# Stochastic Data Acquisition for Answering Queries as Time Goes by

Zheng Li
University of Massachusetts, Lowell
zli@cs.uml.edu

Tingjian Ge
University of Massachusetts, Lowell
ge@cs.uml.edu

## ABSTRACT

Data and actions are tightly coupled. On one hand, data analysis results trigger decision making and actions. On the other hand, the action of acquiring data is the very first step in the whole data processing pipeline. Data acquisition almost always has some costs, which could be either monetary costs or computing resource costs such as sensor battery power, network transfers, or I/O costs. Using outdated data to answer queries can avoid the data acquisition costs, but there is a penalty of potentially inaccurate results. Given a sequence of incoming queries over time, we study the problem of sequential decision making on when to acquire data and when to use existing versions to answer each query. We propose two approaches to solve this problem using reinforcement learning and tailored locality-sensitive hashing. A systematic empirical study using two real-world datasets shows that our approaches are effective and efficient.

## 1. INTRODUCTION

The importance of actions has long been realized—as Aristotle contemplated: "*But how does it happen that thinking is sometimes accompanied by action and sometimes not, sometimes by motion, and sometimes not?*" [20]. Data and actions are often tightly coupled. On one hand, data analysis results trigger decision making and actions. On the other hand, the action of acquiring (and sometimes cleaning) data is the very first step in the whole data processing pipeline [8]. Data acquisition almost always has some costs, which could be either *monetary costs* or *computing resource* costs such as sensor battery power, network transfers, or I/O costs. At the same time, big data often have inherent *redundancy* or *continuity* [8]. For instance, a stream of observing sensory data is a time series which typically exhibits correlations in time and/or space and changes gradually.

EXAMPLE 1. *Consider a dynamic traffic routing service provider. Significant efforts are required to obtain real-time traffic information of each road. The service provider may pay a third party for selectively acquiring traffic updates at specific road segments (as in a data market [17, 3, 1]). Such data is valuable for the business of providing traffic routing (e.g., from address A to address B) based on the current road delays. This is illustrated in Figure 1, where all data updates (changes in delays of various roads) naturally occur in the the data source $\mathbb{S}$ while the query processor $\mathcal{P}$ accesses a data store $\mathbb{D}$, which has a copy of all data items in $\mathbb{S}$ but possibly outdated. Upon a query, $\mathcal{P}$ decides whether to pay the cost of re-acquiring the needed data items from the data source (with the risk of those data items actually unchanged), or to simply use the current version of data items to answer the query (with the risk of outdated answer). The main idea of our work is to statistically estimate (1) whether a data item is likely outdated and affects a query result significantly, and (2) whether it is likely to be used often in the near future. A long-term optimization based on such statistical information is performed to guide the data acquisition decisions.*

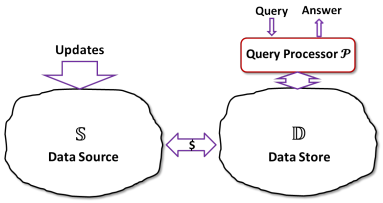


**Figure 1: The sequential data acquisition problem.**

Likewise, data acquisition in *sensor networks* (e.g., for structural health monitoring, gas leak detection, and volcano monitoring) is known to be expensive and consume *significant battery power* [12]. Similar decisions are required on whether to acquire data updates before answering a query. More generally, the cost of acquiring the data for query answering also includes long *network or I/O delays*. For sensor networks, like in Example 1, $\mathbb{S}$ is the environment being observed, and updates to $\mathbb{S}$ (e.g., temperature changes) happen by the nature of the environment, while $\mathbb{D}$ represents the sensor readings $\mathcal{P}$ chooses to acquire from $\mathbb{S}$, by paying data acquisition costs such as battery power.

As another use case, a member of our research group is working with a nonprofit organization on a project where a server program continuously handles the requests of client programs (mobile phones, web services, and other application programs) by requesting continuous *weather forecast* and *alert* streaming data over the Internet from the National Oceanic and Atmospheric Administration (NOAA)'s National Weather Service (NWS) ([4] and [6]). The server program needs to process the selectively retrieved data (e.g., based on regions, as query predicates) from NWS into the format required by a specific client program. NWS con-


*Proceedings of the VLDB Endowment,* Vol. 10, No. 3

stantly updates its weather forecasts and alerts of all regions of the U.S. in its source database ($\mathbb{S}$). The server program ($\mathcal{P}$) must cache its retrieved data ($\mathbb{D}$), as it is impossible for it to keep up with the speed of requests and fetch data from NWS over the Internet for each request, and interpret it (i.e., paying data acquisition cost). Furthermore, different client programs/queries have different requirements on how up-to-date the forecasts or alerts are—some mission-critical programs may have very rigid requirement on up-to-date weather information, while other programs do not have to pay the high cost of data acquisition and can live with slightly outdated data, and so on. Thus, the server may have multiple "service levels" (policies), i.e., multiple utility functions that vary on the relative weight between data acquisition cost and inaccurate query result cost.

As the queries are revealed to the data processor $\mathcal{P}$ in an online manner, the data acquisition decisions are sequential, but not isolated among queries. For example, suppose the travel delay of a particular road segment $r$ in Example 1 has a probability of 0.2 to be updated from one query to the next, and that two consecutive queries $q_1$ and $q_2$ both require $r$. Then the action of acquiring $r$ for $q_1$ will likely benefit $q_2$ as well. Without acquiring $r$ again for $q_2$, with probability 0.8 it is still up-to-date. Therefore, our goal is to find the *long-term* optimal policy which determines the action of either acquiring data from $\mathbb{S}$ for a query (and update $\mathbb{D}$) or merely using the current version in $\mathbb{D}$ to answer the query. The query sequence can be infinite. In essence, the current decisions can have both short-term and long-term effects; the best choice now depends on future situations as well as the actions on them.

## 1.1 Our Contributions

We propose to study this novel problem described above, defined as a *sequential data acquisition* problem in Section 2. Our general approach is to use a machine learning technique called *reinforcement learning* [22]. We first use a mathematical framework called *Partially Observable Markov Decision Process* (POMDP) [16], which characterizes system *states*, *actions* (acquiring data or not), and *rewards*. The model parameters, including state transition probabilities, encode the statistical information that we can learn offline based on recent query processing. Our framework is fairly extensible and incorporates factors such as: (1) each data block may have a different update rate, (2) a query may read an arbitrary subset of data blocks, and (3) we may add new actions and budget constraints for data acquisition.

For a practical system, the number of POMDP states as modeled above is too large to be feasible for any existing POMDP solver to solve the model. We thus devise an algorithm that coalesces the *states* and *observations* in our POMDP into a significantly smaller space, resorting to two means of hash functions—both of which are of the nature of *Locality Sensitive Hashing* (LSH) [19]. The POMDP on this smaller state space can then be efficiently solved. We analyze the accuracy of the resulting model and find experimentally in Section 8 that it is very effective.

An overhead of the above approach is that we have to learn the parameters of POMDP first. However, this learning can be *offline* and the POMDP solver pre-computes a policy once, which is used for a long time. At runtime, looking up the policy for data acquisition decisions is *instantaneous* and *negligible* compared to query processing. Nonetheless, when the data update and query trends change, we have to learn these functions again. Hence, we propose a second approach based on the model-free Q-learning [15], which dynamically maintains an optimal policy (to acquire data or not), balancing between *exploiting* the existing policy and *exploring* new data-update/query trends to adjust the policy.

Somewhat similar to our work is adaptive data caching by Olston et al. [21]. The idea is that the data source server obtains *every* data item update, but an update is only refreshed to a cache sometimes. The cache contains an interval for a value (thus approximate), and a cache refreshment is triggered in two ways: *value-initiated* refresh (when the server finds that the actual value drifts away from the set precision interval), and *query-initiated* refresh (when the set precision does not meet query accuracy requirement). The goal of [21] is to optimize the setting of cached value precision intervals so as to minimize the cache refreshment. Our work is fundamentally different in that we do not have a "server" that obtains *every* data update and decides if an update needs to be refreshed to a cache. We are about *selective* data acquisition.

In addition, Deshpande et al. [12] study data acquisition in sensor networks. However, as in all previous work in data acquisition, [12] does not do sequential planning as we do based on two statistical estimations: (1) whether a value is likely outdated and significantly affects query result, and (2) whether a data item is likely used often in queries in the near future. Our experiments in Section 8 show that our selective data acquisition significantly outperforms the baseline approach that does not do statistical estimation and sequential planning. Nonetheless, the main contributions of [12] include a detailed cost model for various components of data acquisition specific to sensor networks, which we do not have. Instead, we propose a general framework for various data acquisition tasks, into which a data acquisition cost model as in [12] can be integrated. We survey other related work in Section 9. Our contributions are summarized below:

- We propose to study the sequential data acquisition problem common in arising applications (Section 2).
- We use reinforcement learning and the POMDP model for the first approach to this problem (Section 3).
- Coalescing the states in POMDP with two means of locality sensitive hashing, we make the solution feasible for realistic data systems (Sections 4 and 5).
- We propose a second approach using the model-free Q-learning to avoid the overhead of re-learning of model parameters for dynamic environments (Section 6).
- We extend both approaches to solve a practical variant of the problem, where we have a constraint on the maximum budget in a sliding window (Section 7).
- We perform a systematic experimental evaluation using two real-world datasets (Section 8).

