Dynamic Prefetching of Data Tiles for Interactive Visualization

Leilani Battle
MIT
leilani@csail.mit.edu

Remco Chang
Tufts University
remco@cs.tufts.edu

Michael Stonebraker
MIT
stonebraker@csail.mit.edu

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 [17, 15]. 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, although modern database management systems (DBMS's) allow users to perform complex scientific analyses over large datasets [20], 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.

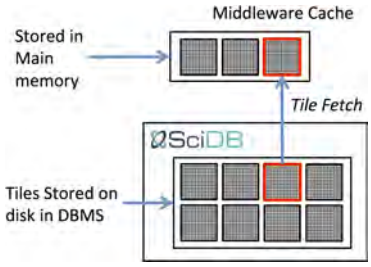


Figure 1: A diagram of ForeCache's tile storage scheme.

In this paper, we present ForeCache, a general-purpose tool for interactive browsing of large datasets. 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 [23], 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, complex scientific analyses take time, and may not execute at interactive speeds in the DBMS. 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* [16].

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

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

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

© 2016 ACM. ISBN 978-1-4503-3531-7/16/06...$15.00

DOI: http://dx.doi.org/10.1145/2882903.2882919

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. Furthermore, we may only have time to fetch a small number of tiles before the user's next request. 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, 8]. 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 analysis goals, 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 high-level analysis goals.

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 *analysis phase (*i.e., *her current frame of mind)*, given her most recent actions. The user's analysis phase hints at her current analysis goals, and thus provides context for which actions in the interface she may use to reach her goals. We provide examples of analysis phases in the following paragraph. Furthermore, users frequently employ several low-level browsing patterns within each analysis phase (*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 pattern. Using this two-level design, our prediction engine tracks changes in the user's current analysis phase, and updates its prediction strategy accordingly. To do this, we increase or decrease the space allotted to each low-level recommendation model for predictions.

Taking inspiration from other user analysis models [19], we have observed that the space of user interaction patterns 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.

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?) [8]; and (b) using data-derived characteristics, or *signatures*, to identify neighboring tiles that are similar to what the user has requested 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 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 enabling interactive exploration 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 interaction patterns.
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.

1.1 Background

ForeCache relies on a diverse set of prediction components and user inputs. Here, we provide an overview of the main concepts utilized in ForeCache, some of which were introduced in Section 1.

User Interactions/Moves: The user's interactions with ForeCache are the actions she makes in the front-end interface to explore her data. We also refer to these interactions as *moves*.

User Session: A user session refers to a single session for which the user has logged into ForeCache and explored a single dataset.

Data Model: The ForeCache data model defines: (1) the structure and layout of data tiles, and (2) how to build data tiles. We explain the ForeCache data model in detail in Section 2.

Analysis Model: Our analysis model is defined by our three analysis phases, and how these phases interact with each other. We explain our analysis model in detail in Section 4.2.

Analysis Phase: The user's current analysis phase represents her frame of mind while exploring data in ForeCache (*i.e.,* Foraging, Navigation, or Sensemaking). Analysis phases can be inferred through the user's interactions; we explain how we predict analysis phases in Section 4.2.2.

Browsing Patterns: Low-level browsing patterns are short chains of interactions repeated by the user (*e.g.,* zooming in three times). We explain how we predict these patterns in Section 4.3.

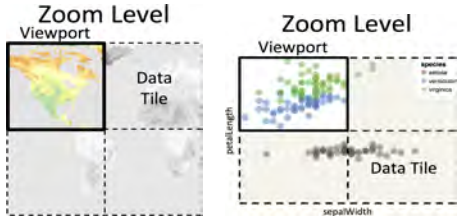
Recommendation Model: A recommendation model is an algorithm used to predict low-level browsing patterns (*e.g.,* Markov chains). ForeCache employs two kinds of recommendation models: Action-Based (Section 4.3.2) and Signature-Based (Section 4.3.3).

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

The datasets that work best with ForeCache share the same properties that make SciDB performant: (a) the majority of column types are numerical (integers, floats, etc.), and (b) the relative position of points within these columns matters (*e.g.,* comparing points in time, or in latitude-longitude position). These properties ensure that the underlying arrays are straightforward to aggregate, partition, and visualize in ForeCache. The following three example datasets share these properties: geospatial data (*e.g.,* satellite im-



(a) **Satellite Imagery** (b) **Multidimensional**

(c) **Timeseries (Heart rate Monitoring)**

Figure 2: Potential tiling schemes for three types of data.

agery in Figure 2a), multidimensional data (*e.g.,* iris flower classification in Figure 2b), and time series data (*e.g.,* heart rate monitoring in Figure 2c). Beyond these three examples, SciDB has also been used for efficiently analyzing genomics data [24] and astronomy data [22]. Given its extensive support for complex analytics over multidimensional datasets, we use SciDB as the back-end DBMS in ForeCache.

