).

To produce a single distance measure for a given candidate-ROI pair, we treat the four resulting distance measures as a single vector, and compute the ℓ^2-norm of the vector (lines 12-13). To adjust how much influence each signature has on our final distance measurements, we can modify the ℓ^2-norm function to include weights for each signature. All signatures are assigned equal weight by default, but the user can update these weight parameters as necessary.

$$\ell^2_{weighted}(A,B) = \sqrt{\sum_{S_i} w_i(d_{i,A,B})^2}$$

At this point, there will be multiple distance values calculated for each candidate tile, one per ROI tile. For example, if we have four ROI tiles, then there will be four distance values calculated per candidate tile. We sum these ROI tile distances, so we have a single distance value to compare for each candidate tile (lines 14-15). We then rank the candidates by these final distance values.

Note that it is straightforward to add new signatures to the SB recommender. To add a new signature, one only needs to add: (1) an algorithm for computing the signature over a single data tile, and (2) a new distance function for comparing this signature (if the Chi-Squared distance is not applicable).

4.3 Cache Allocation Strategies

In this section, we describe the recommendation models generally associated with each analysis phase, and how we use this information to allocate space to each recommendation model in our tile cache.

In the Navigation phase, the user is zooming and panning in order to transition between the Foraging and Sensemaking phases. Thus, we expect the AB recommendation model to be most effective for predicting tiles for this phase, and allocate all available cache space to this model.

In the Sensemaking phase, the user is mainly panning to neighboring tiles with similar visual features. Therefore, we expect the SB recommendation model to perform well when predicting tiles for this phase, and allocate all available cache space to this model.

In the Foraging phase, the user is using visual features as cues for where she should zoom in next. When the user finds a ROI that she wants to analyze, the tiles she zooms into to reach this ROI will share the same visual properties. Thus, the SB model should prove

[2]http://opencv.org

useful for this phase. However, the user will also zoom out several times in a row in order to return to the Foraging phase, exhibiting a predictable pattern that can be utilized by the AB model. Therefore, we allocate equal amounts of space to both models for this phase.

5. EXPERIMENTS

Although the goal behind ForeCache is to reduce user wait times, we will demonstrate in Section 5.5 that there is a linear (constant factor) correlation between latency and the accuracy of the prediction algorithm. As such, we claim that we can improve the observed latency in ForeCache by reducing the number of prediction errors that occur when prefetching tiles ahead of the user. Our aim in this section is to show that ForeCache provides significantly better prediction accuracy, and thus lower latency, when compared to existing prefetching techniques.

We validate our claims about user exploration behavior through a user study on NASA MODIS satellite imagery data, and evaluate the prediction accuracy of our two-level prediction engine using traces collected from the study. To validate our hypothesis that prediction accuracy dictates the overall latency of the system, we also measured the average latency observed in ForeCache for each of our prediction techniques.

To test the accuracy of our prediction engine, we conducted three sets of evaluations. We first evaluate each prediction level separately. At the top level, we measure how accurately we can predict the user's current analysis phase. At the bottom level, we measure the overall prediction accuracy of each recommendation model, with respect to each analysis phase, and compare our individual models to existing techniques. Then we compare the accuracy of the full prediction engine to our best performing individual recommendation models, as well as existing techniques. Last, we evaluate the relationship between accuracy and latency, and compare the overall latency of our full prediction engine to existing techniques.

5.1 MODIS Dataset

The NASA MODIS is a satellite instrument that records imagery data. This data is originally recorded by NASA in a three-dimensional array, where the dimensions are latitude, longitude and time. Each array cell contains a vector of wavelength measurements, where each separate wavelength measurement is called a MODIS "band."

One use case for MODIS data is to estimate snow depths in the mountains. One well-known MODIS snow cover algorithm, which we apply in our experiments, is the Normalized Difference Snow Index (NDSI) [20]. The NDSI indicates whether there is snow at a given MODIS pixel (*i.e.,* array cell). A high NDSI value (close to 1.0) means that there is snow at the given pixel, and a low value (close to -1.0) corresponds to no snow cover. The NDSI uses two MODIS bands to calculate this. We label the two bands used in the NDSI as VIS for visible light, and SWIR for short-wave infrared. The NDSI is calculated by applying the following function to each cell of the MODIS array. It is straightforward to translate this transformation into a user-defined function (UDF) in SciDB:

$$NDSI = \frac{(visible\ light\ -\ short\ wave\ infrared)}{(visible\ light\ +\ short\ wave\ infrared)}.$$

5.1.1 Modifications for User Study

Our test dataset consisted of NDSI measurements computed over one week of raw NASA MODIS data, where the temporal range of the data was from late October to early November of 2011. We downloaded the raw data directly from the NASA MODIS website[3], and used SciDB's MODIS data loading tool to load the data into SciDB. We applied the NDSI to the raw MODIS data as a user-defined function, and stored the resulting NDSI calculations in a separate array. The NDSI array was roughly 10TB in size when stored in SciDB.