# 2. PROBLEM STATEMENT

## 2.1 Problem Formulation

Figure 1 illustrates the setting. A *data source* $\mathbb{S}$ contains a set of $n$ data blocks $b_1, ..., b_n$, which are constantly updated at various rates. A *data store* $\mathbb{D}$ maintains a copy of the data blocks acquired by a *query processor* $\mathcal{P}$, who uses $\mathbb{D}$ to answer queries. A query $q$ received by $\mathcal{P}$ *accesses* a set of data blocks $R$; we say that $q$ *requires* $R$, or $q$ *references* $R$.

To answer a query $q$ that requires a set $R$ of blocks, $\mathcal{P}$ decides in an online manner whether to use the current versions of data blocks in $\mathbb{D}$, or to acquire the most recent versions of blocks in $R$ from $\mathbb{S}$ (and update $\mathbb{D}$) by paying a cost $c_d(R)$, which is called the *data cost*. If $\mathcal{P}$ decides not to pay the data cost, and the version in $\mathbb{D}$ is outdated, there is a *query result penalty cost* $c_q(R, q)$ for its inaccuracy. Let a *policy* be an oracle that tells what next action for $\mathcal{P}$ to take, given $\mathcal{P}$'s current action and the distribution of states $\mathcal{P}$ is believed to be in.

*Definition 1.* **(Sequential Data Acquisition Problem)** An infinite sequence of queries are revealed to a query processor $\mathcal{P}$ in an online manner. Upon receiving each query $q$, $\mathcal{P}$ decides, also in an online fashion, whether or not to acquire the data blocks $R$ required by $q$ from the data source $\mathbb{S}$. The *sequential data acquisition problem* for $\mathcal{P}$ is to find an optimal policy $\pi^*$ so as to minimize the *long term* costs, which include both data costs and query result penalty costs, for answering the sequence of queries.

The notion of "long term costs" will be made more precise in Section 2.2 below (i.e., expected discounted sum of costs). Analogous to DBMS blocks/pages, in this work, we assume simple equal-size partitioning of blocks. For instance, in Example 1, the sequence of data from each road can be treated as a block; data tuples from each sensor in a sensor network are a block. More sophisticated block partitioning is beyond the scope of this paper. In addition, in these applications, typically there are one or a few highly dynamic attributes (which are to be kept track of), while other attributes (e.g., locations) are relatively stable/static. An example is the multidimensional array model which is the most common data model in scientific databases, where the *dimension* attributes are usually static and in the clear. Most, if not all, queries have predicates over ranges of the dimensions. The block partitions should be based on stable/static attributes.

$R$ can be easily determined if there is a predicate over block-partitioning attributes. In the (arguably rare) event that either (E1) there are predicates on static attributes, but none of these attributes is aligned with block partitions, or (E2) there are no predicates on static attributes, then we use a simple two-step process. In the first step, we execute the query using the existing data (without acquisition), and get a version of qualified/required set of blocks $R$. In the second step, we use $R$ to determine the action of whether to acquire the data in $R$ (and re-run the query) or to just stop there (returning the result from the first step). Note that in the event of E2 above, $R$ may be approximate (a precise $R$ would include all blocks of the table, as there is no predicate on stable attributes).

## 2.2 Preliminaries

**POMDP.** Markov Decision Processes (MDPs) provide a mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of a decision maker [22]. MDPs are an extension of Markov chains; the difference is the addition of actions (allowing choice) and rewards (giving motivation). Figure 2 shows an example of a simple MDP with three states $s_0, s_1, s_2$ and two actions $a_0, a_1$. A yellow squiggle arrow shows the reward associated with the corresponding transition black edge. Formally, an MDP is defined as a tuple $(S, A, T, R)$, where $S$ is a set of *states*, $A$ is a set of *actions*, $T$ is the *transition function* $T(s, a, s') = Pr[s_{t+1} = s' \mid s_t = s, a_t = a]$ (where $s_t$ and $a_t$ denote the state and action at time $t$, resp.), and $R(s, a)$ is the *reward* for executing action $a$ in state $s$. The *core problem* of MDPs is to find a *policy* for the decision maker: a function $\pi$ that specifies the action $\pi(s)$ that the decision maker will choose when in state $s$. The goal is to choose a policy $\pi$ that will maximize some cumulative function of the random rewards, typically the expected discounted sum over a potentially infinite horizon: $\sum_{t=0}^{\infty} \gamma^t R(s_t, \pi(s_t))$, where $\gamma$ is called a *discount factor* and satisfies $0 \leq \gamma < 1$. The larger the discount factor (closer to 1), the more effect future rewards have on current decision making. This is equivalent to *minimizing* expected discounted sum of *costs* as in Definition 1.

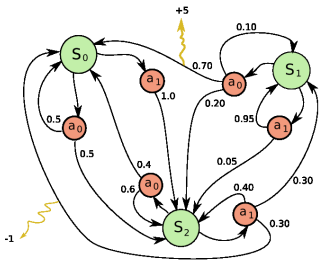


**Figure 2: A simple MDP with 3 states and 2 actions.**

In MDP, we assume that the decision maker knows the environment state *exactly*; in other words, the environment state must be fully observable. However, this assumption often does not hold in reality. A Partially Observable Markov Decision Process (POMDP) is for this problem. In addition to $S$, $A$, $T$, $R$, a POMDP has two extra elements: a set of *observations* $\Omega$ and an *observation function* $O(a, s', o) = Pr[o_{t+1} = o \mid a_t = a, s_{t+1} = s']$. There are a number of tools for solving a POMDP model. We need to first define and learn the $S$, $A$, $T$, $R$, $\Omega$, and $O$ elements, and provide them to a POMDP solver, which in turn computes the optimal policy. At runtime, when each query comes in, we interact with a program and feed our observation to it. Accordingly, the program consults the computed *policy* and notifies us the next action. This step at *runtime* is *instantaneous* as the policy is pre-computed only once. We refer the reader to [22, 16] for the details of POMDP.

**Locality-sensitive hashing.** Locality-sensitive hashing (LSH) is a method of performing probabilistic dimension reduction of high-dimensional data [19]. The idea is to hash the input items so that similar items are mapped to the same buckets with high probability. A key element is defining "similarity". One metric we use is *set similarity*. In particular, the *Jaccard similarity coefficient* between two sets $A$ and $B$ is defined as $J(A, B) = \frac{|A \cap B|}{|A \cup B|}$, which is a value in [0,1] and measures the similarity of two sets (e.g., when $A = B$, $J(A, B) = 1$). The *Jaccard distance* is $1 - J(A, B)$. MinHash is a LSH mechanism based on this metric [10]. Our goal is to hash a set of elements $E$. A simple version of MinHash is to start with $k$ independent conventional (random) hash functions $h_1, ..., h_k$. Then the MinHash value for the whole set $E$ is the concatenation of $k$ minimum hash values: $min_{e \in E} h_1(e), ..., min_{e \in E} h_k(e)$.

**Q-learning.** As surveyed by Kaelbling et al. [15], there are mainly three types of model-free approaches in the literature: (1) Adaptive Heuristic Critic (AHC), (2) Q-learning, and (3) Model-free learning with average reward. AHC architectures are more difficult to work with than Q-learning on a practical level [15]. It can be hard to get the relative

learning rates right in AHC. In addition, Q-learning is exploration insensitive; i.e., it will always converge to the optimal policy regardless of the details of the exploration strategy. Finally, type (3) above, which uses an average reward instead of a time-discounted reward, does not always produce bias-optimal policies [15]. For all these reasons, Q-learning is the most popular and the most effective model-free reinforcement learning algorithm, and is our choice.

In Q-learning, a decision maker learns an action-value function, or Q-function $Q(s, a)$, giving the expected utility of taking a given action $a$ in a given state $s$ and following the optimal policy thereafter. The basic idea of Q-learning is to iteratively update the $Q(s, a)$ values based on $Q(s, a) \leftarrow Q(s, a) + \alpha(R(s) + \gamma max_{a'} Q(s', a') - Q(s, a))$. Here, $0 < \alpha \leq 1$ is called the *learning rate*, which dictates how fast $Q(s, a)$ is adjusted. $R(s)$ is the *immediate reward* of this step at state $s$. Like POMDP, $0 \leq \gamma < 1$ is the *discount factor*, trading off the importance of sooner versus later rewards. Thus, $R(s) + \gamma max_{a'} Q(s', a')$ is the new estimate of $Q(s, a)$, i.e., the total expected rewards from state $s$ onwards, where $s'$ and $a'$ are the next state and action. The difference of this new estimate from the previous one, $Q(s, a)$, is used to adjust it, subject to the learning rate.

# 3. FROM DATA TO MODEL TO ACTION

We present a basic approach that maps our problem to a POMDP. The challenges include: (1) each data block in $\mathbb{S}$ may have a different update rate (unknown to $\mathcal{P}$), which may change over time; (2) each query may require an arbitrary subset of data blocks. To cope with (1), we make the *update rate* part of the POMDP *state* of a data block $b_i$. The key idea is that through *observations*—the data source $\mathbb{S}$ can provide simple statistics such as the number of updates to $b_i$ recently—$\mathcal{P}$ estimates the current update rate of $b_i$. This in turn affects the transition probability on whether $b_i$ is up-to-date in $\mathbb{D}$ from one decision point to the next. Therefore, the dynamicity of update rate immediately takes effect through POMDP's *state transition function*, and the mechanism is encoded in the POMDP model.

