Using Delay to Defend Against Database Extraction[⋆]

Magesh Jayapandian, Brian Noble, James Mickens, and H.V. Jagadish

Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122
{jmagesh,bnoble,jmickens,jag}@umich.edu

Abstract. For many data providers, the "crown jewels" of their business are the data that they have organized. If someone could copy their entire database, it would be a competitive catastrophe. Yet, a data provider is in the business of providing data, so access to the database cannot be restricted entirely. How is the data provider to permit legitimate access to users who request access to small portions of the database while protecting the database from wholesale copying?

We suggest that delay can be used for this purpose. We show, under reasonable assumptions, that it is possible to slow down the copying of the entire dataset by an arbitrary amount ensuring that queries that return a significant portion of the database introduce a delay that is orders of magnitude higher than that for legitimate user queries. We then consider issues of change, and show, under reasonable assumptions of rates of change, how to limit access so that the voyeur is guaranteed never to have a complete up-to-date dataset. We also present several extensions of these two major results.

We have implemented our technique on a commercial relational database, and we present numbers showing that the analytically expected delays are indeed observed experimentally, and also that the overheads of implementing our scheme are small.

1 Introduction

There are many information providers on the web, providing travel-related information, weather forecasts, directory look-up services, coverage of health-related topics, discographies, and so on – for almost any subject you desire, there is likely to be some provider who has invested time and effort to compile a database on the topic. Most of these providers have invested this time and effort with a business reason: even if they do not charge users for each look-up, they may rely upon user traffic for advertising revenues, for referral commissions, or as goodwill to attract customers to related fee-based services.

[⋆] Supported in part by NSF under grants IIS-0219513 and CCR-0208740

1.1 The Problem

Consider an attacker who wishes to steal information from such a provider, for instance to set up a business in competition. This attacker could attempt to hack into the provider's system – such attacks are beyond the scope of the current paper. Here we assume that the provider's computer systems, database, web server, etc. are appropriately protected from system invasion. (There are also many security threats other than data theft – such attacks also are outside the scope of this paper). The attacker still has the "front door" available – the information provider is in the business of providing information from the database to legitimate users, so the attacker only has to masquerade as a legitimate user to gain all the information in the database. To prevent such attacks, most information providers restrict the amount of information that can be queried in one request – users must ask very selective queries. However, such restrictions are easy to overcome – the attacker could trivially construct a robot that repeatedly asks slightly different selective queries whose union is the entire database. Robots can either adopt a *brute force* approach or use available knowledge to generate valid attribute values to pose queries. Such robots are hard to detect especially since each individual query is no different than one a genuine user might make.

1.2 Our Solution

Our scheme involves the inclusion of a strategically computed delay with every data item present in the provider's database. The delays are variable and are based on the popularity of the individual items in the database. They are computed in such a way that, without burdening legitimate users with too high[4] a wait time, they can force an attacker to wait an extremely long time to retrieve sufficient data to recreate the database.

1.3 Contributions

We begin by presenting the basic scheme, assigning delays to tuples according to their popularity; popular items have short delays, while unpopular items have long ones. A simple analysis shows that, for popularity distributions that follow a power law, typical users should expect modest delays, while an adversary attempting to extract the entire database faces delays many orders of magnitude larger. This is true even after capping the maximum possible delay at some value that legitimate users would find tolerable. Such distributions need not be known in advance, and can change over time. Likewise, the cost of computing delays can be kept reasonable.

The success of this scheme depends on access patterns with sufficient skew. If all items have similar popularity, our scheme would assign each of them approximately the same delay. If these delays are small, an adversary bent on extraction would not be penalized sufficiently. On the other hand, if these delays are large, legitimate users would suffer delays beyond their tolerance.

Instead, one can leverage the fact that most databases are updated frequently. We can impose small delays on frequently-updated items, but larger delays on items that are not updated as often. If these update rates exhibit skew, legitimate access to this database will tend to have low delays and fresh data, while an attacker will find that extracted copies of most of the data will always be stale, since the delay incurred in retrieving all of them is collectively greater than the update periods of several of them individually.

We evaluate the success of our approach with a variety of experiments. The first uses real datasets to show that skewed access patterns do indeed produce the desired results. Users can expect to suffer delays on the order of a few milliseconds, while adversaries are faced with nearly 90% of the costs one would achieve by penalizing each query with the maximum individual delay. We also show synthetic benchmarks that explore the range of costs and benefits one might expect if delay is assigned based on access rate as well as based on update rate. Finally, we show that even an untuned implementation of this scheme imposes overheads of 20% in the worst case.