Consider Figure 2a, where the user is exploring an array of snow cover measurements computed from satellite imagery. Each array cell has been mapped to a pixel, where orange and yellow pixels correspond to snow. 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; the user can move to other tiles by panning in the client-side interface. The user can also zoom in or out to explore different zoom levels.

2.2 Interactions Supported by ForeCache

In this paper, we focus on supporting data exploration through two-dimensional (2D) views, where exploration means that the user can browse, but not modify the underlying dataset. In addition, we assume that users are interacting with the data using consistent, incremental actions that only retrieve a fraction of the underlying dataset. For example, if the user wants to go from zoom level 0 to 4 in ForeCache, she must go through levels 1, 2, and 3 first. Otherwise, users are essentially performing random accesses on the underlying data, which are generally difficult to optimize for any back-end DBMS (*e.g.,* "jumping" to any location in the dataset).

These assumptions define a specific class of exploration interfaces, characterized by the following four rules: (a) the interface supports a finite set of interactions (*i.e.,* no open-ended text boxes); (b) these interactions cannot modify the underlying dataset; (c) each interaction will request only a small fraction of data tiles; and (d) each interaction represents an incremental change to the user's current location in the dataset (*i.e.,* no "jumping"). Note that given rule (c), ForeCache does not currently support interactions that force a full scan of the entire dataset, such as searches (*e.g.,* find all satellite imagery pixels with a snowcover value above 0.7).

2.3 Building Data Tiles

To improve performance, ForeCache builds all data tiles in advance, and stores them on disk in SciDB. In this section, we explain how ForeCache builds zoom levels and data tiles in advance, which is done in 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 (*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.

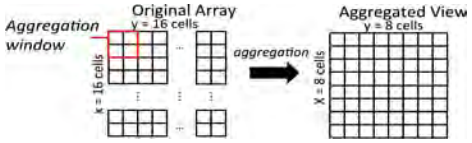


Figure 3: A 16x16 array being aggregated down to an 8x8 array with aggregation parameters $(2,2)$.

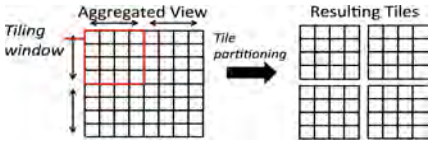


Figure 4: A zoom level being partitioned into four tiles, with tiling parameters $(4,4)$.

Building Materialized Views: 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 16x16 array (on the left), with two dimensions labeled x and y, respectively. Aggregation parameters of $(2,2)$ correspond to aggregating every 2 cells along dimension x, and every 2 cells along dimension y (*i.e.,* the red box in Figure 3). If we compute the average cell value for each window in the 16x16 array, the resulting array will have dimensions 8x8 (right side of Figure 3).

Partitioning the Views: Next, we partition each computed zoom level into data tiles. To do this, we assign a tiling interval to each dimension, which dictates the 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 one tile at zoom level i translates to four higher-resolution tiles at 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. Thus, all tiles have the same dimensions (*i.e.,* tile size), regardless of zoom level.

Computing Metadata: 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.

Choosing a Tile Size: Pre-computing tiles ensures that ForeCache provides consistently fast performance across zoom levels. However, choosing a bad tile size can negatively affect performance. For example, increasing the tile size reduces the number of tiles that can be stored in the middleware cache (assuming a fixed cache size), which could reduce ForeCache's prefetching capabilities. In our evaluation (Section 5), we take this into account by varying the number of tiles that are prefetched by ForeCache in our experiments. We plan to perform an in-depth study of how tiling param-

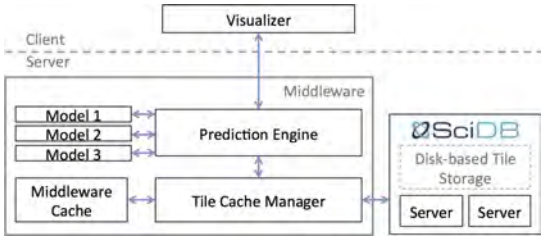


Figure 5: Overview of the ForeCache architecture.

eters affect performance as future work.

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.