Another novelty is that we incorporate the *queries* as part of the *state* of POMDP, in response to (2) above. Whether each data block $b_i$ is used in the *current* query is treated as part of the state. The state transition function will thus encode the data block access *correlation* in the query sequence. This information is significant. For example, due to "locality of reference" [11], it is very likely that a data block $b_i$ is referenced many times in a short amount of time (e.g., traffic on a particular road segment during rush hour is queried frequently). The optimal policy of a POMDP would probably acquire $b_i$ once, knowing that it will be used by subsequent queries many times before its next update. We now describe each component of the POMDP model.

**State Set.** Every state in the state set $S$ consists of the information of each data block $b_i$, denoted as $\{(u_i, p_i, r_i) \mid 1 \leq i \leq n\}$, where $u_i$ is a Boolean indicating whether $b_i$ in $\mathbb{D}$ is currently up to date, $p_i$ is the update probability (from one decision-point/query to the next), and $r_i$ signifies whether $b_i$ is required in the current query. We discretize $p_i$ into a number of levels (e.g., five levels $p_i = 1$ to 5, corresponding to five buckets between 0 and 1). Note that $\mathcal{P}$ does not know the true update probabilities at $\mathbb{S}$, but can only estimate $p_i$ from observations, which will be discussed shortly. $u_i$ is a hidden random variable which can be estimated from $p_i$. This is because we do not know whether a block has changed until someone acquires it; someone must pay the cost even if it turns out to be unchanged. In Example 1, someone has to pay the effort to observe/report the traffic of a road. In sensor networks, $\mathbb{S}$ is the environment and power must be consumed to acquire a reading before knowing it has changed or not. In the NWS example, network cost must be paid to get updates, if any. The system has to account for the data acquisition cost regardless ($\mathcal{P}$ benefits from the data and pays for it). Note that even if $u_i$ were known for some application, our general framework would still work in that special case. More delicate cost accounting strategies are a topic of future study.

**Action Set.** The action set $A$ contains two values {*acquire, skip*}, denoting to acquire the currently required data blocks, or to skip the acquisition and use the existing versions in $\mathbb{D}$ for query processing, respectively. A subtle point here is that, in our model, an action (to acquire or to skip) in the optimal policy depends on the data and query, since the system state $s$ contains the information on each block (in particular, the ones required by the current query and their freshness). More importantly, we can include in the system state $s$ whichever factors we deem important for the action. For example, we can put query type in $s$, and our framework works in the same manner. We can have actions with a finer granularity too. For instance, we can have an action that says "acquire top-$k$ required data blocks that are outdated, ranked in descending order of their update probability". Yet another extension is discussed and implemented in Section 7. These indicate good extensibility of the framework.

**Stochastic Transition Function.** Next we look at the stochastic transition function $T(s, a, s') = Pr[s_{t+1} = s' \mid s_t = s, a_t = a]$. Let $s = \{(u_i, p_i, r_i)\}$ and $s' = \{(u'_i, p'_i, r'_i)\}$. The transition function $T$ describes the connections among states and actions. For example, when the action $a = skip$, $T(s, a, s') = 0$ if $u_i = false$ and $u'_i = true$ for some $i$. This is because $b_i$ cannot turn from outdated to up-to-date when the action is *skip*. On the other hand, the probability of transitioning from $u_i = true$ to $u'_i = false$ is simply $p_i$ (update probability). When $a = acquire$, we must have $u'_i = true$ if $b_i$ is referenced in the query. We discuss how to obtain the complete $T$ function in Section 5.

**Reward Function.** The reward function $R(s, a)$ is the reward for executing action $a$ in state $s$. The reward in our model is the negative value of two costs: (1) the cost $c_d(R)$ for acquiring referenced data blocks $R$ if the action $a = acquire$, or (2) the query result inaccuracy cost $c_q(R, q)$ for answering the current query $q$ with outdated data blocks if the action $a = skip$. As the decision-maker, there is a choice between the two types of cost that one has to make: either to pay the cost to acquire the data, or to (potentially) pay the cost of getting inaccurate results. For someone who has to choose, the "preference" has to be somehow quantified in a uniform manner. This is common in AI and economics, where it is also called a *utility function* [22]. How to combine the two costs is application dependent. For example, as a simple setting of preference which is used in our experiments, we assume that acquiring $f$ $(0 < f < 1)$ fraction of all data blocks is equivalent in cost to the current query's result potentially being inaccurate by $f$ fraction also. Certainly any other preference balance point within the spectrum can be easily adopted too. As such, any fraction of query inaccuracy can be converted to an equivalent cost of the number of

blocks to be acquired, and the reward is the negative value of the sum of these two costs.

**Observation Set.** We discuss the set of observations $\Omega$ in our POMDP model. An observation $o$ consists of two parts: (1) the set of required data blocks in the *next* query, and (2) the estimated update probabilities of each data block required by the *current* query, if the action $a = acquire$. If the action $a = skip$, part (2) is empty.

These two parts are exactly what one can observe from the system, and what chains one state $s$ with the next state $s'$. Part (1) can be observed as discussed in Section 2.1. We now show part (2). There are multiple ways to learn a block's update probability (e.g., based on historical data); one way is as follows. It is easy to keep the timestamps of the last $c$ updates for each data block at $\mathbb{S}$, as metadata (for a small $c$). When a data block $b_i$ is acquired by $\mathcal{P}$, we can estimate its *update rate* (i.e., number of updates per unit time) as $\theta_i = \frac{c}{t_0 - t_c}$, where $t_0$ is the current time, and $t_c$ is the timestamp of the $c$'th to last update. Alternatively, $\theta_i$ can be estimated from the amount of change between two acquisitions of $b_i$. Since the query processor $\mathcal{P}$ knows the timestamp of every query, it is also easy to estimate the *query rate* $\theta$ (i.e., number of queries per unit time—to any data blocks). Then we can estimate the *update probability* (i.e., the probability of being updated between two consecutive queries) for this data block as follows.

THEOREM 1. *A maximum likelihood estimate of the update probability $p_i$ of a data block $b_i$ is $p_i = min(\frac{\theta_i}{\theta}, 1)$, where $\theta_i$ is the update rate of $b_i$ and $\theta$ is the current query rate (to any data blocks).*

PROOF. Consider the scenario that during time interval $t_0$ - $t_c$ we observe $\theta(t_0 - t_c)$ new queries (to any blocks) and $c = \theta_i(t_0 - t_c)$ new updates to $b_i$. First consider the case $\theta_i < \theta$. Suppose we are given the parameter $p_i$, i.e., updates to $b_i$ currently arrive randomly with probability $p_i$ between two queries (which may vary later). Then the number of new updates $N$ follows a binomial distribution $B(\theta(t_0 - t_c), p_i)$. Thus, the likelihood function is $\mathcal{L}(p_i|N = c) = \binom{\theta(t_0-t_c)}{\theta_i(t_0-t_c)} p_i^{\theta_i(t_0-t_c)}(1-p_i)^{\theta(t_0-t_c)-\theta_i(t_0-t_c)}$. To get the maximum of the likelihood function, we take the log of it and let: $\frac{\partial ln\mathcal{L}}{\partial p_i} = \frac{\theta_i(t_0-t_c)}{p_i} - \frac{\theta(t_0-t_c)-\theta_i(t_0-t_c)}{1-p_i} = 0$, which gives us the maximum likelihood estimate of $\hat{p_i} = \frac{\theta_i}{\theta}$. Clearly this probability cannot be greater than 1 when $\theta_i > \theta$. □

In practice we expect usually $\theta_i < \theta$, for $\theta_i$ is the update rate of one particular data block $b_i$ while $\theta$ is the query rate of the whole system, which makes the estimate $p_i < 1$.

**Observation Function.** The observation function is $O(a, s', o) = Pr[o_{t+1} = o \mid a_t = a, s_{t+1} = s']$, which encodes the connection between observed evidence and the next state, given the action. The next state $s'$ clearly needs to be consistent with the observation $o$. Recall that the observation only contains $r_i$'s and possibly some blocks' $p_i$'s and $u_i$'s (if the action is *acquire*); thus we cannot deterministically observe the next state, but only estimate a distribution of states (called *belief state*).

# 4. REDUCED MODEL

A POMDP model constructed in Section 3 can be too large to be solved efficiently. The state-of-the-art solver can only solve a POMDP of no more than a *few thousands* of states in a reasonable amount of time [23]. We now devise a novel method to first reduce the states and observations, and then build the POMDP model on top of them.

**Observations and intuitions.** The main idea of our state and observation reduction is as follows. The system states presented in Section 3 have fine granularities. The solution is not scalable. For example, there can be $20^n$ states, where $n$ is the number of blocks, when each block has 20 states (2, 5, and 2 possible values for $u_i$, $p_i$, and $r_i$, respectively). Moreover, we treat each data block equally when determining the states, i.e., each block maps to the same number of random variables.

We first observe that the fine granularities may be unnecessary since similar system states probably possess the same optimal actions. The notion of similarity will be made clear shortly. Secondly, the states of the data blocks required by the *current* query are more important than other blocks when deciding the action for the current query, although the states of other blocks do play a role in the overall policy as we try to optimize the long-term rewards for data blocks with update and query correlations.

The general technique we use is called *locality-sensitive hashing* (LSH) [19], with which we can hash *similar* states to the same new state. Thus, the number of new states can be kept manageable. However, we cannot simply use an existing method based on a single distance metric. This is because our state space is complex, and we combine set-similarity based LSH and Hamming-distance based LSH for the reduction, as detailed below.