2 The Core Proposal

In nearly any dataset, some items are more popular than others. Legitimate users tend to access popular items more frequently than unpopular ones, but an adversary bent on extraction must eventually request every element in the set. We can use these skewed preferences to assign small (or zero) delays to popular items, but large delays to unpopular ones. Such an assignment does not often penalize legitimate users, but it imposes substantial, frequent penalties to an attacker bent on extraction. Any dataset with a known, skewed popularity distribution is amenable to this technique. What we are leveraging is that the query distribution of an attacker is different from the query distribution of legitimate users. If the legitimate query workload has a uniform distribution over the data elements, then the core proposal described here will not work: however, we may still be able to exploit skews in data updates as we shall describe in Section 3.

For concreteness, in this paper, we will assume a relational model, with each tuple being a unit data element for retrieval. We will assume a query load comprised purely of selection queries against this relation, and assign to each tuple a popularity score reflecting how frequently it is present in the query result. To each tuple retrieval, we assign a delay that is inversely proportional to the popularity of the tuple. With appropriate, often trivial, modifications, our scheme can also be applied to other data models, query models, and delay models.

2.1 A Simple Zipfian Analysis

A Zipf distribution is frequently used to model skew, and occurs in a wide variety of settings. It was originally observed in English word choice [22], and it holds for Web [8] and streaming media [11] workloads. In a Zipf distribution – sometimes also called a *power law* – the i^{th} most popular object is requested at a frequency proportional to $i^{-\alpha}$, where α is called the *Zipf parameter*. This distribution has been widely reported in the database literature as well, and is probably applicable to many data sources on the web. Though our technique is not dependent on the specific form of the query distribution, it is convenient for purposes of analysis to focus on one distribution, and the Zipf distribution is a natural choice.

For purposes of this analysis, we also simplify the query model slightly – we assume that each query to the database eventually results in exactly one tuple. (This is not a major restriction, since a query that returns multiple tuples can simply be considered the aggregate of multiple simple queries that return one tuple each.) The delay, d, for which the database engine pauses before yielding the i^{th} most popular tuple is

$$d = \frac{1}{N}\left(\frac{i^{\alpha+\beta}}{f_{max}}\right) . \tag{1}$$

Here N is the number of tuples in the relation, f_{max} is the frequency with which the most popular item is requested, and α is the Zipf parameter of the underlying popularity distribution. The constant β is chosen to balance the desired penalty imposed on an extraction attack with the undesirable delays to legitimate users. If an adversary were then to pose a sequence of queries to extract the complete relation, the total delay incurred would be

$$d_{total} = \sum_{i=1}^{N} d(i) = \frac{1}{N f_{max}} \sum_{i=1}^{N} i^{\alpha+\beta} . \tag{2}$$

It is easy to see that an adversary must face longer delays with higher β values. Legitimate users, on the other hand, would expect to see median delays, d_{med} for the typical request. For skewed distributions, we believe that a quantile metric such as the median is more representative and fair than other statistical measures such as mean, variance and standard deviation, which often fall victim to outliers in the data, that are commonly found in Zipfian distributions. We can compute the asymptotic rank of the median frequency using the integral test; it is a function of of skew, α, and dataset size, N:

$$i_{med} \in \begin{cases} \Theta\left(2^{\frac{1}{\alpha-1}} N\right) & \text{if } \alpha < 1; \\ \Theta\left(\sqrt{N}\right) & \text{if } \alpha = 1; \\ \Theta\left(\log N\right) & \text{if } \alpha > 1 . \end{cases} \tag{3}$$

For our technique to be successful, it is imperative that d_{total}, the total delay imposed during an extraction attack, be significantly higher than d_{med}.

$$\frac{d_{total}}{d_{med}} = \frac{\left(\frac{1}{N f_{max}}\right) \sum_{i=1}^{N} i^{\alpha+\beta}}{\left(\frac{1}{N f_{max}}\right) i_{med}^{\alpha+\beta}} \in \begin{cases} \Theta\left(2^{\frac{\alpha+\beta}{1-\alpha}} N\right) & \text{if } \alpha < 1; \\ \Theta\left(N^{\frac{\beta+3}{2}}\right) & \text{if } \alpha = 1; \\ \Theta\left(N\left(\frac{N}{\log N}\right)^{\alpha+\beta}\right) & \text{if } \alpha > 1 . \end{cases} \tag{4}$$

Thus, for skews equal to or greater than 1, setting β to a high yet acceptable value ensures that the delay for an adversary's query will indeed be orders of magnitude higher than that for a genuine user's query. For skew less than 1, the benefit over the naïve approach is only linear in N. However, as these fractional skews approach 1 – as they are expected to – the exponential coefficient dominates, thereby maintaining the ratio of adversary to legitimate user delay at a desirably high value.