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 is that the visualizer must communicate with the back-end through tile requests. For our experiments, we implemented a lightweight visualizer that runs in a web browser on the client machine (see Section 5 for more details).

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 top-most 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.

Tile Cache Manager: The cache manager decides which tiles will be stored in the main-memory middleware 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 patterns, 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 analysis phase. At the bottom level, it models the observed analysis phase with a suite of recommendation models.

We chose a two-level design because we have found that users frequently switch their browsing patterns over time. In contrast, recommendation models make strict assumptions about the user's browsing patterns, and thus ignore changes in the user's behavior. For example, Markov chains assume that the user's past moves will always be good indicators of her future actions. However, once the user finds a new region to explore, the panning actions that she used to locate this region will be poor predictors of the future zooming actions she will use to move towards this new region. 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 analysis phase that best matches her current goals, and apply the corresponding recommendation model(s) to make predictions. 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 patterns 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.

4.1 Prediction Formalization

Here, we provide definitions for all inputs to and outputs from ForeCache, and a formalization of our general prediction problem.

User Session History: The user's last n moves are constantly recorded by the cache manager and sent to the prediction engine as an ordered list of user requests: $H = [r_1, r_2, ..., r_n]$. Each request $r_i \in H$ retrieves a particular tile T_{ri}. Note that n (*i.e.,* the history length) is a system parameter set before the current session starts.

Training Data: Training data is used to prepare the prediction engine ahead of time, and is supplied as a set of traces: $\{U_1, U_2, ...\}$. Each trace U_j represents a single user session, and consists of an ordered list of user requests: $U_j = [r_1, r_2, r_3, ...]$.

Allocation Strategy: The cache manager regularly sends the current allocation strategy to the prediction engine: $\{k_1, k_2, ...\}$, were k_1 is the amount of space allocated to recommendation model m_1.

General Prediction Problem: Given a user request r for tile T_r, a set of recommender allocations $\{k_1, k_2, ...\}$, and session history H, compute an ordered list of tiles to prefetch $P = [T_1, T_2, T_3, ...]$, where each tile $T_i \in P$ is at most d moves away from T_r. The first tile (T_1) has highest priority when prefetching tiles. d is a system parameter set before the current session starts (default is $d = 1$).

The user must set the following prediction parameters in ForeCache: (1) allocation strategies for the tile cache; (2) distance threshold d; (3) user history length n; and (4) user traces as training data.

4.2 Top-Level Design

In this section, we explain the three analysis phases that users alternate between while browsing array-based data, and how we

Table 1: Input features for our SVM phase classifier, computed from traces from our user study (see Section 5 for more details).

Feature Name	Information Recorded	Accuracy for this Feature
X position (in tiles)	X position	0.676
Y position (in tiles)	Y position	0.692
Zoom level	zoom level ID	0.696
Pan flag	1 (if user panned), or 0	0.580
Zoom-in flag	1 (if zoom in), or 0	0.556
Zoom-out flag	1 (if zoom out), or 0	0.448

use this information to predict the user's current analysis phase.

4.2.1 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 [19]. 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 extended existing analysis models to match these observed 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 interaction patterns to achieve their goals. We model the low-level patterns separately in the bottom half of our prediction engine, which we describe in detail in Section 4.3.

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.

4.2.2 Predicting the Current Analysis Phase

The top half of our two-level scheme predicts the user's current analysis phase. This problem is defined as follows:

Sub-Problem Definition: given a new user request r and the user's session history H, predict the user's current analysis phase (Foraging, Sensemaking, or Navigation).

To identify the current analysis phase, we apply a Support Vector Machine (SVM) classifier, 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 RBF kernel. We implemented our classifier using the LibSVM Java Library[1].

To construct an input to our SVM classifier, we compute a feature vector using the current request r, and the user's previous request $r_n \in H$. The format and significance of each extracted feature in our feature vector is provided in Table 1. 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 collected user traces from our user study. We then hand-labeled each user request from the user traces with its corresponding analysis phase. We describe our user study and evaluate the accuracy of the analysis phase classifier in Section 5.

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

Algorithm 1 Pseudocode to update the last ROI after each request.