To ensure that participants could complete the study in a timely manner, we chose to simplify the dataset to make exploring it easier and faster. Prior to the study, The NDSI dataset was flattened into a single, one-week window, reducing the total dimensions from three (latitude, longitude, time) to two (latitude and longitude only). The NDSI dataset contained four numeric attributes: maximum, minimum and average NDSI values; and a land/sea mask value that was used to filter for land or ocean pixels in the dataset. After applying ForeCache's tile computation process, there were nine total zoom levels in this dataset, where each level was a separate layer of cooked tiles.

5.1.2 Calculating the NSDI in SciDB

In this section, we explain how to compute the NSDI in SciDB. We assume that we already have a an NDSI UDF written in SciDB, which we refer to as "ndsi_func".

Let S_{VIS} and S_{SWIR} be the SciDB arrays containing recorded data for their respective MODIS bands. We use two separate arrays, as this is the current schema supported by the MODIS data loader for SciDB [19]. S_{VIS} and S_{SWIR} share the same array schema. An example of this schema is provided below.

$$S_{VIS/SWIR}(\text{reflectance})[latitude, longitude].$$

The array attributes are denoted in parentheses (reflectance) and the dimensions are shown in brackets (latitude and longitude). The attributes represent the MODIS band measurements recorded for each latitude-longitude coordinate.

The following is the SciDB query we execute to compute the NDSI over the S_{VIS} and S_{SWIR} arrays:

Query 1: SciDB query to apply the NDSI.

```
1  store(
2      apply(
3          join(S_VIS, S_SWIR),
4          ndsi,
5          ndsi_func(S_VIS.reflectance,
6              S_SWIR.reflectance)
7      ),
8      NDSI
9  );
```

We first perform an equi-join, matching the latitude-longitude coordinates of the two arrays (line 3). Note that SciDB implicitly joins on dimensions, so latitude and longitude are not specified in the query. We then apply the NDSI to each pair of joined array cells by calling the "ndsi_func" UDF (lines 5-6). We pass the reflectance attribute of S_{VIS} and the reflectance attribute of S_{SWIR} to the UDF. We store the result of this query as a separate array in SciDB named *NDSI* (line 8), and the NDSI calculations are recorded in a new "ndsi" attribute in this array (line 4).

5.2 Experimental Setup

5.2.1 Hardware/Software setup

The ForeCache front-end for the study was a web-based visualizer. The D3.js Javascript library was used to render data tiles. We describe the interface in more detail below.

The data was partitioned across two servers running SciDB version 13.3. Each server had 24 cores, 47GB of memory, and 10.1 TB of disk space. Both servers ran Ubuntu Server 12.04. The

[3]http://modis.gsfc.nasa.gov/data/

first server was also responsible for running the ForeCache middleware (prediction engine and cache manager), which received tile requests from the client-side interface.

5.2.2 Measuring Accuracy

The number of cache misses (*i.e.,* prediction accuracy) directly impacts whether delays occur in the ForeCache front end, and thus also determines the length of user wait times (*i.e.,* the latency). Therefore, we used prediction accuracy as one of our primary metrics for comparing prefetching strategies, similar to the prediction accuracy metric used by Lee *et al.* [11]. To compute this, we ran our models in parallel while stepping through tile request logs, one request at a time. For each requested tile, we collected a ranked list of predictions from each of our recommendation models, and recorded whether the next tile to be requested was located within the top k of these rankings. For example, we would set $k = 3$ to test whether the next tile was found within the top three recommendations in each list. To compute the final accuracy for the trace, we counted all correct predictions for each recommender and divided these counts by the total number of requested tiles.

We simulated having limited space in our middleware cache in our experiments by varying k in our accuracy measurements. Thus measuring prediction accuracy becomes equivalent to measuring the hit rate of our tile cache. A value of $k = 1$ meant that ForeCache only had space to fetch a single tile before the user's next request, $k = 2$ represented available space to fetch two tiles, and so on. We varied k from 1 to 8 in our experiments. Note that at $k = 9$, we are fetching all nine of the user's possible next moves, and are thus guaranteed to prefetch the correct tile. This is because the user was restricted to a small set of actions in the front-end visualizer. Specifically, the user could perform the following nine moves in our interface: zoom out, pan (left, right, up, down), and zoom in (users could zoom into one of four tiles at the zoom level below).

Given that ForeCache prefetches new tiles after every request, we found that having ForeCache predict further than one move ahead did not actually improve accuracy. Therefore, predicting beyond the user's next move was irrelevant to the goals of these experiments, and we only considered the tiles that were exactly one step ahead of the user. We leave prefetching more than one step ahead of the user as future work.

5.2.3 Comparing with Existing Techniques

To compare our two-level prediction engine with existing techniques, we implemented two models proposed in [7], the "Momentum" and "Hotspot" models. Several more recent systems, such as ATLAS [6] and ImMens [15] apply very similar techniques (see Section 7 for more information).

Momentum: The Momentum model assumes that the user's next move will be the same as her previous move. To implement this, the tile matching the user's previous move is assigned a probability of 0.9, and the eight other candidates are assigned a probability of 0.0125. Note that this is a Markov model, since probabilities are assigned to future moves based on the user's previous move.