2.2 Capped Maximum Delay

This simple scheme provides low *median* delay, but can produce unacceptably long delays for legitimate users from time to time. After all, even the the least popular tuples will eventually be required by some legitimate queries, and the scheme as described above will delay these for a long time. Such delays may displease these legitimate users – something that the information provider certainly does not wish to do.

To prevent excessive delays – and the resulting unhappy customers – we cap the maximum delay that will be added to the retrieval of a single data item, no matter how infrequently accessed. This approach retains the benefits of the simple scheme; the asymptotic relationships between adversary and median query remain the same.

There is some tuple, at rank M, that is assigned the maximum acceptable delay, d_{max}. Since delay increases with decreasing popularity, all tuples that are accessed less frequently will have their delays capped at d_{max}, rather than as computed by the basic scheme. We can express d_{max} as

$$d_{max} = \frac{1}{N}\left(\frac{M^{\alpha+\beta}}{f_{max}}\right) . \tag{5}$$

This alters our expression for the delay for an adversary:

$$d_{total} = \sum_{i=1}^{M} d(i) + (N-M)d_{max} = \frac{1}{Nf_{max}}\left(\sum_{i=1}^{M} i^{\alpha+\beta} + (N-M)M^{\alpha+\beta}\right) . \tag{6}$$

The maximum delay allowed is obviously greater than the median delay. Therefore, the median rank of the distribution does not change in this scenario, and the median delay, d_{med} remains the same as before. The modified relationship between adversary and genuine user delays is now

$$\frac{d_{total}}{d_{med}} = \frac{\left(\frac{1}{Nf_{max}}\right)\left(\sum_{i=1}^{M} i^{\alpha+\beta} + (N-M)M^{\alpha+\beta}\right)}{\left(\frac{1}{Nf_{max}}\right) i_{med}^{\alpha+\beta}} . \tag{7}$$

Since $M \in \Theta(N)$ (i.e., M increases linearly with N), our initial results remain true.

2.3 Learning the Distribution

While the analysis in the preceding sections was worked out for the Zipf distribution, there is nothing in the definition of the scheme itself that requires that the workload follow such a distribution. Moreover, even if the query workload did follow such a distribution, we would still have the task of determining the popularity rank, and the frequency of access, for each tuple to determine the correct amount of delay to add.

We associate a *count* with each tuple that tracks the number of times that tuple has been requested. The value of this count, normalized by a global count of all requests, directly indicates the popularity of the tuple.

The simplest way to implement this is to add a *count* attribute to each tuple in the relation. However, this has the undesirable effect of turning every read access into a read-modify-write access, and therefore causing a substantial performance hit. We propose several mechanisms to keep reasonable the cost of maintaining these counts (Section 4.4).

Start-Up Transients. While counts are fine for indicating popularity of a tuple in steady state, we still have to deal with a (potentially long) start-up period during which representative statistics have not yet been gathered. Placing caps on maximum delays gives us a convenient mechanism to manage these start-up transients. We assume all items are equally unpopular with frequencies of zero. With these initial conditions, early queries will generate high delays, even if they are for popular items. However, the capped delay allows us to serve these queries in reasonable time while we are learning the distribution. The delay associated with popular items falls rapidly thereafter.

Changing Distributions. Many datasets will have popularity distributions that change over time. Unfortunately, this presents a problem for our basic scheme. Because there are often many more newly-popular requests, they have a significant impact on median delay.

We solve this by introducing a weight for each request which decays exponentially with age. The decay is applied at each request, uniformly to all counts. We use a static decay term, δ; the choice of appropriate δ depends upon the underlying dynamics of the popularity distribution. In situations where it is not known, one can simultaneously track counts with more than one decay term, switching to the appropriate set as the request pattern warrants – a technique used previously in both wireless networking [16] and energy management [10]. This adaptive strategy has the added benefit of tracking distributions with non-stationary second-order terms.

It is expensive to discount the value of every count at each access. Instead, we inflate the value by which each count increases at each access, and normalize counts by this value, with the same effect. To prevent overflow, we must reset counters from time to time, at some loss of precision.

2.4 Attacks

Our scheme applies delays to individual queries, but does not inhibit queries posed in parallel. An adversary able to manufacture identities can use these identities to pose queries in parallel. Because our database engine cannot label these queries as coming from the same user, the adversary pays only the maximum among individual penalties, not the sum over all of them.

It is important to note that true *Sybil* attacks [13] are difficult to engineer. For example, suppose that routable IP addresses are used as identities; clearly, IP addresses are trivial to forge, but it is more difficult to also control the route to that forged address to receive the result. An adversary may be able to control many addresses within a single subnet, but any given subnet can be treated as an aggregate, with responses rate-limited across all users in that subnet.