Input: A user request r for tile T_r
Output: *ROI*, a set of tiles representing the user's last visited ROI.

```
1:  ROI ← {}
2:  temp_ROI ← {}
3:  inFlag ← False
4:  procedure UPDATEROI(r)
5:      if r.move ="zoom-in" then
6:          inFlag ← True
7:          temp_ROI ← {T_r}
8:      else if r.move ="zoom-out" then
9:          if inFlag = True then
10:             ROI ← temp_ROI
11:         inFlag ← False
12:         temp_ROI ← {}
13:     else if inFlag = True then
14:         add T_r to temp_ROI
15:     return ROI
```

4.3 Bottom-Level Design

Once the user's current analysis phase has been identified, ForeCache employs the corresponding recommendation model(s) to predict specific tiles. ForeCache runs these models in parallel, where each model is designed to predict specific low-level browsing patterns. These recommendation models can be categorized into two types of predictions: (a) *Action-Based (AB)*: learning what to predict from the user's previous moves (*e.g.,* pans and zooms); and (b) *Signature-Based (SB)*: learning what to predict by using data characteristics, or *signatures*, from the tiles that the user has recently requested (*e.g.,* histograms). For an individual recommendation model m, the prediction problem is as follows:

Sub-Problem Definition: given a user request r, a set of candidate tiles for prediction C, and the session history H, compute an ordering for the candidate tiles $P_m = [T_1, T_2, ...]$. The ordering signifies m's prediction of how relatively likely the user will request each tile in C. The prediction engine trims P_m as necessary, depending on the amount of space allocated to m.

Here, we describe: (1) the inputs and outputs for our recommendation models; (2) how the individual models were implemented; and (3) how we allocate space to each model per analysis phase.

4.3.1 General Recommendation Model Design

Our Signature-Based recommendation model requires one additional input in order to make predictions: the last location in the dataset that the user explored in detail, which we refer to as the user's most recent *Region of Interest*, or ROI. Here we explain how we derive C, and the user's most recent ROI.

Candidate Tiles for Prediction: We compile the candidate set by finding all tiles that are at most d moves away from r. For example, $d = 1$ represents all tiles that are exactly one move away from r.

Most Recent ROI: We represent the user's most recent ROI as a set of data tiles, and use a simple heuristic to compute this set; the pseudocode is provided in Algorithm 1. When a new user request is received, the prediction engine calls UPDATEROI to update the user's most recent ROI. To find the most recent ROI, this heuristic searches through H for a match to the following pattern: one *zoom-in*, followed by zero or more *pan*'s, followed by one *zoom-out*. In lines 5-7 of Algorithm 1, a *zoom-in* triggers the collection of

a new temporary ROI ($temp_{ROI}$), and the requested tile T_r is added to $temp_{ROI}$ (line 7). We track *zoom-in*'s using the $inFlag$ variable (line 6). In contrast, an observed *zoom-out* tells the prediction engine to stop adding tiles to $temp_{ROI}$ (lines 8-12). If the $inFlag$ was set while the zoom-out occurred, we replace the user's old ROI with $temp_{ROI}$ (lines 9-10). Then, $temp_{ROI}$ is reset (line 12). Last, if $r.move = pan$ while the $inFlag$ is true, T_r (*i.e.,* the requested tile) is added to $temp_{ROI}$ (lines 13-14).

Algorithm 2 Pseudocode showing the Markov chain transition frequencies building process.

Input: For PROCESSTRACES, a set of user traces, and sequence length n.
Output: F, computed transition frequencies.

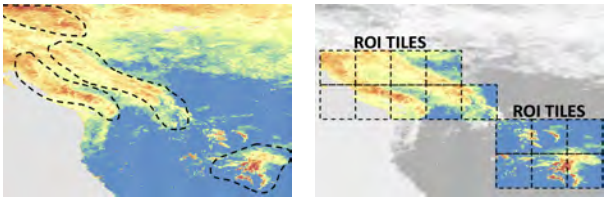
```
1:  procedure ProcessTraces({U_1, U_2, ..., U_j, ...}, n)
2:      F ← {}
3:      for user trace U_j do
4:          V_j ← GetMoveSequence(U_j)
5:          F ← UpdateFrequencies(V_j, F, n)
6:      return F
7:  procedure GetMoveSequence(U_j)
8:      V_j ← [ ]
9:      for i = 1, 2, ..., where i ≤ | V_j | do
10:         V_j[i] ← U_j[i].move
11:     return V_j
12: procedure UpdateFrequencies(V_j = [v_1, v_2, v_3, ...], F, n)
13:     for i = n+1, n+2, ..., where n < i ≤ | V_j | do
14:         F[sequence(v_{i-n}, v_{i-(n-1)}, v_{i-(n-2)}, ..., v_{i-1}) → v_i] += 1
15:     return F
```

4.3.2 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 [8]. We expand on this idea with our AB recommender, which builds an n-th order Markov chain from users' past actions.