Hotspot: The Hotspot model is an extension of the Momentum model that adds awareness of popular tiles, or *hotspots*, in the dataset. To find hotspots in the NDSI dataset, we counted the number of requests made for each tile visited in our user study, and chose the tiles with the most requests. When the user is not close to any hotspots, the Hotspot model defaults to the behavior of the Momentum model. When a hotspot is nearby, the Hotspot model assigns a higher ranking to any tiles that bring the user closer to that hotspot, and a lower ranking to the remaining tiles. We trained the Hotspot model on trace data ahead of time. This training process took less than one second to complete.

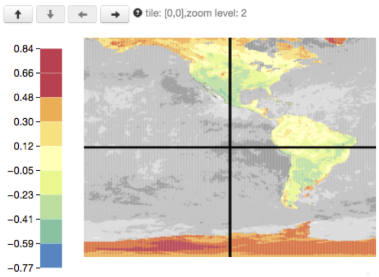


Figure 9: ForeCache browsing interface.

5.3 User Study

To generate a diverse workload for evaluating the ForeCache system, we conducted a user study with domain scientists exploring the NDSI dataset. In this section, we outline the study design, and validate whether our analysis phases are an appropriate classification of user behavior using results from the study. We avoided biasing the behavior of our study participants by caching all data tiles in main memory while the study was being conducted. This prevented our participants from choosing their their movements based on response time (*e.g.,* avoiding zooming out if it is slower than other movements). This also ensured that all study participants had the same browsing experience throughout the study.

5.3.1 Study Subjects

The study consisted of 18 domain scientists (graduate students, post doctoral researchers, and faculty). Most of our participants were either interested in or actively working with MODIS data. Subjects were recruited at the University of Washington and University of California, Santa Barbara.

5.3.2 Browsing Interface

Figure 9 is an example of the client-side interface. Each visualization in the interface represented exactly one data tile. Participants (*i.e.,* users) used directional buttons (top of Figure 9) to move up, down, left, or right. Moving up or down corresponded to moving along the latitude dimension in the NDSI dataset, and left or right to the longitude dimension. Each directional move resulted in the user moving to a completely separate data tile. User's left clicked on a quadrant to zoom into the corresponding tile, and right clicked anywhere on the visualization to zoom out.

5.3.3 Browsing Tasks

Participants completed the same search task over three different regions in the NDSI dataset. For each region, participants were asked to identify four data tiles (*i.e.,* four different visualizations) that met specific visual requirements. The tasks were as follows:

1. Find four data tiles in the continental United States at zoom level 6 with the highest NDSI values.
2. Find four data tiles within western Europe at zoom level 8 with NDSI values of .5 or greater.
3. Find 4 data tiles in South America at zoom level 6 that contain NDSI values greater than .25.

A separate request log was recorded for each user and task. Therefore, by the end of the study we had 54 user traces, each consisting of sequential tile requests.

5.3.4 Post-Study: General Observations

The most popular ROI's for each task were: the Rocky Mountains for Task 1, Swiss Alps for Task 2, and Andes Mountains for

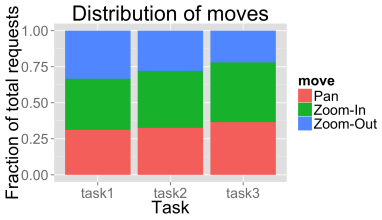


Figure 10: Distribution of moves across all users, recorded per task.

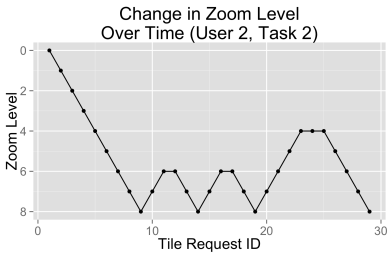


Figure 11: Change in zoom level per request as study participant 2 completed task 2.

Task 3. The average number of requests per task are as follows: 35 tiles for Task 1, 25 tiles for Task2, and 17 tiles for Task 3. The mountain ranges in Tasks 2 and 3 (Europe and South America) were closer together and had less snow than those in task 1 (US and Southern Canada). Thus, users spent less time on these tasks, shown by the decrease in total requests.

We also tracked whether the request was a zoom in, zoom out, or pan. Figure 10 shows the distribution of directions across all study participants, recorded separately for each task. We see that for all tasks, our study participants spent the most time zooming in. This is because users had to zoom to a specific zoom level for each task, and did not have to zoom back out to the top level to complete the task. In tasks 1 and 2, users panned and zoomed out roughly equally. In task 3, we found that users clearly favored panning more than zooming out.

5.3.5 *Evaluating the Suitability of Our Three Analysis Phases*

To demonstrate some of the patterns that we found in our user traces, consider Figure 11, which plots the change in zoom level over time for one of our user traces. The coarsest zoom level is plotted at the top of Figure 11, and the most-detailed zoom level was plotted at the bottom. The x-axis represents each successive tile request made by this user. A downward slope corresponds to the user moving from a coarser to a more detailed zoom level; an upward slope corresponds to the reverse; and a flat line (*i.e.,* slope of 0) to the user panning to tiles at the same zoom level.