However, even if an adversary is able to manufacture identities, one can prevent unbounded parallelism through rate-limiting the granting of access to the database itself. If only one new user every t seconds is given an account to access the database, we can place a lower bound on the time it would take an adversary to accumulate enough identities for the parallel attack to become feasible. If this time is comparable to the delay imposed on an adversary with a single identity, then the parallel attack is rendered moot. Equivalently, one can charge a small fee for registration, computed so that a parallel adversary would have to spend as much in registration fees as to collect the data separately.

Storefront and *cached storefront* attacks are more difficult to defend against. Since the attack only forwards queries from legitimate users, the adversary need not register an undue number of identities. If the adversary is of significant size, we will notice the increased traffic, and a simple imposition of a limit on queries from a single user will suffice as a defense. (If the adversary attempts to counter this defense by manufacturing multiple identities, then our protection against a Sybil attack will protect us in this case as well). The success of a storefront attack ultimately depends on the business model of the source data provider – clearly, it costs the attacker at least as much to provide this service as the source provider charges, making it more difficult for the attacker to compete.

3 Exploiting Data Change

The main limitation of our scheme is that it depends on skew in access patterns. If access patterns are uniform, all items in the database have approximately the same rank, and hence the same delay. However, even without skew in access patterns, delay can be used in many cases by exploiting differences among the rates at which elements in a dataset change. Most databases are not static, one-time collections of information: rather, they are updated frequently. Where data changes, it is no longer necessary to thwart an attacker for an arbitrarily long time – it suffices to introduce a large delay relative to the data change rate. In the resulting system, an attacker can never have a consistent, current snapshot of the dataset, since some extracted tuples will always be stale. This scheme is entirely independent of access pattern, and so applies to datasets with uniform access patterns; exactly the scenario our access rate-based scheme cannot handle.

The basic idea is to charge small delays to frequently-updated items, but large delays to infrequently-updated ones. Tuples that stay fresh longer take longer to retrieve than tuples whose values change more often. This ensures that the delay incurred in retrieving the entire dataset is sufficient to cause a large fraction of the retrieved data to be obsolete. While this technique does not depend on skew in access rates, it does depend on skew in update rates. Such skew has been observed in practice [9], and has been used to identify which pages a crawler [15] needs to re-fetch frequently [12].

3.1 Assigning Delays

Consider a dataset with uniform access frequency, but with a skewed update frequency. Let the update frequency have a Zipf distribution with Zipf parameter α. As in the case of skewed access frequency, let the delay of an item, i, be inversely proportional to its update rate, r_i.

$$d(i) \propto \frac{1}{r_i} \tag{8}$$

More precisely,

$$d(i) = \frac{c}{N}\left(\frac{i^\alpha}{r_{max}}\right) . \tag{9}$$

An item in the dataset is considered stale if its value changes at least once during the execution of the adversary's query, i.e., its value is no longer the same as that obtained via the query. For the ith ranked item to be stale,

$$d_{total} \geq \frac{1}{r_i} . \tag{10}$$

If the ith item is the least frequently updated item that becomes stale during this time, then all items ranked higher than i, i.e, those that are more frequently updated, will also have become stale. In other words, the number of stale items in the dataset is i. Thus, a fraction S of the dataset (of size N) will be stale if the (SN)th ranked item is stale. From the equation above, the maximum value of S, S_{max}, can be computed as follows:

$$(S_{max}N)^\alpha = \frac{c_{max}}{N}\sum_{i=1}^{N} i^\alpha \tag{11}$$

$$S_{max} \approx \left(\frac{c_{max}}{1+\alpha}\right)^{\frac{1}{\alpha}} . \tag{12}$$

This value of S_{max} tells us what fraction of the dataset is guaranteed to be obsolete and of little use when stolen by an adversary. This fraction is limited only by c_{max}, the maximum allowable value for c, a constant which, given a particular data size and update rate distribution, places a reasonable upper bound on the delay for a legitimate user query.

4 Evaluation

It is easy to demonstrate experimentally that our scheme works with accesses that are Zipf-distributed, since that is precisely the assumption used in our analysis. Running such experiments with synthetic data and synthetic query loads produces observed results that match analytical predictions perfectly. We show results of such experiments in this section.

However, apart from these intuitive results, we also ask how the scheme performs with real data. To this end, we applied our scheme to two sets of real access traces. The first is a trace-history of web-page requests, while the second is based on sales of tickets to movies screened during a single calendar year. Since both datasets exhibit some skew, we can apply delays based on popularity of items, on both of them.

For each of these traces, we ask two questions. First, what is the median delay a legitimate user might expect to see in our scheme? Second, what is the delay that an adversary would expect to see for this dataset? The answers to these questions depend on how well the database has succeeded in learning the popularity of an item, and in tracking changes to this popularity over time. For this purpose, recall that our frequency-of-access observations are weighted with time, with a specified decay rate. We explore a range of decay rates in each dataset. In all cases, an adversary suffers delays many orders of magnitude beyond that of legitimate users.