To build the Markov chain, we create a state for each possible sequence of moves, where we only consider sequences of length n (*i.e.,* the length of H). For example, if $n = 3$, then the following are two sequences that would have their own states in the Markov chain: panning left three times (*i.e.,* $left$, $left$, $left$), and zooming out twice and then panning right (*i.e.,* out, out, $right$). After creating our states, we create an outgoing transition from each state for every possible move the user can make in the interface. In the $n = 3$ case, if the user is in state ($left$, $left$, $left$) and then decides to pan right, we represent this as the user taking the edge labeled "$right$" from the state ($left$, $left$, $left$) to the state ($left$, $left$, $right$).

We learn transition probabilities for our Markov chains using traces from our user study; the traces are described in Section 4.1. Algorithm 2 shows how we calculate the transition frequencies needed to compute the final probabilities. For each user trace U_j from the study, we extract the sequence of moves observed in the trace (lines 7-11). We then iterate over every sub-sequence of length n (*i.e.,* every time a state was visited in the trace), and count how often each transition was taken (lines 12-15). To do this, for each sub-sequence observed (*i.e.,* for each state observed from our Markov chain), we identified the move that was made immediately after this sub-sequence occurred, and incremented the relevant counter (line 14). To fill in missing counts, we apply Kneser-Ney smoothing, a well-studied smoothing method in natural language processing for Markov chains [7]. We used the BerkeleyLM [18] Java library to implement our Markov chains.



(a) **Potential snow cover ROI's in the US and Canada.** (b) **Tiles in the user's history, after visiting ROI's from (a).**

Figure 6: Example ROI's in the US and Canada for snow cover data. Snow is orange to yellow, snow-free areas in green to blue. Note that (a) and (b) span the same latitude-longitude range.

Table 2: Features computed over individual array attributes in ForeCache to compare data tiles for visual similarity.

Signature	Measures Compared	Visual Characteristics Captured
Normal Distribution	Mean, standard deviation	average position/color/size of rendered datapoints
1-D histogram	histogram bins	position/color/size distribu-tion of rendered datapoints
SIFT	histogram built from clustered SIFT descriptors	distinct "landmarks" in the visualization (*e.g.,* clusters of orange pixels)
DenseSIFT	same as SIFT	distinct "landmarks" *and* their positions in the visualization

4.3.3 Signature-Based (SB) Recommender

The goal of our SB recommender is to identify neighboring tiles that are visually similar to what the user has requested 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 6a, where the user is exploring snow cover data derived from a satellite imagery dataset. Snow is colored orange, and regions without snow are blue. Thus, the user will search for ROI's that contain large clusters of orange pixels, which are circled in Figure 6a. 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 6b is an example of some tiles that may be in the user's history if she has recently explored some of 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 2 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 2 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.

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 2). SIFT is used to identify and compare visual "landmarks" in an image, called *keypoints*. Much

like how seeing the Statue of Liberty can help people distinguish pictures of New York city from pictures of other cities, SIFT keypoints help ForeCache compare visual landmarks in two different visualizations (*e.g.,* two satellite imagery heatmaps with similar clusters of orange pixels, or two line charts showing unusually high peaks in heart-rate signals). We used the OpenCV library to compute our SIFT and denseSIFT signatures[2].

Algorithm 3 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 3). 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).

4.4 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 recommender 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 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.

[2]http://opencv.org

Algorithm 3 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 $D = \{d_1, d_2, ..., d_A, ...\}$

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 (latitude, longitude and time). Each array cell contains a vector of wavelength measurements, where each 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) [21]. 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 to 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. ForeCache's tile computation process resulted in nine total zoom levels for this dataset, where each level was a separate layer of data 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 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 [20]. 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).

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

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 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 ForeCache, 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 comparison, similar to Lee *et al.* [12]. 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 list.

We simulated space allocations in our middleware cache by varying k in our accuracy measurements. Thus measuring prediction accuracy becomes equivalent to measuring the hit rate of our tile cache. For example, $k = 2$ meant that ForeCache only had space to fetch two tiles before the user's next request. We varied k from 1 to 8 in our experiments. At $k = 9$, we are guaranteed to prefetch the correct tile, because the interface only supports nine different moves: 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 [8], the "Momentum" and "Hotspot" models. Several more recent systems, such as ATLAS [6] and ImMens [16] 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 chain, 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 num-

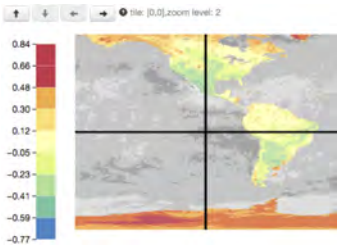


Figure 7: ForeCache browsing interface.

ber 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.