We see that the user alternates between zooming out to a coarser zoom level, and zooming into more detailed zoom levels. We know that the coarser views were used to locate snow, and the high-resolution views to find specific tiles that satisfied task 2 (hence the four tile requests specifically at zoom level 8).

We see in Figure 11 that this user's behavior corresponds directly to the three exploration phases described in Section 4.1.1. The user's return to coarser views near the top of Figure 11 correspond to the user returning to the Foraging phase (*e.g.,* at request ID's 20 to 23). The user's zooms down to the bottom half of the plot correspond to the user moving to the Sensemaking phase, as they searched for individual tiles to complete the task requirements. Furthermore, we found that 13 out of 18 users exhibited this same general exploration behavior throughout their participation in the

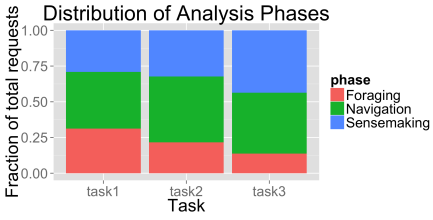


Figure 12: Distribution of Analysis phases for each task.

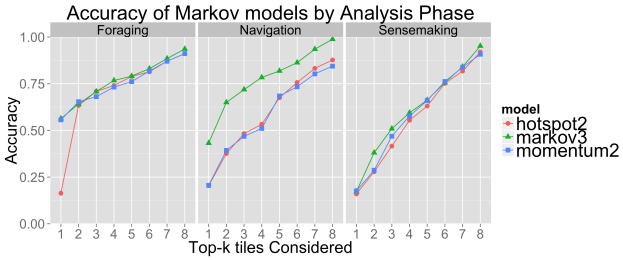


Figure 13: Accuracy of our best AB model (Markov3) compared to existing models.

study. 16 out of 18 users exhibited this behavior during 2 or more tasks. Furthermore, we found that only 57 out of the 1390 total requests made in the study were not described adequately by our exploration model.

Therefore, we conclude that our three analysis phases provide an accurate classification of how the vast majority of users actually explored our NDSI MODIS dataset.

5.4 Evaluating the Prediction Engine

Now that we have established that our three analysis phases provide a comprehensive labeling scheme for user behavior, we move on to evaluating our two-level prediction engine. In particular, we evaluated each level of our prediction engine separately, and then compared the complete prediction engine to the existing techniques described in our experimental setup.

At the top level of our prediction engine, we measured how accurately we could predict the user's current exploration phase. At the bottom level, we measured the accuracy of each recommendation model, with respect to each analysis phase.

The following experiments were conducted using *leave-one-out cross validation* [10], a common cross-validation technique used in the HCI community for evaluating user study data. For each user, the models were trained on the trace data of the other 17 out of 18 participants, and tested on the trace data from the remaining participant that was removed from the training set.

5.4.1 *Predicting the User's Current Analysis Phase*

The goal was to measure how accurately we could predict the user's current analysis phase. To build a training and testing set for

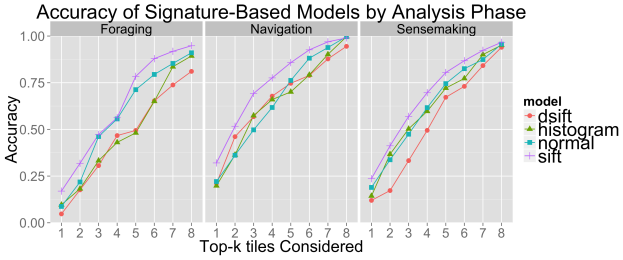


Figure 14: Accuracy of the four signatures in our SB model.

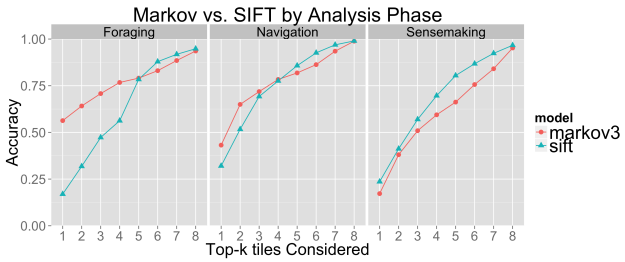


Figure 15: Accuracy of our top two recommendation models.

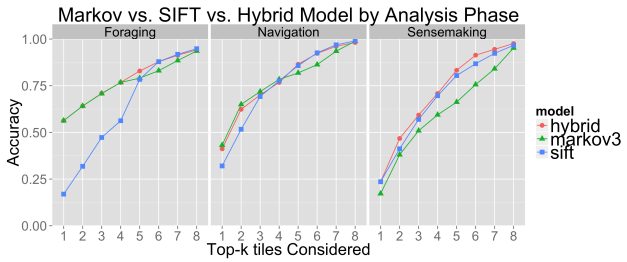


Figure 16: Accuracy of our final prediction engine (hybrid) compared to our best individual models.

this experiment, we manually labeled each request in our request logs with one of our 3 analysis phases. Figure 12 shows the distribution of phase labels. We see that users spent noticeably less time in the Foraging phase for tasks 2 and 3 (*i.e.,* looking for new ROI's), which is consistent with our user study observations.