**The algorithm.** The set of data blocks $R$ required by the current query may be a very small portion of all data blocks. Hence, if $R$ is treated in the same manner as other blocks, an approximation can easily lose significant information from $R$. However, $R$ is arguably more important as it directly contains information about the data blocks being considered for acquisition. Thus, we use different hash reduction techniques for (1) $R$ and (2) all blocks in general, and the final reduced state value is the concatenation of the two parts. Part (1) contains the $(u_i, p_i)$ information of the data blocks $R$ referenced by the current query. Since this is a set, we use *set similarity* based LSH (i.e., MinHash). Part (2) has a fixed number of bits (over all $n$ data blocks), for which we use Hamming distance based LSH.

We also need to reduce an *observation*, which includes the data blocks referenced by the next query and the $p_i$ values of data blocks referenced by the current query (if the action is *acquire*). As this information can also be regarded as a set, similar to part (1) of a state, we use *set similarity* based hash reduction. The algorithm is presented as COALESCESTATES.

We start with creating part (1) of the reduced state value. Lines 1-6 construct a set $R$ that contains the $(u_i, p_i)$ of all blocks required by the current query. $u_i$ is a Boolean value and we directly add $(i, u_i)$ into the set (line 4). $p_i$ is an integer level (e.g., one of 5 levels 1 to 5), for which we add $(i, 1), ..., (i, p_i)$ into $R$. This is so that closer $p_i$ values have less difference in the set $R$. In lines 7-12, we get the minimum hash values (over all elements of set $R$) of $k$ hash functions. An element of $R$ is represented in a binary string before it is hashed. We use a *universal hash function* family $h_{a,b}(x) = ((ax + b) \, mod \, p) \, mod \, m$ [25] (where $p$ is a large prime, $a, b < p$ are random integers, and $m$ is the number of hash bins). The selection of $k$ has to do with Theorem 2 below (we use $k = 3$ by default). This is the MinHash

[10] that happens to be consistent with the Jaccard similarity coefficients. In line 13, we hash the concatenation of $k$ minimum hash values above and reduce them into $l_1$ bits, which is part (1) of the new state value $\sigma$.

**Algorithm 1:** CoalesceStates$(s, o)$

```
   Input: s: a state in the original POMDP model,
   o: an observation
   Output: coalesced and reduced state σ and observation ω
   /* Construct a set for information of referenced
      blocks in s                                          */
1  R ← ∅
2  for each data block b_i ∈ 𝔻 do
3      if r_i = true in s then
4          R ← R ∪ (i, u_i)
5          for j ← 1 to p_i do
6              R ← R ∪ (i, j)
7  for i ← 1 to k do
8      min[i] ← MAX_INTEGER
9  for each element e ∈ R do
10     for i ← 1 to k do
11         if min[i] > h_i(e) then
12             min[i] ← h_i(e)
13 σ[1..l_1] ← h(min[1]...min[k]) mod 2^{l_1}
14 B ← an empty bit vector
15 for each data block b_i ∈ 𝔻 // n blocks in total
16  do
17     add u_i to B
18     for each probability level j do
19         if j ≤ p_i then
20             add 1 to B
21         else
22             add 0 to B
23 for i ← 1 to l_2 do
24     σ[l_1 + i] ← uniformly random bit from B
25 R ← ∅ // now coalesce observation o into ω
26 for each required data block b_i in o do
27     R ← R ∪ (i)
28 for each block b_i and its observed p_i in o do
29     for j ← 1 to p_i do
30         R ← R ∪ (i, j)
31 do lines 7-12
32 ω ← h(min[1]...min[k]) mod 2^{l_ω}
33 return σ and ω
```

Lines 14-24 create the $l_2$-bit part (2) of the new state. We first create a bit vector in which each block has the same number of bits. Then we get $l_2$ uniformly random bits from this bit vector, forming part (2) of the new state value $\sigma$. Lines 25-32 coalesce the observation $o$ into a smaller ($l_\omega$-bit) value $\omega$ in a similar fashion as part (1) of the state value. The difference is that we need to include information on which data blocks are required in the *next* query, and the observed update probability levels for blocks in the *current* query (if any). The latter (lines 28-30) is applicable if the action is *acquire*. Note that the $k+1$ hash functions in lines 11-13 and the random bit choices in line 24 must be kept the same amongst all invocations (i.e., all input states and observations) of CoalesceStates. In this way, we have a consistent coalescence for all states and observations.

Example 2. *Suppose there are 5 data blocks $b_1, ..., b_5$, and the current state—with format $(u_i, p_i, r_i)$—is $b_1(F, 4, T)$, $b_2(T, 2, F)$, $b_3(T, 1, T)$, $b_4(T, 1, F)$, $b_5(F, 5, F)$. That is, $b_1$ and $b_3$ are referenced by the current query ($r_i = T$); $b_1$ is not up-to-date ($u_1 = F$) and its update probability level $p_1 = 4$. In* CoalesceStates*, we first construct set $R$ (part 1) by adding information about $b_1$ and $b_3$ (lines 3-6): $\{(1, F), (1, 1), (1, 2), (1, 3), (1, 4), (3, T), (3, 1)\}$. Suppose $k = 3$ and we apply the hash functions to each element in $R$ (lines 7-12) and get $min[1] = 0101$, $min[2] = 1000$, $min[3] = 0100$. Then we get the first $l_1$ (say 5) bits of the coalesced state by $h(min[1]min[2]min[3])$ mod $2^5 = 10100$. Now we construct part (2) by iterating through each data block and constructing a bit vector: for $b_1$ we get 011110, for $b_2$ we get 111000, for $b_3$ we get 110000, etc. Next we randomly pick $l_2$ (say 3) bits 011 from this bit vector and append them to the first 5 bits we have constructed for part (1). Then the coalesced state $\sigma = 10100011$.*

Theorem 2. *In the* CoalesceStates *algorithm, consider two original POMDP states $s_1$ and $s_2$ that are dissimilar in that (1) their required-data-block-set state information $R_1$ and $R_2$ (lines 1-6) have a Jaccard distance at least $1 - \eta$, and (2) their all-block information has a Hamming distance at least $(1 - \theta)|B|$, where $|B|$ is the total number of bits in the all-block information $B$ (lines 14-22). Then the probability that $s_1$ and $s_2$ are coalesced to the same state is at most $(\eta^k + \frac{1-\eta^k}{2^{l_1}})\theta^{l_2}$, where $k$, $l_1$, and $l_2$ are as described above.*

Proof. Through separate invocations of CoalesceStates, $s_1$ and $s_2$ produce $R_1$ and $R_2$ in lines 1-6. For each of the $k$ hash functions, the probability that $R_1$ and $R_2$ have the same minimum hash value is exactly their Jaccard similarity $\frac{|R_1 \cap R_2|}{|R_1 \cup R_2|}$, following the MinHash principle [10]. In line 13, in order for the part (1) of $s_1$ and $s_2$ to generate the same value $\sigma[1..l_1]$, either (1) all $k$ minimum hash values are the same between $R_1$ and $R_2$, denoted as event $E_a$, or (2) $E_a$ does not occur, but all $l_1$ bits happen to be the same by chance, denoted as event $E_b$. From above, $Pr[E_a \cup E_b] \leq \eta^k + \frac{(1-\eta^k)}{2^{l_1}}$. Finally, for part (2) of $s_1$ and $s_2$ to generate the same value in lines 23-24, all $l_2$ bits must be the same, which has probability $\theta^{l_2}$. The theorem follows. □

LSH reduction maps multiple original POMDP states into one. Let the original POMDP solution be an optimal policy $\pi_0^*$ that gives the optimal action at each state. Let the reduced POMDP solution be an optimal policy $\pi_r^*$ that also gives the optimal action at each coalesced state. Thus, the only power loss of the reduced model is due to the function mapping of states—when two original states $s_1$ and $s_2$ have *different actions* in $\pi_0^*$, but the coalescing maps them to the same state $\sigma$, forcing them to have the same action in $\pi_r^*$. Theorem 2 exactly targets this scenario, and shows that dissimilar states in the original POMDP have *exponentially* low probability to be coalesced into the same state. That is, given two original states $s_1$ and $s_2$ above, by slightly (linearly) increasing $k$ or $l_1$ or $l_2$, the probability that $s_1$ and $s_2$ are coalesced to the same state decreases exponentially fast.

# 5. LEARNING THE REDUCED MODEL

To use a POMDP for solving the data acquisition problem, we first need to obtain its parameters, namely the *stochastic transition function $T$*, the *observation function $O$*, and the *reward function $R$*. We discuss how to learn the parameters for the reduced model after the state and observation coalescence. Once these parameters are learned, we feed them into

the POMDP solver which outputs an optimal "policy". This is done only *once*. Then at runtime, as each query comes in, we obtain the observation and feed it to a program, which looks up the policy and instantaneously advises us on the optimal action to take. We learn the $T$, $O$, and $R$ functions in two possible ways: in a *real run* with an arbitrary existing policy, or by *simulation*. In fact, these two ways can be run at the same time in parallel if one would like to learn faster than only using real-time queries. Our learning algorithm LearnParameters efficiently fills in the empirical probability distributions for the reduced states and observations.