5.3 User Study

To ascertain whether prefetching was a viable strategy for exploring multidimensional scientific datasets, we worked directly with earth and ocean scientists at the University of California Santa Barbara (UCSB) and the University of Washington (UW) to: (1) choose a use case of interest to our collaborators (MODIS snow cover); and (2) develop a set of search tasks for this use case that domain scientists with diverse backgrounds and skill sets could complete during the study. 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 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 Participants

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. Participants were recruited at UW and UCSB.

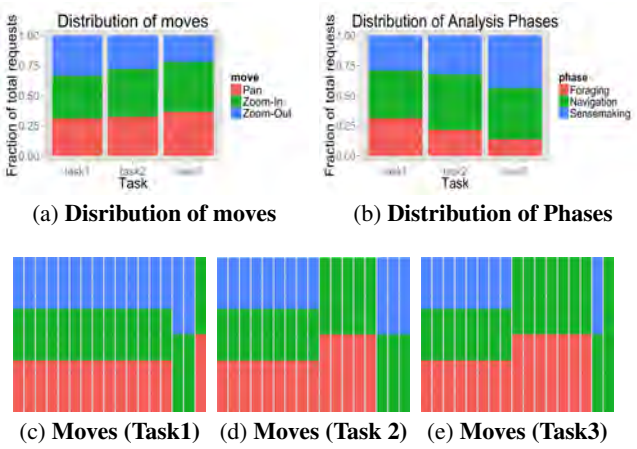
5.3.2 Browsing Interface

Figure 7 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 7) 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.

Directional buttons ensured that users' actions mapped to specific data tiles. Though different from existing geospatial interfaces (*e.g.,* Google Maps), our browsing interface provides clear and efficient button-based navigation through the dataset. Furthermore, study participants commented that this navigation design is useful when selecting specific data ranges for further analysis.

5.3.3 Browsing Tasks

Participants completed the same search task over three different regions in the NDSI dataset. For each region, participants were



(a) **Disribution of moves** (b) **Distribution of Phases**

(c) **Moves (Task1)** (d) **Moves (Task 2)** (e) **Moves (Task3)**

Figure 8: Distribution of moves (a) and phases (b), averaged across users, partitioned by task; distribution of moves for individual users for Task 1 (c), Task 2 (d), and Task 3 (e). In Figures (c), (d), and (e): panning is red, zooming in is green, and zooming out is blue; users with similar move distributions are grouped together.

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 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 8a 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. We also found that large groups of users shared similar browsing patterns, shown in Figures 8c-8e. These groupings further reinforce the reasoning behind our analysis phases, showing that most users can be categorized by a small number of specific patterns within each task, and even across tasks.

5.3.5 Evaluating Our Three Analysis Phases

To demonstrate some of the patterns that we found in our user traces, consider Figure 9, which plots the change in zoom level

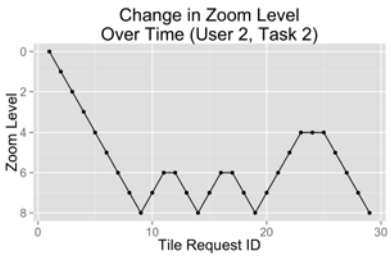


Figure 9: Change in zoom level per request as study participant 2 completed task 2.

over time for one of our user traces. The coarsest zoom level is plotted at the top of Figure 9, 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 9 that this user's behavior corresponds directly to the three analysis phases described in Section 4.2.1. The user's return to coarser views near the top of Figure 9 correspond to the user returning to the Foraging phase (*e.g.,* 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. Furthermore, we found that 13 out of 18 users exhibited this same general exploration behavior throughout their participation in the 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 apply *leave-one-out cross validation* [11], a common cross-validation technique 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. After evaluating each user individually, we averaged the results across all users to produce our final accuracy calculations.

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 this experiment, we manually labeled each request in our request logs with one of our 3 analysis phases. Figure 8b 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

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 test whether our AB and SB models excelled in accuracy for their intended analysis phases. To do this, we evaluated how accurately our individual 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 $n = 10$, which we refer to as Markov2 through Markov10, respectively. Each Markov chain only took milliseconds to train. We found that $n = 2$ was too small, and resulted in worse accuracy. Otherwise, we found negligible improvements in accuracy for lengths beyond $n = 3$, and thus found $n = 3$ (*i.e.,* Markov3) to be most efficient Markov chain for our AB model.