In addition to these real traces with biased access patterns, we also give results for a synthetic workload with uniform access patterns, but changing underlying data. We consider a variety of rates of bias in update rates. For each case, we quantify the median delay imposed on legitimate users, the total delay expected of adversaries, and the fraction of the extracted database that will be stale.

4.1 Delays for Static Popularity

The first dataset is a year-long trace of web-client usage [2], to analyze user queries posed to a web-server. This trace exhibits a relatively static popularity distribution over its lifetime. While this is a log of accesses to web pages rather than database tuples, it is reasonable to expect that similar access patterns may apply in both cases.

The trace itself loosely follows an exponential popularity distribution with $\alpha \approx 1.5$, as shown in Figure 1. In this graph, the x axis lists the 10 most popular items by rank, while the y axis gives the number of requests to that object over the lifetime of the trace.

We replayed all 725,091 requests in this trace, subjecting each request to the delay as computed by our scheme to assign delays based on popularity, with a maximum delay of 10 seconds. The dataset however, has 12,179 records, which pales in comparison with real-world databases which are often terabytes and quickly approaching petabytes in size. For this reason, we first built synthetic datasets of larger sizes, and created a new scenario identical to the Calgary trace

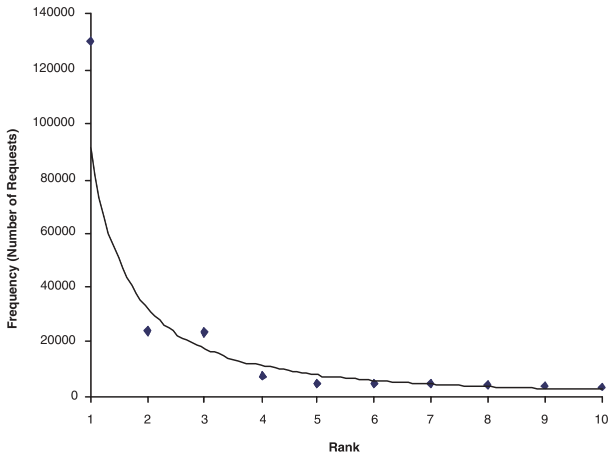

Fig. 1. Request Distribution: Calgary Trace

(in terms of access pattern, delay computation, etc.) except for the number of tuples. This helps to determine how our technique would perform on a real database. Our results appear in Table 1.

Table 1. Delays in Synthetic Traces

Database Size (tuples)	Median User Delay (ms)	Adversary Delay (weeks)
100,000	0.0	2
500,000	0.0	8
1,000,000	0.0	17

The typical user delay is observed to be negligibly small, while delays faced by an adversary are substantial, just as we expect. However, on implementing our scheme on the actual database, the small data size limits the largest penalty an adversary could possibly be subject to, to 34 hours, which is not large in absolute terms. This is due in large part to the very small size of the underlying database combined with a relatively modest ten-second maximum cap. This can be improved in two ways. Real datasets are likely to be much larger, so that total adversary delay should scale appropriately (as we have also shown). Secondly, raising the cap has no impact on the median delay, but directly affects the total delay imposed on an adversary, as shown in Table 2.

At the start of trace replay, we assumed nothing was known about the eventual distribution, but learned it over the course of the trace. After these legitimate requests were consumed, we computed the delay that would be imposed on an adversary if it were to extract the entire dataset, by examining the access counts after the trace was replayed. This was repeated for six different rates of

Table 2. Scaling Maximum Delay Costs

Cap (sec)	Adversary Delay (hours)
0.1	0.33
1	3.16
10	30.17
100	282.70

decay applied to the popularity metric, from 1 (no decay) to 1.0002. Note that because decay rates are exponents, results do not scale linearly; therefore we examine decay rates in logarithmic steps. The results appear in Table 3.

Table 3. Delays in Calgary Trace

Decay Rate	Median User Delay (ms)	Adversary Delay (hours)
1.000000	15.4	30.17
1.000001	24.9	31.06
1.000002	38.3	31.75
1.000005	118.6	32.76
1.000010	421.4	33.27
1.000020	2,241.6	33.61

Because this dataset exhibits a static access pattern, it is best to use the full history of prior accesses in determining delay. Therefore, a decay rate of 1.0 – no decay – provides the lowest possible delay for users without substantially lowering the delay imposed on an adversary. Even with this aggressive scheme, an adversary must wait more than a day to obtain this modest dataset of just over 12,000 objects; this is nearly 90% of the maximum possible delay.