**Algorithm 2:** LearnParameters($s_0, real\_run$)

**Input:** $s_0$: an initial state in the original POMDP model,
$real\_run$: whether this is based on a real run
**Output:** $T(\sigma, a, \sigma')$: the state transition function
$R(\sigma, a)$: the reward function
$O(a, \sigma', \omega)$: the observation function

```
1  s ← s_0
2  for m ← 1 to α do
3      for j ← 1 to α and each act ← {acquire, skip} do
4          s' ← s
5          if real_run then
6              set s'.r_i same as o.r_i
7          else
8              set s'.r_i based on a query model
9          if act = acquire then
10             for each i ∈ {i|s.r_i} do
11                 set s'.u_i to be true
12                 if real_run then
13                     set s'.p_i same as o.p_i
14                 else
15                     update s'.p_i based on an update model
16         for each s'.u_i not set in line 11 do
17             set s'.u_i ← false with probability s'.p_i
18         coalesce s, o, and s' to σ, ω, and σ', respectively
19         increment the counter for P̂(σ'|σ, act)
20         increment the counter for P̂(ω|σ', act)
21         if act = acquire then
22             R(σ, act) ← R(σ, act) − c_d({i|s.r_i})
23         else
24             R(σ, act) ← R(σ, act) − c_q(s, q)
25     s ← s'
```

The loop in lines 2-25 iterates through possible starting state $s$ (with an arbitrary initial $s_0$) as in the transition function $T(s, a, s')$. The loop in lines 3-24 explores the space of all possible ending states $s'$ under each action (*acquire* and *skip*). Our goal is to learn the empirical probability distributions in lines 19-20 (which correspond to $T(\sigma, a, \sigma')$ and $O(a, \sigma', \omega)$, resp.), as well as the average reward value based on lines 21-24. The number of loop rounds parameter $\alpha$ in lines 2-3 signifies a tradeoff between the time needed for learning and the accuracy of the functions learned. Using standard statistics (e.g., [9]), it is easy to determine a good value of $\alpha$ that gives tight confidence intervals, since the counter value of each bucket of the distribution follows a Binomial distribution. For example, if the reduced state space size is 256, then an $\alpha$ value of 2,000 (which is the default we use in experiments) will give a good accuracy, while the whole algorithm is still efficient. In addition, we use *add-one smoothing* [22] when learning $\hat{P}(\sigma'|\sigma, act)$ and $\hat{P}(\omega|\sigma', act)$, and we need to divide the $R(\sigma, act)$ final values by a total count to get the average rewards.

In line 6, $o.r_i$'s in the observation indicate which blocks are required by the next query; hence we set $s'.r_i$'s to be the same. If this is not a real run, in line 8, we randomly set $s'.r_i$'s based on a *query model*, which essentially specifies the distributions of data blocks being required by each query. For example, a simple query model just includes the reference probability of each data block. Then $s'.r_i$ is a sample from the distribution. Line 11 marks all blocks referenced by the current query as up-to-date if the current action is *acquire*. We also set the update probability level $s'.p_i$ to be consistent with the observation if it is a real run (line 13). If it is not a real run (line 15), we set $s'.p_i$ based on an update model, which specifies a block's $p_i$ change distribution in an observation. For example, a simple one is a uniform distribution with values chosen from the current $p_i$ and its two adjacent values. Line 17 sets $s'.u_i$ (that is previously true) to false with probability $s'.p_i$. This is exactly how the dynamically learned update probability $p_i$ influences the estimation of up-to-date status $u_i$ of a block.

Line 18 calls CoalesceStates in Section 4, and lines 19-20 increment the counters that are set up for the empirical probability distributions as discussed earlier. With $R(\sigma, act)$ initialized as 0, lines 21-24 sum up the rewards, which are the negative values of two types of cost: the cost of acquiring the referenced data blocks $c_d(\{i|s.r_i\})$ in line 22 or, if the action is *skip*, the query result error cost $c_q(s, q)$ in line 24. Since this is the offline learning phase which occurs infrequently, for $c_q(s, q)$, the algorithm actually acquires the data to figure out the ground-truth result and gets the error, from which it gets the reward, as discussed in Section 3.

# 6. A MODEL-FREE APPROACH

We study a different approach than POMDP for the data acquisition problem, using a *model-free* reinforcement learning method called Q-learning [15]. We do not need to learn the $T$, $R$, and $O$ functions as in Section 5 for POMDP. Hence it is easier to adapt to dynamic data update/query trends, without learning and re-learning the parameter functions.

**State Set.** Q-learning requires a *fully observable* environment. Our POMDP model has $(u_i, p_i, r_i)$ for each block in a state, where $u_i$ is hidden. Thus, we change a block's state information to $(m_i, r_i)$, where $m_i$ describes "the expected number of updates to $b_i$ since the previously observed update time". Clearly, $m_i$ can be obtained from the last observed update time $t_i$ and update rate $\theta_i$ (as in Theorem 1). Specifically, the expected number of updates $\mu_i$ is equal to $(t_0 - t_i)\theta_i$, where $t_0$ is the current time. Since $\mu_i$ itself may have many values, we partition it into ranges, and assign the range index to $m_i$. For example, $m_i = 1$ to 4 for $\mu_i \in [0, 1)$, $[1, 3)$, $[3, 6)$, and $[6, \infty)$, respectively.

The major idea of ModelFreeAcquisition is to first coalesce the original states described above into reduced states in a similar manner as Section 4, and then use Q-learning, which incrementally builds a table $Q[\sigma, a]$ that describes the best rewards (Q value) and action $a$ for state $\sigma$.

In line 1, we initialize the $Q[\sigma, a]$ array to all 0's, where $\sigma$ is a reduced state, $a$ is an action at that state, and $Q[\sigma, a]$ is the target value of doing action a in state $\sigma$—including the discounted rewards from all subsequent states under optimal actions. This is basically a dynamic programming approach: we incrementally and iteratively update the $Q$ values.

**Algorithm 3:** ModelFreeAcquisition($qstream$)

**Input:** $qstream$: a query stream to $\mathbb{D}$
**Output:** a sequence of decisions on data acquisition

1 initialize $Q[\sigma, a]$ to all 0's
2 $\sigma \leftarrow null$
3 **for** *each* $q \in qstream$ **do**
4  $R \leftarrow \emptyset$
5  **for** *each* $i \in \{i | r_i = true\ for\ q\}$ **do**
6  update $m_i$ based on $t_i$ and $\theta_i$
7  **for** $j \leftarrow 1$ *to* $m_i$ **do**
8  $R \leftarrow R \cup (i, j)$
9  hash $R$ to $\sigma'[1..l_1]$ as in lines 7-13 of CoalesceStates
10  **for** $k \leftarrow 1$ *to* $l_2$ **do**
11  $j \leftarrow random(0, 4n)$
12  $i \leftarrow j/4$; $m \leftarrow j\%4$
13  update $m_i$ based on $t_i$ and $\theta_i$
14  $\sigma'[l_1 + k] \leftarrow (m < m_i)\ ?\ 1 : 0$
15  **if** $\sigma \neq null$ **then**
16  increment $n[\sigma, a]$
17  $Q[\sigma, a] \leftarrow$
 $Q[\sigma, a] + \alpha(n[\sigma, a])(r + \gamma max_{a'} Q[\sigma', a'] - Q[\sigma, a])$
18  $\sigma, a, r \leftarrow \sigma', argmax_{a'} f(Q[\sigma', a'], n[\sigma', a']), r'$
19  use action $a$ for $q$

Lines 4-9 create an $l_1$-bit MinHash $\sigma'[1..l_1]$ for the original state information $m_i$ in each block in $R$. Then analogous to CoalesceStates, we add $l_2$ bits chosen uniformly at random from the state of all blocks. In line 11, we choose a bit uniformly at random from $4n$ bits. Here we assume each of the $n$ data blocks takes up four bits for the four $m_i$ ranges (any other number of ranges could be used here). Then $i$ and $m$ in line 12 are the data block index and a random bit in $m_i$, respectively. Line 14 sets one bit in $\sigma'$ to 1 if $m < m_i$ (or else 0). Effectively, this is equivalent to treating $m_i$ as in unary, padded with 0 to the right. For instance, if $m_i = 2$, it chooses a random bit from 1100 ($m$'th bit from left). In line 16, we increment the count of visiting state $\sigma$ with action $a$. Recall that $\sigma, a$ is the previous state and action of $\sigma', a'$. In line 17, we adjust the previous estimate of $Q[\sigma, a]$ due to the update of $Q[\sigma', a']$. Line 17 gives the adjustment of $Q[\sigma, a]$ based on the current estimate of the next state's optimal action's $Q$ value. The $0 < \gamma < 1$ in line 17 is the discount factor for future rewards, while $r$ is the immediate reward received at this state $\sigma$ under the current action $a$. Then $r + \gamma max_{a'} Q[\sigma', a']$ is the updated estimate of $Q[\sigma, a]$ based on the best action $a'$ and $Q$ value presently selected for the next state $\sigma'$. The $\alpha(n[\sigma, a])$ is the *learning rate* function, which decreases as the number of times a state has been visited ($n[\sigma, a]$) increases [22]. Like [22], we use $\alpha(n[\sigma, a]) = 60/(59 + n[\sigma, a])$. Essentially line 17 adjusts the $Q[\sigma, a]$ estimate according to the learning rate function, which ensures that $Q[\sigma, a]$ will converge to the correct value.

The $f(Q[\sigma', a'], n[\sigma', a'])$ function in line 18 is called the *exploration function* [22], which determines how greed (preference for high values of $Q$) is traded off against curiosity (preference for low values of $n$—actions that have not been tried often). $f$ should be increasing in $Q$ and decreasing in $n$. Similar to [22], we use: $f(Q[\sigma', a'], n[\sigma', a']) = \begin{cases} 0 & \text{if } n[\sigma', a'] < 5 \\ Q[\sigma', a'] & \text{otherwise} \end{cases}$. This has the effect of trying each action-state pair at least five times, since our $Q$ values are always non-positive (negative value of costs). Lines 18-19 set the current state, best action chosen, and immediate reward and take this action for the current query $q$. Finally, the algorithm loops back to take the next query, and the process of dynamically updating $Q$ table proceeds in the same fashion.