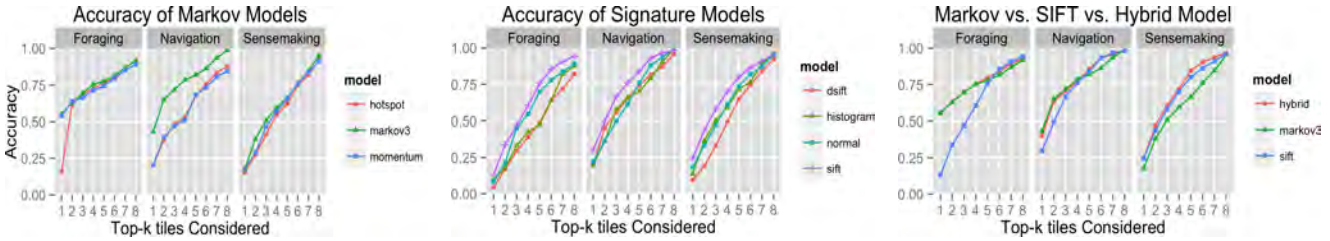
Figure 10a shows the prediction accuracy of our AB model compared to the Momentum and Hotspot models, with increasing values of k. Note that k represents the total space (in tiles) that each model was given for predictions (see Section 5.2.2 for more information). In Figure 10a, we see that for the Foraging 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 10b 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 provided the best overall accuracy. We expected a machine vision feature like SIFT to perform well, because users are comparing images when they analyze MODIS tiles.

We found that the denseSIFT signature did not perform as well as SIFT. denseSIFT performs worse because it matches entire images, whereas SIFT only matches small regions of an image. Here, relevant visualizations will contain clusters of orange snow pixels, 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 (orange) pixels. Thus, many relevant tiles will not appear to be similar with regards to the denseSIFT signature.

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 AB recommender with $n = 3$ (*i.e.,* Markov3)



(a) **Accuracy of our best AB model (Markov3) and existing models.** (b) **Accuracy of the four signatures in our SB model.** (c) **Accuracy of our final engine (hybrid) and our best individual models.**

Figure 10: Accuracy of our AB model, SB model, and final prediction engine (*i.e.,* hybrid model).

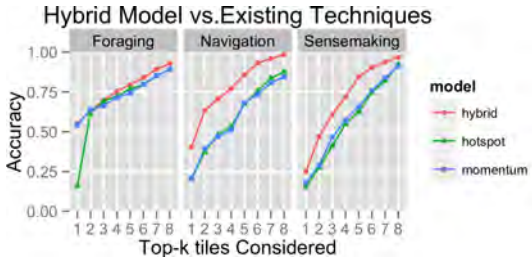


Figure 11: Accuracy of the hybrid model compared to existing techniques.

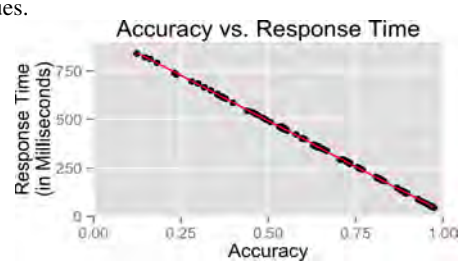


Figure 12: Plot of average response time given prefetch accuracy, for all models and fetch sizes (linear regression: Adj R^2=0.99985, Intercept=961.33, Slope=-939.08, P=1.1704e-242.

and the SIFT SB recommender. Note that we updated our original allocation strategies based on our observed accuracy results. 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 $k > 4$.

Figure 10c shows that our final prediction engine successfully combined the strengths 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 11. We see that for the Foraging phase, our prediction engine performs as well (if not better) than existing techniques. For the Navigation phase, we achieve up to 25% better prediction accuracy. Similarly for the Sensemaking phase, we see a consistent 10-18% improvement in accuracy.

5.5 Latency

We used the same setup from our accuracy experiments to 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 these two 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

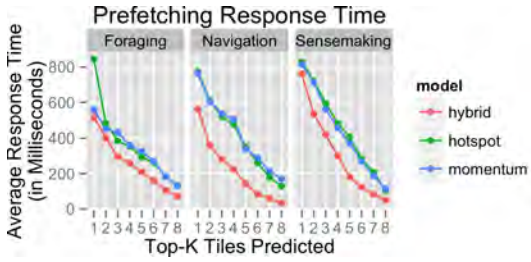


Figure 13: Average prefetching response times for hybrid model and existing techniques.

missing tile, which was slower. On average, the middleware took 19.5 ms to send tiles for a cache hit, and 984.0 ms for a cache miss.