4.2 Delays for Dynamic Popularity

The second dataset is based on cinema box office sales for 2002. This dataset has a rapidly shifting popularity distribution – new movies are released all the time, become immensely popular for a while, and then rapidly fade away as others take their place in the popular psyche. While we were not able to get access to actual traces of access to movie reviews, we obtained weekly box office sales data for all movies released in the year 2002 [3]. We used box office sales as the metric of popularity, and generated user requests to a database of movie records in proportion to the sales data for each week.

Compared to the Calgary trace, the resulting dataset does not exhibit the same degree of skew when viewed in its entirety. This is illustrated in Figure 2. In this graph, the x axis lists the 10 most popular movies by annual sales, while the y axis gives the sales totals for that film. Each week considered separately

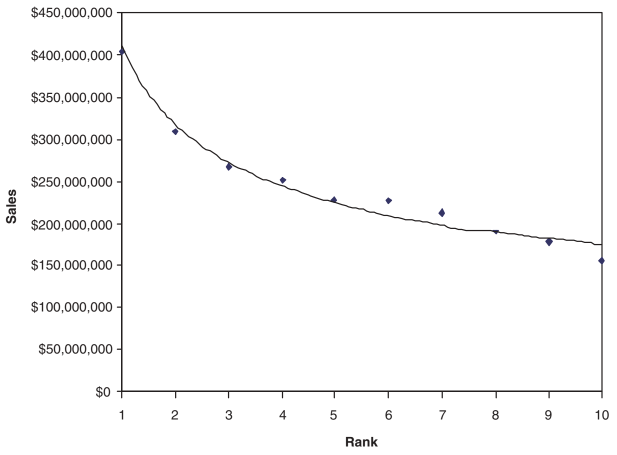

Fig. 2. Sales Distribution of Top 10 Movies of 2002

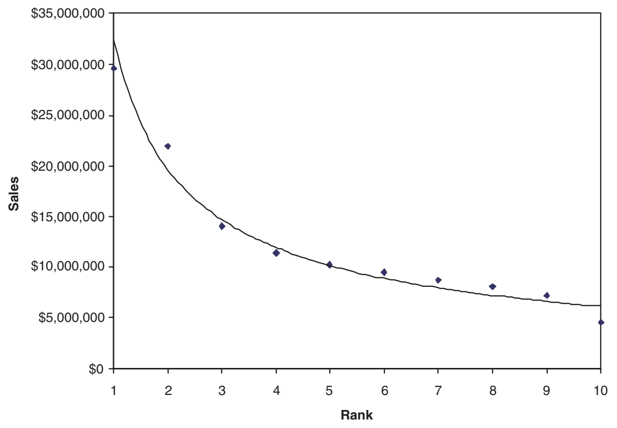

Fig. 3. Top 10 Movies for First Week of 2002

exhibits a more sharply skewed distribution. For example, Figure 3 shows the most popular films from the first week of 2002.

There were 634 films released over the course of 2002. We generated requests by week, one per \$100,000 in weekly box office sales. Delays were computed by popularity, applying decay factors at weekly boundaries. The maximum allowable delay was again 10 seconds. With this maximum delay, the largest penalty an adversary could possibly be subject to is 1.76 hours. However, being a tiny dataset, it is important to guage our scheme by the fact that an adversary incurs 100% of the maximum possible total delay in this scenario, which when scaled to a more typical real-world database, is a huge success!

As in the Calgary dataset, the request distribution was learned over time. We computed median and adversary delays for nine different rates of decay applied to the popularity metric, from 1 (no decay) to 5 and the results appear in Table 4.

Table 4. Delays in Box Office Data

Decay Rate	Median User Delay (ms)	Adversary Delay (hours)
1.00	0.03	1.33
1.01	0.04	1.51
1.02	0.05	1.45
1.05	0.08	1.46
1.10	0.14	1.61
1.20	0.26	1.70
1.50	0.53	1.74
2.00	0.79	1.75
5.00	1.26	1.76

The popularity distribution in this set varies quickly. Thus, the different decay factors give fairly similar median and adversary delays.

4.3 Dynamic Data Simulations

Where skew exists in access patterns, it can be exploited to penalize extraction without inconveniencing users. To measure our ability to do so, we created a relation with 100,000 tuples, and simultaneously posed queries and posted updates to this relation. Queries were posed with a uniform distribution, while updates were posted with Zipfian distributions with α values ranging between 0.25 and 2.5. Objects are assigned delays based on their relative rate of updates; the most frequently updated object is given the minimum delay, while the least frequently updated object is given the maximum delay. These delays were set so that an adversary should expect to obtain stale values for at least part of the set once the attempt at extraction is complete. For each skew rate, we measure the median expected user delay, the total expected adversarial delay, and the fraction of the set one would expect to be stale once extracted. The results are in Figures 4–6. Note that the first two have logarithmic y axes.