(a)

| Q State \ Action | Acquire | Skip |
|---|---|---|
| 0 | −1 | −3 |
| 1 | −7 | −50 |
| 2 | −15 | −20 |
| 3 | −45 | −10 |

| n State \ Action | Acquire | Skip |
|---|---|---|
| 0 | 1 | 1 |
| 1 | 4 | 5 |
| 2 | 2 | 3 |
| 3 | 5 | 5 |

(b)

| Q State \ Action | Acquire | Skip |
|---|---|---|
| 0 | −1 | −3 |
| 1 | −12.2 | −50 |
| 2 | −15 | −20 |
| 3 | −45 | −10 |

| n State \ Action | Acquire | Skip |
|---|---|---|
| 0 | 1 | 1 |
| 1 | 5 | 5 |
| 2 | 2 | 3 |
| 3 | 5 | 5 |

**Figure 3: The $Q$ and $n$ tables before and after an iteration of update, shown in (a) and (b), respectively.**

Example 3. *Suppose* ModelFreeAcquisition *has run for a while, and at the moment we have a snapshot of $Q$ and $n$ tables in Figure 3(a). At this iteration, we calculate the state $\sigma' = 1$. In line 18, we set $\sigma = 1$, and set action $a$ based on which $a'$ gives the maximum $f$ value. Since $f(Q[1, acquire], n[1, acquire]) = 0$ and $f(Q[1, skip], n[1, skip]) = -50$, we have $a \leftarrow acquire$ in line 18, and the current query is answered with action "acquire" in line 19. We then go to the next iteration, and suppose this time we get $\sigma' = 3$ after lines 4-14. Now in line 16, we increment $n[1, acquire]$ from 4 to 5 as shown in Figure 3(b). In line 17, we update $Q[1, acquire]$ to be $-7 + \frac{60}{64}[-3 + 0.95 \times max_{a'} Q[3, a'] - (-7)] = -12.2$, where $max_{a'} Q[3, a'] = -10$, as indicated in the dashed circle in Figure 3(a). The updated $Q$ value is also shown in the circle in Figure 3(b).*

## 7. BUDGET CONSTRAINTS

We consider a budget rate constraint on how fast one can spend resources. That is, the query processor $\mathcal{P}$ must not pay more than $B$ dollars in any window of size $w$ (either time windows or count windows based on the number of queries). This is a practical problem. For example, there can be a practical constraint on how much money one can spend on data acquisition in a data market. Similarly, in sensor networks or other networked environments, due to hardware limits (e.g., sensor power consumption or network transmission rates), there may be a constraint on how fast data can be acquired.

It turns out that both POMDP and Q-learning can be easily modified to solve this problem. The key idea is that we use a small number of bits (e.g., 2) in the reduced state to keep track of the *spending level* (i.e., money spent in the current window). One of the spending levels is "no money left for the current window" (corresponding to the state that the "acquire" action is disallowed), and other spending levels correspond to different amounts (intervals) of money left. Let the maximum budget be $B$ per window. For example, suppose there are 4 spending levels. For a state $\sigma$, denote the average cost of a query as $|q|$. Typically $B \gg |q|$. Then the four spending levels correspond to the expense in the current window being $[0, \frac{B-|q|}{3})$, $[\frac{B-|q|}{3}, \frac{2(B-|q|)}{3})$, $[\frac{2(B-|q|)}{3}, B - |q|)$, and $[B - |q|, B]$, respectively, where the last spending level disallows *acquire* as that would be over the budget. The modification to the POMDP approach is as follows:

(1) Append $\beta$ bits (e.g., $\beta = 2$) to a reduced state $\sigma$ and a reduced observation $\omega$. The added bits keep track of $2^\beta$ possible budget levels for the current window.

(2) During both LearnParameters (either a real run or simulation) and the online execution of the POMDP model, we maintain the exact amount of money (cost) spent for acquiring data in the current window, and translate it into the spending levels in reduced observations and states.

The above changes to the algorithms essentially "encode" the budget constraint into the POMDP model, and hence the solution will optimize the target function subject to the constraint. Likewise, we can modify ModelFreeAcquisition in the same manner to handle a budget constraint.

# 8. EXPERIMENTS

## 8.1 Datasets and Setup

We use two real-world datasets: **(1) Traffic dataset.** It is produced by loop detector sensors located on major highways of the Greater Toronto Area [5]. The data is collected over time, with each sensor continuously outputting events. Each event contains information about the number of cars detected during the current interval, the average speed, and the occupancy rate, etc. **(2) Mobility dataset.** It includes a set of continuous traces compiled in 2007 [7]. The traces contain battery usage data for laptops. Apart from battery-usage data, the traces also contain data on whether the machine was on AC, Internet connectivity, CPU utilization, disk space, and idle time for the laptop users, etc.

In Traffic data, we treat the sequence of data from each sensor as a block, while in the Mobility data, the sequence from each laptop is a block. For both datasets, there are some data collecting costs: the infrastructure for loop detector sensors can be expensive to set up and maintain; collecting real-time mobile device data from users can be disturbing and should be kept to minimum. We use the following queries: $Q_1$ (Traffic data) The current travel speeds of a set of roads; $Q_2$ (Mobility data) The remaining battery capacity of a set of laptops; $Q_3$ (Mobility data - aggregate) The average remaining battery of a set of laptops; $Q_4$ (Traffic data - join) The fraction of time in which two roads have close travel speeds ($\pm 10 km/h$) within three hours in their respective sequences; $Q_5$ (Traffic data - aggregate) The total numbers of vehicles per hour for each region; $Q_6$ (Mobility data - join & aggregate) The maximum difference of remaining battery capacity of any two laptops; $Q_7$ (Traffic data - subquery) The speeds of the roads which are higher than the average speed of all roads; $Q_8$ (Mobility data - aggregate) The maximum idle time of the laptops whose *ID*'s are in a certain range. $Q_1$, $Q_4$, $Q_5$, and $Q_7$ are for the Traffic dataset while $Q_2$, $Q_3$, $Q_6$, and $Q_8$ are for the Mobility dataset.

For POMDP solver, we use a state-of-the-art one [2] that implements the SARSOP algorithm [18]. We $modify$ the POMDP simulator code from [2] such that, upon taking an action suggested by the simulator: (1) our program follows the action to either acquire data or use the existing data in $\mathbb{D}$ to answer the query; (2) our program informs the simulator of a new observation. Accordingly, the simulator will compute the belief state and advise us on the next action. This process continues. The POMDP solver and simulator [2] are implemented in C++; so is our modification to the simulator as described above. We implement all our algorithms described in this paper in Java, and use JNI to communicate with the POMDP simulator. All experiments are performed on a machine with an Intel Core i7 2.50 GHz processor and an 8GB memory, running Ubuntu Linux.

## 8.2 Experimental Results

**POMDP approach.** The first step is to learn the functions $T$, $O$, and $R$. We measure the learning time under different parameters. All results are based on the average of at least five runs. In Figure 4, we vary the number of blocks from 100 up to 100,000, while fixing the parameters $l_1 = 5$, $l_2 = 3$, and $l_\omega = 5$ ($l_1$ and $l_2$ are the lengths of the two parts of a reduced state, and $l_\omega$is the length of a reduced observation). Note that each data block corresponds to a number of data sequences. Hence, the number of blocks need not be large while the data volume is very large.

We use the default parameters $l_1 = l_\omega = 5$ and $l_2 = 3$. We use a simulated run for Figure 4 since this does not change the learning time and since we need to vary $n$. To study how well our approaches adapt to the dynamic fluctuations of update rates at different blocks, we let each data block's update probability $p_i$ be randomly chosen from [0,1] every two seconds. The *query model* reflects the locality of reference: when a query accesses a range of data blocks, query $i$ has probability 0.5 to start from a random block $b$ used in query $i-1$, and query $i$ references a contiguous chunk of blocks centered around block $b$ with a random size. We will also vary this correlation probability and study its impact on the resulting policy in a later experiment (Figure 8). Figure 4 shows the learning times under various numbers of blocks. For instance, when the number of blocks is $10^5$, the learning time is 32 seconds. This is consistent with the fact that LearnParameters time is proportional to the average number of referenced blocks in a query.

We then measure the learning time by varying $l_1$ from 3 to 8, shown in Figure 5. For both datasets, the learning time increases exponentially with $l_1$. This is because $l_1$ is the number of bits, and the reduced state space size increases exponentially with $l_1$. The whole learning time is proportional to the reduced state space size. This works in practice as $l_1$ is a small number, and the learning is offline and infrequent. After learning the parameters, we feed them to the POMDP solver, which typically requires a long time to converge to an exact solution. However, it is an anytime algorithm; we can stop it any time and it will output a policy with an upper bound $UB$ and a lower bound $LB$ of the target long-term reward (we use a discount factor $\gamma = 0.95$). We say that the *precision gap* of the solver at this moment is $\frac{|UB-LB|}{|UB|}$. When the precision gap is 0, the solution is exact; a value farther away from 0 indicates a less accurate solution. In all parameter settings of our experiments, we let the solver run for 3 minutes, and find that the *precision gap* is no more than 0.04. In subsequent experiments, we will evaluate the effectiveness of these output policies.