To test our SVM classifier, we performed leave-one-out cross validation (see above), where all requests for the corresponding user were placed in the test dataset, and the remaining requests were placed in the training set. Training the classifier took less than one second. We found that our overall accuracy across all users was 82%. For some users, we could predict the current analysis phase with 90% accuracy or higher.

5.4.2 *Accuracy of Recommendation Models*

In order to validate the accuracy of our individual recommenders, we conducted two sets of experiments, where we (1) compared the accuracy of our AB recommender to existing techniques, and (2) measured the prediction accuracy of our SB recommender separately for each of our four tile signatures.

The goal of these experiments was two-fold. First, we wanted to find the phases where existing techniques performed well, and where there was room for improvement. Second, we wanted to evaluate whether our new recommendation models excelled in prediction accuracy for their intended analysis phases.

To do this, we evaluated how accurately our individual recommendation models could predict the user's next move, for each analysis phase.

Action-Based (AB) Model To evaluate the impact of history length on our AB recommender, we implemented a separate Markov chain for $n = 2$ to $n = 10$. Each Markov chain model only took milliseconds to train. We found that a history length of 2 was too small, and resulted in worse accuracy compared to the other history lengths we tried. Otherwise, we found that there was no significant improvement in accuracy when we increased n beyond 3, and thus found $n = 3$ to be most efficient for our AB model.

Figure 13 shows the prediction accuracy of our AB model compared to the Momentum and Hotspot models, with increasing values of k. Note that k represents the total space (in tiles) that each model was given for predictions (see Section 5.2.2 for more information). In Figure 13, we see that for the Foraging and Sensemaking phases, our AB model matches the performance of existing techniques for all values of k. Furthermore, we found that our AB model achieves significantly higher accuracy during the Navigation phase for all values of k. This validates our decision to use the AB model as the primary model for predicting the Navigation phase.

Signature-Based (SB) Model: Figure 14 shows the accuracy of each of our individual signatures, with respect to analysis phase. To do this, we created four separate recommendation models, one per signature. Amongst our signatures, we found that the SIFT signature achieved better accuracy overall compared to the other signatures. We expected a machine vision feature like SIFT to perform well, because users are essentially comparing images when they analyze MODIS tiles.

Surprisingly, we found that the denseSIFT signature did not perform nearly as well as SIFT. We attribute this to the fact that denseSIFT effectively matches an entire image, where the original SIFT only matches small regions of an image. In our snow cover use case, relevant visualizations will contain clusters of snow pixels (*i.e.,* red pixels, like in the examples in Figure 8), but will not look similar otherwise. For example, the Rockies will look very different from the Andes, but they will both contain clusters of snow (red) pixels. Thus, many relevant tiles will not appear to be similar with regards to the denseSIFT signature.

Comparing the Best AB and SB Models: We took the best AB model (Markov chains with history length 3, labeled as "Markov3") and best SB model (SIFT), and compared their accuracy for each analysis phase. We present the results in Figure 15.

For the Foraging phase, we found that the AB model had significantly higher accuracy than our SB model when $k < 5$, with more than a 25% difference in accuracy for $k < 3$. This is because users shared the same predictable pattern of zooming out several times to return to the Foraging phase. As a result, our AB model was able to learn these pre-Foraging patterns and predict accordingly.

Similarly, the AB model had noticeably better performance during the Navigation phase when $k < 4$, with greater than 10% difference in accuracy when $k = 2$. However, we see that the SB model eventually surpasses our AB model for higher values of k. We attribute this to the fact that SB model has the ability to successfully identify visual features at other zoom levels.

For the Sensemaking phase, we found that the SB model consistently outperformed our AB model. This was expected, as our SB models were designed with the Sensemaking phase in mind.

5.4.3 *Evaluating the Final Prediction Engine*

We used the accuracy results for our phase predictor and best individual recommendation models as inputs to our final two-level prediction engine. Our prediction engine only incorporated two recommenders, the Markov3 AB recommender and the SIFT SB recommender, re-allocating space for each model in the middleware cache based on the user's predicted analysis phase. When the Sensemaking phase is predicted, our model always fetches predictions from our SB model only. Otherwise, our final model fetches the first 4 predictions from the AB model (or less if $k < 4$), and then starts fetching predictions from the SB model if $k > 4$.

Note that we updated our original allocation strategies based on our observed accuracy results for the best SB model and the AB model. Originally, all space was allocated to our AB model during the Navigation phase. However, we found that the SB model eventually outperformed the AB model when $k > 4$, and thus we now allocate space to both models when $k > 4$.

We compared the accuracy of the final prediction engine to our best individual recommenders, shown in Figure 16. Figure 16 shows that our final prediction engine successfully combined the strengths

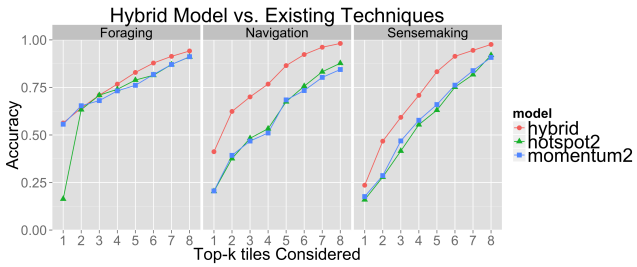


Figure 17: Accuracy of the hybrid model compared to existing techniques.