Delays imposed on adversary queries are effective only when noticeable skew exists in update rate. Thus, when both access pattern and update rate are uniform, the delay technique is not applicable. Such cases either assign high delays to legitimate queries, or penalize an extraction attack insufficiently.

The delay imposed on an adversary can be substantial in this scheme, as much as ten seconds per tuple for realistic skews. However, imposing delay in itself is not the principal goal. Rather, we wish to ensure that some non-trivial fraction of the database is stale once the adversary has extracted it. At modest skew levels, nearly all of the database is likely to be stale, since updates are distributed over most of the tuples during extraction. When updates are more focused, a smaller fraction of the database will be stale. However, at these levels of skew, the adversary will always incur the maximum possible delay.

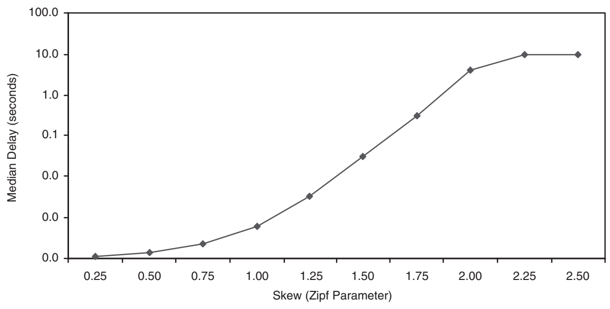

Fig. 4. Median User Delay – Assigned by Update

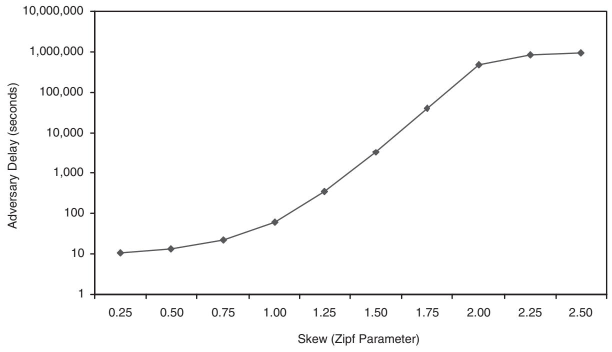

Fig. 5. Total Delay for Adversary – Assigned by Update

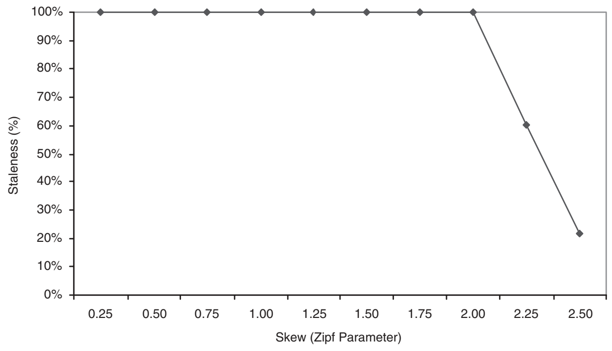

Fig. 6. Fraction of Stale Data – Assigned by Update

4.4 Implementation Overhead

Finally, we must quantify the cost of maintaining counts and computing delay. We do so in the context of the shortest queries possible – a set of simple selection queries. We posed 100 random selection queries to the database, each returning precisely one tuple, averaging across these queries. During the experiment, we maintained a small, write-behind cache of tuple counts. However, not all counts are kept in memory, resulting in some I/O overhead. Using the sampling for synopsis technique described by Gibbons [14] would reduce these modest overheads even further. Nevertheless, overheads are small. On average, each selection query took 55.17 ms, with a standard deviation of 15.61 ms, without any delay computation or maintenance of tuple counts. With the addition of these costs, the average selection query took 66.20 ms, with a standard deviation of 27.84 ms. This yields an average overhead of 11.04 ms, or 20%. These results are summarized in Table 5.

Table 5. Overheads in Simple Selection Queries

Base query cost		Total cost		Overhead
avg (ms)	stdev (ms)	avg (ms)	stdev (ms)	(ms)
55.17	(15.61)	66.20	(27.84)	11.04

5 Related Work

There are several systems which use delay as a security mechanism: password-based authentication [5, 19], self-securing storage [20], and archival repositories [18]. To the best of our knowledge, our work is the first to propose the use of delay to defend against extraction attacks, in a database system or elsewhere.