To evaluate our claim that accuracy dictates latency, we plotted the relationship between prefetching accuracy and average response time (*i.e.,* average latency), shown in Figure 12. 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 13, where we calculated the average response times for three models. 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.,* [8, 6, 16]). Specifically, as shown in Figure 13, 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 [17, 15]. 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 11, 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 provides significantly better performance over other systems, there is still room for improvement. Here, we discuss extending the ForeCache prediction engine beyond two dimensions, and present several ideas for further improvements to ForeCache both in performance and applicability to other datasets.

6.1 Prediction Beyond 2D datasets

The current ForeCache system focuses on 2D visualization. To support multidimensional exploration (*e.g.,* 3D datasets), we can employ a *coordinated view* design, where tiles are represented by several visualizations at the same time. For example, to extend our snowcover example to three dimensions (latitude, longitude and time), we could create two navigable views: (1) a snowcover heatmap (*e.g.,* Figure 7); and (2) a line chart showing maximum snow cover over time for the current tile. To navigate via latitude and longitude, the user moves in the heatmap. To navigate via time, the user moves in the line chart.

However, the number of tiles grows exponentially with the number of dimensions, increasing the search space and making prediction more challenging. One solution is to insert a pruning level between our phase classifier and recommendation models to remove low-probability interaction paths. A second solution is to restrict the number of possible interactions ahead of time by having the user choose only two dimensions to explore at a time. This technique reduces the user's range of possible actions, enabling ForeCache to still make accurate predictions. We plan to implement 3D visualizations and pruning in ForeCache, and plan to conduct a new user study on higher-dimensionality datasets to evaluate the accuracy and response time benefits provided by these new techniques.

6.2 Future Work

We describe three future directions for ForeCache: (1) extending our tiling scheme; (2) implementing more signatures; and (3) new prediction and caching techniques for multi-user environments.

Our proposed tiling scheme works well for arrays. 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. We also plan to study how tiling parameters affect performance for array and non-array datasets.

We manually identified four signatures for our SB recommender, and found that SIFT works best for the NDSI dataset. However, other features may be more appropriate for different datasets. For example, counting outliers or computing linear correlations may work well for prefetching time series data. We plan to build a general-purpose signature toolbox with more of these signatures, and plan to extend ForeCache to learn what signatures work best for a given dataset automatically.

To evaluate ForeCache's new tiling scheme and signature toolbox, we plan to conduct a user study on non-array-based datasets. For example, we plan to support the MIMIC II medical dataset, which provides hospital data recorded for thousands of patients and many data types (*e.g.,* unstructured text, tabular, and array data).

In addition, 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.* [9] 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 [10] utilizes pre-computed samples and a distributed DBMS environment to shorten execution times for data 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.* [14] 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 [16] 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 boost prefetching accuracy. Lee *et al.* [12] and Cetintemel *et al.* [5] propose using Markov chains to predict user movements. Chan *et al.* [6] and Doshi *et al.* [8] propose several strategies–the most sophisticated being Markov chains–to predict future user requests. We compare with two of these strategies, Momentum and Hotspot, in our experiments. Yesilmurat *et al.* [25] propose techniques similar to the Momentum model for prefetching geospatial data. Li *et al.* [13] combine the Hotspot model with Markov chains to further improve performance. ForeCache builds on these existing techniques 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 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).

Acknowledgements: This project was supported by NSF grants IIS-1452977 and IIS-1218170, and the Intel Science and Technology Center for Big Data.

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] S. F. Chen and J. Goodman. An empirical study of smoothing techniques for language modeling. *Computer Speech & Language*, 13(4):359–394, Oct. 1999.

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

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

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

[11] 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.

[12] 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.

[13] 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.

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

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

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

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

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

[19] 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.

[20] 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.

[21] 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.

[22] E. Soroush, M. Balazinska, S. Krughoff, and A. Connolly. Efficient Iterative Processing in the SciDB Parallel Array Engine. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*, SSDBM '15, pages 39:1–39:6, New York, NY, USA, 2015. ACM.

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

[24] R. Taft, M. Vartak, N. R. Satish, N. Sundaram, S. Madden, and M. Stonebraker. GenBase: A Complex Analytics Genomics Benchmark. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 177–188, New York, NY, USA, 2014. ACM.

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