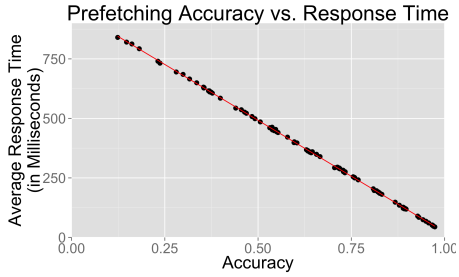


Figure 18: Plot of average response time given prefetch accuracy, for all models and fetch sizes (linear regression: Adj R^2=0.99985, Intercept=961.33, Slope=-939.08, P=1.1704e-242.

of our two best prediction models. It was able to match the accuracy of the best recommender for each analysis phase, resulting in better overall accuracy than any individual recommendation model. We also compared our final prediction engine to existing techniques, shown in Figure 17. We see that for the Foraging phase, our prediction engine performs as well (if not better) than existing techniques. For the Navigation phase, we can achieve up to 25% better prediction accuracy compared to Momentum and Hotspot. Similarly for the Sensemaking phase, we see a consistent 10-18% improvement in accuracy using our hybrid strategy.

5.5 Latency

We used the same setup from our accuracy experiments to also measure latency. To measure the latency for each tile request, we recorded the time at which the client sent the request to the middleware, as well as the time at which the requested tile was received by the client. We calculated the resulting latency by taking the difference between the two time measurements. On a cache hit, the middleware was able to retrieve the tile from main memory, allowing ForeCache to send an immediate response. On a cache miss, the middleware was forced to issue a query to SciDB to retrieve the missing tile, which was much slower. On average, the middleware took 19.5 ms to send tiles for a cache hit, and 984.0 ms for a cache miss.

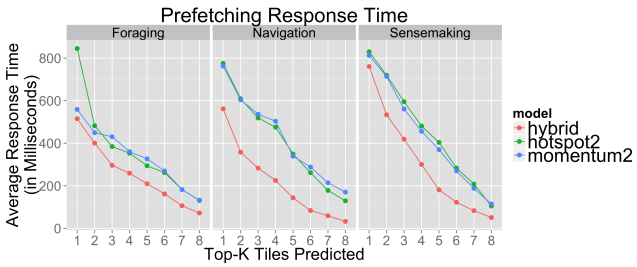


Figure 19: Average prefetching response times for hybrid model and existing techniques.

To evaluate our claim that accuracy dictates latency, we plotted the relationship between prefetching accuracy and average response time (*i.e.,* average latency), provided in Figure 18. Accuracy and response times were plotted for all models and fetch sizes. We see a strong linear relationship between accuracy and response time, where a 1% increase in accuracy corresponded to a 10ms decrease in average response time (adjusted R^2 = 0.99985). Given this constant accuracy-latency factor, we found that the higher prediction accuracy of our hybrid algorithm translates to a time savings of 150-250ms per tile request, when compared with existing prefetching techniques. The difference in latency is plotted in Figure 19, where we calculated the average response times for three models. Furthermore, we found that our hybrid model reduced response times by more than 50% for $k \geq 5$, compared with existing techniques.

This latency evaluation indicates that ForeCache provides significantly better performance over not only traditional systems (*i.e.,* exploration systems without prefetching), but also existing systems that enable prefetching (*e.g.,* [7, 6, 15]). Specifically, as shown in Figure 19, with a prefetch size of 5 tiles ($k = 5$), our system demonstrates a 430% improvement over traditional systems (*i.e.,* average latency of 185ms vs. 984ms), and 88% over existing prefetching techniques (average latency of 185 ms vs. 349 ms for Momentum, and 360 ms for Hotspot).

In addition, ForeCache provides a much more fluid and interactive user experience than traditional (no prefetching) systems. As shown in HCI literature, a 1 second interaction delay is at the limit of a user's sensory memory [4]. Delays greater than 1 second make users feel like they cannot navigate the interface freely [16, 14]. In this regard, traditional systems (*i.e.,* a constant latency of 1 second per request) are not considered interactive by HCI standards.

In contrast, ForeCache remains highly interactive during most of the user's interactions with the interface, with only 19.5ms of delay per tile request. As shown in Figure 17, with a fetch size of 5 tiles ($k = 5$), the prediction algorithm succeeds the vast majority of the time (82% of the time), making a cache miss (and the full 1 second delay) an infrequent event. Thus our techniques allow systems with limited main memory resources (*e.g.,* less than 10MB of prefetching space per user) to operate at interactive speeds, so that many users can actively navigate the data freely and in parallel.

6. DISCUSSION AND FUTURE WORK

While ForeCache has demonstrated a significant advantage over existing systems in both latency and user experience, there is still room for improvement. In this section, we discuss an extension of ForeCache by integrating the *pre-computation* of tiles during runtime. In addition, we present several venues for further improvements to ForeCache both in performance and applicability to other large-scale scientific datasets.