In Figure 6, we first examine the query result accuracy for the various types of queries $Q_1$ to $Q_8$ (described in Section 8.1), following the POMDP policy as well as the model free Q-learning approach (more results on Q-learning will be detailed later). The y-axis shows the average *relative* error $|\frac{actual\_value - true\_value}{true\_value}|$, where *actual_value* is from our algorithms and *true_value* is the ground-truth result using all data from datasets. $Q_1$, $Q_4$, $Q_5$, and $Q_7$ are for the Traffic dataset while $Q_2$, $Q_3$, $Q_6$, and $Q_8$ are for the Mobility dataset. Figure 6 shows that they all have very small errors no more than 0.025. Model free Q-learning has slightly

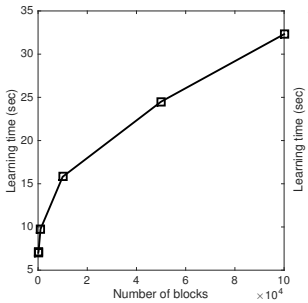


Fig. 4 Learning time vs blocks

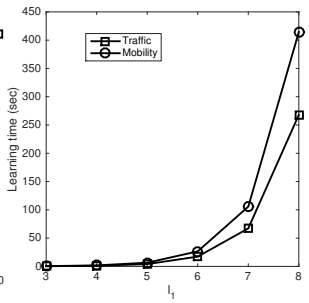


Fig. 5 Learning time vs $l_1$

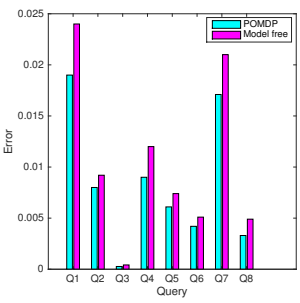


Fig. 6 Accuracy of queries

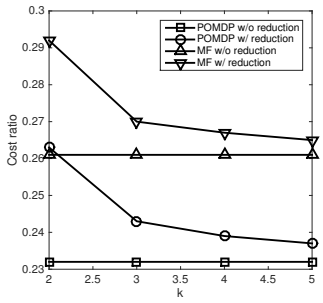


Fig. 7 Cost ratio vs reduction

greater errors as it makes no assumption about the model or the solution (and its advantage is the quicker adaptivity to changes). $Q_3$ has the smallest error because it is a sum/average query over many data items and the errors (if any) in individual data items are reduced (or canceled out) when they are added together.

To study the effectiveness of the policies generated from our models, we first define a *baseline*. It examines the cost of action "acquire" vs. "skip" over the true state of the system and chooses the one with a smaller cost. Note that the baseline is given *more power than the actual query processor* since it knows the *true* state (i.e., the query result error with current data and hence the true cost), just for experiments. Hence, the baseline is an idealized policy with *auxiliary information*. However, it does not do sequential planning over all queries, which is the distinctive goal of our work. The baseline establishes a common ground for us to compare different approaches (POMDP and model-free), and various parameter settings. We also compare with two generally weaker, more naïve baselines in some following experiments (Figures 9 and 10). As discussed earlier, the costs (i.e., negative values of rewards) consist of two parts, the *data acquisition* cost and the *query result error* cost, which we always know in our controlled experiments. We calculate the ratio of the cost of running the policy output from our model and that of a baseline policy.

In Theorem 2 and the subsequent discussion, we analyze that the power loss (i.e., error) due to our state reduction can be made exponentially small with respect to $k$ (number of hash functions), $l_1$, and $l_2$. We now experimentally verify this point. When the problem size, the number of blocks $n$, is very small, we can afford to use POMDP and Q-learning without the state reduction. In Figure 7, we show the cost ratios (w.r.t. the baseline with auxiliary information) of both the POMDP and model-free (MF) approaches with and without the state reduction for $n = 3$ using the Traffic dataset (the Mobility dataset shows similar results). We vary the parameter $k$ between 2 and 5, while keeping other parameters at default values. We can see that as $k$ increases, the cost ratio with reduction approaches that without reduction precipitously, for both POMDP and MF (we examine MF more in later experiments). In general, the reduction performs well; we use as default $k = 3$ and study $l_1$ and $l_2$ in subsequent experiments.

We next study the effect of varying query sequence. We have mentioned earlier that, by default, we use an adjacent-query block correlation probability $p_c = 0.5$, and queries are chosen uniformly at random from the templates $Q_1$ to $Q_8$. We now vary these parameters. For the query sequence, a key insight is that what affects data acquisition decision is each query's result *sensitivity to data updates*. This is because lower sensitivity of query result to data perturbation/updates will cause our models to acquire data less often, and hence the overall cost is smaller. The result in Figure 6 gives evidence of the sensitivities of each query to updates. The smaller the error is, the less sensitive that query is to updates (intuitively, recall that our reward/utility function "balances" between the data acquisition cost and result accuracy cost). Thus, in addition to the random order, we also examine queries issued in *ascending* sensitivity order (and an ascending cycle repeats after the maximum sensitivity query has been issued); we examine queries in *descending* sensitivity order too. Figure 8 shows the result for the POMDP and model-free (MF) approaches with $p_c$ varying between 0.1 and 0.9, for the Mobility dataset (the result of the Traffic dataset has a similar trend). First, we can see that as $p_c$ increases, the advantage of our approaches is more significant. The reason is that more correlation of queries benefits more from our model's sequential planning which is not in the baseline. Second, our approaches perform slightly better under ascending or descending query sensitivity orders than random order, especially for high $p_c$. Again, this is because more regularity in the query sequence will show the advantage of sequential planning. Thus, in what follows, we only report the results with random order. Finally, regardless of input query sequence properties, our stochastic model optimizes a mathematical function and achieves the best it can of the situation.

Next, in Figure 9, we report the cost ratio for the policies produced by the POMDP models constructed under different $l_1$ values over the baseline with auxiliary information as described above (denoted as $b^*$ in Figure 9), as well as two other generally weaker baselines (denoted as $b1$ and $b2$). These two new baselines do not know the ground truth of query result errors. They have to estimate the number of outdated blocks among the required ones $R$ for the query and make the decision. In $b1$, if the expected number of outdated blocks in $R$ is more than the expected number of up-to-date ones, we acquire $R$. In $b2$, we make such decision for each block in $R$ independently: if its probability of being outdated is greater than 0.5, we acquire it. Likewise in Figure 10, we vary $l_2$ from 0 to 4 bits (while using the default $l_1 = l_\omega = 5$). First, we see that $b^*$ performs better than $b1$ and $b2$ (i.e., it has lower cost, hence higher cost ratios for our models). This is because $b^*$ makes more informed decisions with the ground-truth information of query result errors and costs. $b2$ performs better than $b1$ because it performs actions with a finer granularity by treating each block individually. From now on, for clarity, we only discuss the results w.r.t. the baseline with auxiliary information $b^*$.

In Figure 9, for both datasets, the cost ratio decreases from around 0.38 to around 0.2 as $l_1$ increases from 3 to

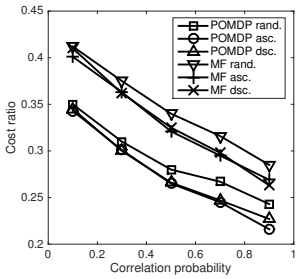


**Fig. 8 Query sequence variations**

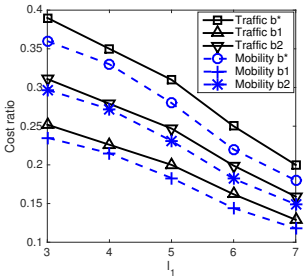


**Fig. 9 Cost ratio vs. $l_1$**

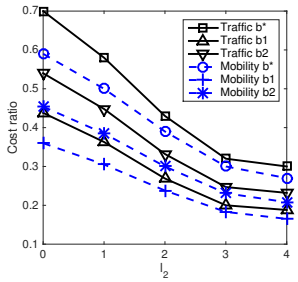


**Fig. 10 Cost ratio vs. $l_2$**

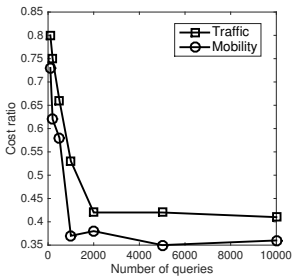


**Fig. 11 Model-free, cost ratio**

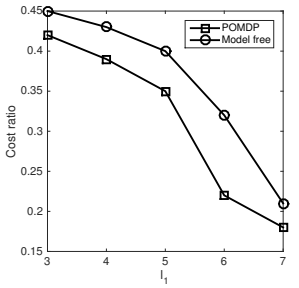


**Fig. 12 Traffic dataset**

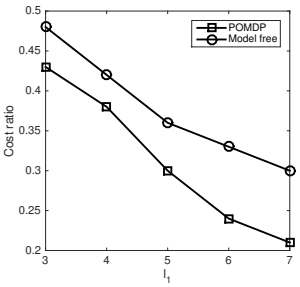


**Fig. 13 Mobility dataset**

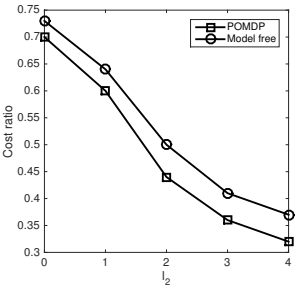


**Fig. 14 Traffic dataset**

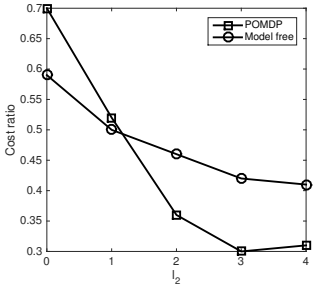


**Fig. 15 Mobility dataset**

7. This is because a greater $l_1$ results in more states in the model, which makes the action policy more fine-grained and accurate. Of course, the tradeoff for greater $l_1$ is longer times for learning and solving the model. Note that looking up a POMDP policy for the action to take is instantaneous (similar to the baseline) and insignificant (compared to answering queries). The same is true for the model-free Q-learning. Thus we do not need to report their execution times. Figure 9 shows that POMDP combined with LSH is effective. In Figure 10, the cost ratio decreases as $l_2$ increases. This demonstrates that the second part of the reduced state, the locality sensitive hashing of the "big environment" (all blocks in general) is indeed necessary—having 0 bit $l_2$ only gives a cost ratio of over 0.6. On the other hand, interestingly, the curve levels off at some point. $l_2 = 3$ and $l_2 = 4$ have very close cost ratios. This is because the information (on blocks used in the current query) in part 1 of the reduced state ($l_1$ bits) has a more direct and crucial role in the optimal policy, while part 2 of the reduced state has some impact on the policy in the long term.