In addition to raising the expense of an extraction attack, one can consider watermarking – indelibly identifying the source of data in the data itself. A watermark [21] is an undetectable signature embedded in some data object; the watermark serves to identify the source of the document. Originally applied to still images, watermarks have also been applied to video [17], audio [6], and rich-text documents [7]. In general, watermarking relies on embedding patterns in the low-order bits of an object's representation – be it image, sound, or spatial layout. Watermarking has also been applied to relational databases [1], by making minor modifications to the values of non-key attributes. Although robust and high-performance, this technique still requires the challenging task of detecting that theft has occurred and then analyzing the pirated data. Our strategy, which is orthogonal to the use of watermarking, raises the barrier to data theft in the first place.

6 Conclusion

Data providers face a difficult challenge. They spend significant effort collecting their repository. They must provide public access to their datasets, yet they cannot allow another entity to re-use their work by copying the database wholesale.

Delay can be used to provide legitimate access while preventing extraction attacks. By taking advantage of skew – either in access or update pattern – the provider can impose delays such that legitimate users are not inconvenienced, yet an adversary either suffers intolerable delay or retrieves data with a substantial number of stale components. This technique can be implemented with modest overhead, providing increased protection against such extraction attacks.

References

1. R. Agrawal and J. Kiernan. Watermarking relational databases. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 155–166, Hong Kong, China, August 2002.
2. M. F. Arlitt and C. L. Williamson. Web server workload characterization: the search for invariants. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 126–137, Philadelphia, PA, May 1996.
3. P. Bart, editor. *Variety*. Reed Business Information, New York, NY, 1905-.
4. N. Bhatti, A. Bouch, and A. Kuchinsky. Integrating user-perceived quality into Web server design. In *Proceedings of the Ninth International World Wide Web Conference*, pages 1–16, Amsterdam, Netherlands, May 2000.
5. D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson. TENEX: a paged time-sharing system for the PDP-10. *Communications of the ACM*, 15(3):135–143, March 1972.
6. L. Boney, A. H. Tewfik, and K. N. Hamdy. Digital watermarks for audio signals. In *Proceedings of the International Conference on Multimedia Computing and Systems*, pages 473–480, Hiroshima, Japan, June 1996.
7. J. Brassil and L. O'Gorman. Watermarking document images with bounding box expansion. In *Information Hiding First International Workshop*, pages 227–235, Cambridge, UK, May 1996.
8. L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: evidence and implications. In *IEEE INFOCOM '99: Conference on Computer Communications*, volume 1, pages 126–134, New York, NY, March 1999.
9. B. E. Brewington and G. Cybenko. Keeping up with the changing Web. *Computer*, 33(5):52–58, May 2000.
10. J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle. Managing energy and server resources in hosting centers. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 103–116, Banff, AB, Canada, October 2001.
11. M. Chesire, A. Wolman, G. M. Voelker, and H. M. Levy. Measurement and analysis of a streaming-media workload. In *Proceedings fo teh 3rd USENIX Symposium on Internet Technologies and Systems*, pages 1–12, San Francisco, CA, March 2001.
12. J. Cho and H. Garcia-Molina. The evolution of the Web and implications for an incremental crawler. In *Proceedings of the 26th International Conference on Very Large Data Bases*, pages 200–209, Cairo, Egypt, September 2000.
13. J. R. Douceur. The Sybil attack. In *Proceedings of the First International Workshop on Peer-to-Peer Systems*, pages 251–260, Cambridge, MA, March 2002.
14. P. B. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 331–342, Seattle, WA, June 1998.
15. A. Heydon and M. Najork. Mercator: a scalable, extensible Web crawler. *World Wide Web*, 2(4):219–229, 1999.

16. M. Kim and B. D. Noble. Mobile network estimation. In *7th ACM Conference on Mobile Computing and Networking*, pages 298–309, Rome, Italy, July 2001.
17. B. M. Macq and J.-J. Quisquater. Cryptology for digital TV broadcasting. *Proceedings of the IEEE*, 83(6):944–957, June 1995.
18. P. Maniatis, D. S. H. Rosenthal, M. Roussopoulos, M. Baker, T. J. Giuli, and Y. Muliadi. Preserving peer replicas by rate-limited sampled voting. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 44–59, Bolton Landing, NY, October 2003.
19. R. Morris and K. Thompson. Password security: A case history. *Communications of the ACM*, 22(11):594–597, November 1979.
20. J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-securing storage: Protecting data in compromised systems. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, pages 165–179, San Diego, CA, October 2000.
21. A. Z. Tirkel, G. A. Rankin, R. M. van Schyndel, W. J. Ho, N. R. A. Mee, and C. F. Osborne. Electronic water mark. In *Proceedings, Digital Image Computing: Techniques and Applications*, volume 2, pages 666–673, Sydney, Australia, December 1993.
22. G. Zipf. *Selective Studies and the Principle of Relative Frequency in Language.* Harvard University Press, Cambridge, MA, 1932.