6.1 Pre-computation

When disk space is also a limited resource, it may be worthwhile to favor resource-saving optimizations over costly optimizations with lower latency. One immediate way to save resources in ForeCache is to pre-compute tiles at runtime, instead of computing every tile offline. As new predictions come in, we compute as many top-ranked tiles as we can during the user's think time. We then choose which of these tiles to copy to main memory.

However, it is considerably more complicated to evaluate the latency benefits of this technique, and thus to compare the resource-latency tradeoff. In particular, the time required to compute tiles can vary widely, based on the underlying DBMS configuration (de-

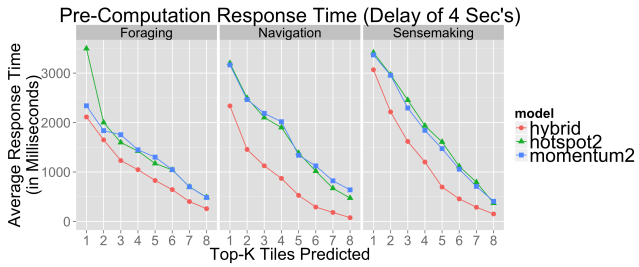


Figure 20: Average pre-computation response times comparison for several models.

gree of parallelism, use of SSD's, etc.). To test this, we ran ForeCache with two different configurations of SciDB, and found that computing tiles could range from 33 seconds on average (very basic, single-threaded setup) to 3 seconds (optimized, multithreaded setup). Consider the 3 second case. Pre-computing *and* prefetching will take 4 seconds for 1 tile (3 seconds to compute, then 1 second to fetch), 8 seconds for 2 tiles, etc. In Figure 20, we calculated the average latency for this strategy, using the exact same prediction setup as our prefetching case. We see that our hybrid model generally saves about 1 second of latency compared to existing models (including the 200ms we save from prefetching). We also see that for just 16 seconds of pre-computation and prefetching time (*i.e.,* $k = 4$), the hybrid model takes close to 1 second to return tiles, whereas the other models take 1.5 to 2 seconds. Furthermore, existing techniques require at least 8 more seconds of pre-computation and prefetching time to match the performance of the hybrid model.

Unfortunately, in the event of a cache miss, this strategy could lead to a worst case scenario of a 4 second wait time (compared to the 1 second latency, when all tiles were computed offline). Due to the potentially high cost of a cache miss, an important future work for this project will be to develop a balanced approach that can take advantage of the resource savings provided by pre-computation, without incurring the high cost of cache misses.

6.2 Future Work

In this section, we describe three future directions for the ForeCache system: (1) applying our tiling scheme to other dataset types; (2) extending our prediction engine to work well with high dimension datasets; and (3) creating new prediction and caching techniques to take advantage of multi-user environments.

Our proposed tiling scheme works well for any array-based dataset. However, when considering other types of data (*e.g.,* social graphs or patient health records), it is unclear how to map these datasets to tiles. We plan to develop a general-purpose tiling mechanism for relational datasets.

ForeCache works well with low-dimension datasets. However, the number of tiles grows exponentially with the number of dimensions (*i.e.,* the possible moves in the client-side interface), making it harder to predict which direction users will explore next. One solution is to allow users to "pivot" along dimensions, where the user chooses two or three dimensions to explore at any given time. Pivoting limits the user's range of possible actions, enabling ForeCache to still make predictions.

Our prediction framework does not currently take into account potential optimizations within a multi-user scheme. For example, it is unclear how to partition the middleware cache to make predictions for multiple users exploring different datasets, or how to share data between users exploring the same dataset. We plan to extend our architecture to manage coordinated predictions and caching across multiple users.

7. RELATED WORK

Many systems use online computation, pre-computation or prefetching techniques to support exploratory browsing of large datasets. The ScalaR system [2] computes aggregates and samples in the DBMS at runtime to avoid overwhelming the client with data. However, aggregation and sampling queries are too slow to complete at interactive speeds. Fisher *et al.* [8] combine online sampling with running summaries to improve response times. However, users must still wait for queries to complete if they want accurate results.

Several systems instead build pre-computed data reductions to speed up query execution. BlinkDB [1] builds stratified samples over datasets as they are loaded into the DBMS, and provides error and latency guarantees when querying these samples. The DICE system [9] utilizes pre-computed samples and a distributed DBMS environment to shorten execution times for date cube queries. The techniques used in ForeCache can be used alongside sampling techniques to further improve performance.

Two recent systems have created specialized data structures to support data exploration. Lins *et al.* [13] developed new data cube structures, which they call nanocubes, to efficiently visualize large datasets. However, the entire nanocubes data structure must fit in memory. Thus, nanocubes do not scale easily to larger datasets. Similar to our approach, the imMens system [15] builds data tiles in advance, and fetches these tiles from the backend at runtime. However, users must build the data tiles manually, and imMens does not utilize comprehensive prediction techniques to fetch tiles.