**Model-free approach and budget constraints.** We now examine our model-free Q-learning approach, which does not have offline learning, but does have an initial "warm-up" period, in which the Q-table values gradually converge to the optimal. In Figure 11, we measure the cost ratio (compared to the baseline), as the number of queries increases. This experiment clearly shows the warm-up period. For the Traffic data, after about 2,000 queries, the cost ratio levels off at the optimal value, while the warm-up period is around 1,000 queries for the Mobility data. It is thus interesting to compare the cost ratio of Q-learning after the warm-up with that of the POMDP. Figure 12 shows the result for the Traffic data under various $l_1$'s. The POMDP's cost ratio is slightly less than the model-free's. Similarly, the result of the Mobility data is shown in Figure 13, where the cost ratio difference is more noticeable. We then compare their cost ratios under different $l_2$'s. The results are shown in Figures 14 and 15 for the two datasets respectively. Like in POMDP, the second part of the reduced state (LSH of the all-block environment) also plays a role in the model-free approach for the optimization of long-term rewards. Figures 12-15 show that, overall, our first approach's policy is slightly more effective than our second approach.

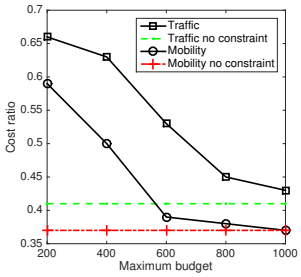


**Fig. 16 Budget & Q-learning**

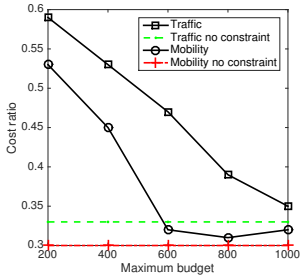


**Fig. 17 Budget & POMDP**

We finally study budget constraints. First we examine the Q-learning approach, and vary the maximum budget in the range of 200 to 1,000 (using parameters $\beta = 2$ bits and $w = 10$ queries). The results are shown in Figure 16. For comparison, we also draw two horizontal constant lines in the figure to show the cost ratios without any constraints on maximum budgets. As the maximum budget increases, the cost ratio of our algorithm decreases, eventually staying at the level of the cost ratio without any constraints. This is because, while the budget constraint may reduce part 1 of the costs (acquiring data), it would increase part 2 of the costs (penalty for outdated data) even more. The optimal solution without the constraint cannot be worse than the optimal solution with the constraint. The result for POMDP is in Figure 17, where the impact to the solution with budget constraints is the same. Another interesting fact evident in both figures is that the curves for the Mobility dataset level off to the lowest cost ratio faster (at the maximum budget value around 600) than the Traffic dataset. This indicates that the optimal policy for Mobility data without constraints has a lower maximum expense (on acquiring data) in the sliding windows.

**Summary.** The experimental results show that our approach of using POMDP combined with LSH requires a rea-

sonably short time for learning parameters and for solving the POMDP model under practical parameter settings. The resulting policy from this approach is also effective. Both parts of the reduced state are necessary for the effectiveness of the resulting policy, with the first part ($l_1$ bits) being more crucial. The model-free approach has the advantage of its simplicity and no overhead for learning and solving a model, although it does have a warm-up period for it to be effective. The final solution of approach 1 is slightly more effective than that of approach 2. Finally, both approaches can be easily modified to handle additional constraints such as maximum budgets, due to the good extensibility.

## 9. OTHER RELATED WORK

The most closely related work ([21] and [12]) is discussed in Section 1; we survey other work here. To our knowledge, sequential data acquisition problem has not been studied before. The cost of data acquisition has been long realized, e.g., in data market work in both academia [17] and industry [3, 1]. Feder et al. [14] study the problem of computing the $k$'th smallest value among uncertain variables $X_1, ..., X_n$, which are guaranteed to be in specified intervals. It is possible to query the precise value of $X_j$ at a cost $c_j$ ($1 \leq j \leq n$). The goal is to pin down the $k$'th smallest value to within a precision $\sigma$ at a minimum cost. By contrast, our work aims at general query types rather than the specific $k$'th smallest value; more importantly, we study decision making over a sequence of queries instead of just one query.

Fan et al. [13] study the problem of determining *data currency*. They assume a database has multiple versions of a tuple, but there are no valid timestamps. The problem is to determine which tuple has the *most current* attribute value. They give a number of complexity results in the presence of copy relationships among data sources. This is a different problem than ours. POMDP and Q-learning have been used in diverse areas. In [23], POMDP is used in collision avoidance for unmanned aircrafts. As reported in [23], the state-of-the-art POMDP solver can only solve POMDP with the number of states up to *a few thousands* to generate acceptable policies in a reasonable amount of time. Thus, to scale up, we propose the additional usage of LSH.

## 10. CONCLUSIONS AND FUTURE WORK

We formalize the sequential data acquisition problem motivated by many applications. We use two approaches of reinforcement learning in a novel way, each of which has its advantages. To make the solution scalable, we use tailored locality sensitive hashing. Our solution framework is also fairly extensible to incorporate additional constraints and cost/action models. Our comprehensive experiments demonstrate the feasibility and effectiveness of our approaches. The model-free approach adapts to dynamic environments. For the POMDP approach, we may incorporate in the state set crucial dynamic parameters such as update rates as we already do; then the remaining model/policy is relatively stable, and we can re-learn the model every once in a long while. Or we can treat it as a multi-armed bandit [24] exploitation vs. exploration problem (i.e., using existing policy vs. learning a new policy), and use the adaptive epsilon-greedy solution there. We plan to study this in the future.

**Acknowledgment**. This work was supported in part by the NSF, under the grants IIS-1149417, IIS-1319600, and IIS-1633271.

## 11. REFERENCES

[1] Aggdata. http://www.aggdata.com/.
[2] APPL POMDP solver download page http://bigbird.comp.nus.edu.sg/pmwiki/farm/appl/.
[3] Azure data market. https://datamarket.azure.com/.
[4] http://graphical.weather.gov/xml/.
[5] http://msrg.org/datasets/traffic.
[6] https://alerts.weather.gov/.
[7] http://traces.cs.umass.edu/index.php/power/power.
[8] D. Agrawal and others (21 authors). *Challenges and Opportunities with Big Data. 2012. http://cra.org/ccc/docs/init/bigdatawhitepaper.pdf.*
[9] C. M. Bishop. *Pattern recognition and machine learning*. Springer, 2006.
[10] A. Broder, M. Charikar, A. Frieze, and M. Mitzenmacher. Min-wise independent permutations. *J. of Computer & Sys. Sciences*, 2000.
[11] P. J. Denning. The locality principle. *Communications of the ACM*, 2005.
[12] A. Deshpande, C. Guestrin, S. R. Madden, J. M. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *VLDB*, 2004.
[13] W. Fan, F. Geerts, and J. Wijsen. Determining the currency of data. *ACM Trans. Database Syst.*, 2012.
[14] T. Feder, R. Motwani, R. Panigrahy, C. Olston, and J. Widom. Computing the median with uncertainty. In *ACM STOC*, 2000.
[15] L. Kaelbling et al. Reinforcement learning: A survey. *J. of Artificial Intelligence Research*, 1996.
[16] L. Kaelbling et al. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 1998.
[17] P. Koutris, P. Upadhyaya, M. Balazinska, B. Howe, and D. Suciu. Toward practical query pricing with QueryMarket. In *SIGMOD*, 2013.
[18] H. Kurniawati, D. Hsu, and W. S. Lee. SARSOP: Efficient point-based POMDP planning by approximating optimally reachable belief spaces. In *Robotics: Science and Systems*, 2008.
[19] J. Leskovec, A. Rajaraman, and J. Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2014.
[20] M. C. Nussbaum et al. *Aristotle's De Motu Animalium: Text with translation, commentary, and interpretive essays*. Princeton University Press, 1985.
[21] C. Olston, B. T. Loo, and J. Widom. Adaptive precision setting for cached approximate values. *ACM SIGMOD Record*, 2001.
[22] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2003.
[23] S. Temizer et al. Collision avoidance for unmanned aircraft using Markov decision processes. In *AIAA Guidance, Navigation, and Control Conference*, 2010.
[24] R. Weber. On the Gittins index for multiarmed bandits. *Annals of Applied Probability*, 1992.
[25] P. Woelfel. Efficient strongly universal and optimally universal hashing. In *Mathematical Foundation of Computer Science*. 1999.