A number of systems use prediction algorithms to improve prefetching accuracy. Lee *et al.* [11] and Cetintemel *et al.* [5] propose using Markov chains to predict user movements. Chan *et al.* [6] and Doshi *et al.* [7] also propose several prediction strategies–the most sophisticated being Markov chains–to predict the user's next requests. We compare with two of these strategies, Momentum and Hotspot, in our experiments. Yesilmurat *et al.* [22] propose techniques similar to the Momentum model for prefetching geospatial data. Li *et al.* [12] combine the Hotspot model with Markov chains to boost prefetching performance. ForeCache builds on the techniques presented in these systems with new signature-based prediction algorithms and a novel adaptive framework for running multiple algorithms in parallel.

8. CONCLUSION

In this paper, we presented ForeCache, a general-purpose exploration system for browsing large datasets. ForeCache employs a client-server architecture, where the user interacts with a lightweight client-side interface, and a the data to be explored is retrieved from a back-end server running a DBMS. We inserted a middleware optimization layer in front of the DBMS that uses a two-level prediction engine and main-memory cache to prefetch relevant data tiles given the user's current analysis phase and recent tile requests. We presented results from a user study we conducted, where 18 domain experts used ForeCache to explore NASA MODIS satellite imagery data. We tested the performance of ForeCache using traces recorded from our user study, and presented accuracy results showing that our prediction engine provides: (1) significant accuracy improvements over existing prediction techniques (up to 25% higher accuracy); and (2) dramatic latency improvements over current non-prefetching systems (430% improvement in latency), and existing prediction techniques (88% improvement in latency).

9. REFERENCES

[1] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *Proc. EuroSys 2013*, pages 29–42, New York, NY, USA, 2013. ACM.

[2] L. Battle, M. Stonebraker, and R. Chang. Dynamic reduction of query result sets for interactive visualizaton. In *IEEE BigDataVis Workshop*, pages 1–8, 2013.

[3] E. Brown, A. Ottley, H. Zhao, Q. Lin, R. Souvenir, A. Endert, and R. Chang. Finding Waldo: Learning about Users from their Interactions. *IEEE TVCG*, 20(12):1663–1672, Dec. 2014.

[4] S. K. Card, G. G. Robertson, and J. D. Mackinlay. The Information Visualizer, an Information Workspace. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '91, pages 181–186, New York, NY, USA, 1991. ACM.

[5] U. Cetintemel, M. Cherniack, J. DeBrabant, Y. Diao, K. Dimitriadou, A. Kalinin, O. Papaemmanouil, and S. B. Zdonik. Query steering for interactive data exploration. In *CIDR*, 2013.

[6] S.-M. Chan, L. Xiao, J. Gerth, and P. Hanrahan. Maintaining interactivity while exploring massive time series. In *VAST*, 2008.

[7] P. Doshi, E. Rundensteiner, and M. Ward. Prefetching for visual data exploration. In *Proc. DASFAA*, 2003.

[8] D. Fisher. Incremental, approximate database queries and uncertainty for exploratory visualization. In *LDAV*, 2011.

[9] N. Kamat, P. Jayachandran, K. Tunga, and A. Nandi. Distributed interactive cube exploration. *ICDE*, 2014.

[10] R. Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145, 1995.

[11] D. H. Lee, J. S. Kim, S. D. Kim, K.-C. Kim, Y.-S. Kim, and J. Park. Adaptation of a Neighbor Selection Markov Chain for Prefetching Tiled Web GIS Data. ADVIS '02, pages 213–222, London, UK, UK, 2002. Springer-Verlag.

[12] R. Li, R. Guo, Z. Xu, and W. Feng. A Prefetching Model Based on Access Popularity for Geospatial Data in a Cluster-based Caching System. *Int. J. Geogr. Inf. Sci.*, 26(10):1831–1844, Oct. 2012.

[13] L. Lins, J. Klosowski, and C. Scheidegger. Nanocubes for real-time exploration of spatiotemporal datasets. *IEEE TVCG*, 2013.

[14] Z. Liu and J. Heer. The Effects of Interactive Latency on Exploratory Visual Analysis. *IEEE TVCG*, 20(12):2122–2131, Dec. 2014.

[15] Z. Liu, B. Jiang, and J. Heer. immens: Real-time visual querying of big data. *Proc. EuroVis*, 32, 2013.

[16] J. Nielsen. Powers of 10: Time Scales in User Experience, Oct. 2009.

[17] A. Pauls and D. Klein. Faster and smaller n-gram language models. HLT, pages 258–267, Stroudsburg, PA, USA, 2011.

[18] P. Pirolli and S. Card. The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis. In *Proc. International Conference on Intelligence Analysis*, volume 2005, pages 2–4, 2005.

[19] G. Planthaber, M. Stonebraker, and J. Frew. Earthdb: Scalable analysis of modis data using scidb. In *BigSpatial*, pages 11–19, New York, NY, USA. ACM.

[20] K. Rittger, T. H. Painter, and J. Dozier. Assessment of methods for mapping snow cover from modis. *Advances in Water Resources*, 51(0):367 – 380, 2013.

[21] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman. The architecture of scidb. In *SSDBM*, pages 1–16. Springer, 2011.

[22] S. Yesilmurat and V. Isler. Retrospective Adaptive Prefetching for Interactive Web GIS Applications. *Geoinformatica*, 16(3):435–466, July 2012.