Searching the Deep Web

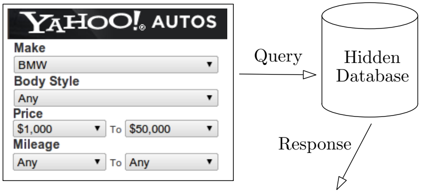
Yufei Tao

KAIST

June 9, 2013

We already know that a search engine discovers the world wide web with a web crawler, which works by following hyperlinks. The term surface web is often used to refer to the collection of web pages that can be found by a web crawler.

Opposite to this is the deep web. It alludes to those web pages that cannot be unsurfaced by simply crawling hyperlinks. Often times, such web pages do not exist until they are dynamically generated in response to users' queries.



Make: BMW
Body Style: Any
Price: $1,000 To $50,000
Mileage: Any To Any

Make	Body style	Price	Mileage
BMW	sedan	\$17,500	68,647 mi
BMW	sedan	\$17,500	76,072 mi
BMW	coupe	\$3,299	158,573 mi
BMW	convertible	\$50,000	5,231 mi
	...		

As an example, consider Yahoo Auto, a popular website for people to trade used vehicles. A potential buyer fills in a form that solicits her/his preferences (see the above figure). Then, the system queries its backend database – one that is hidden from the public's direct access – to find vehicles satisfying the preferences, whose information is displayed in the user's browser. Such information, which did not exist before the query, is part of the deep web.

It has been widely recognized that the deep web contains a gigantic amount of valuable information. It would therefore be useful for a search engine to be able to manage such information as well.

Towards that purpose, a search engine must acquire the information in the first place. Why is this difficult? For example, in the scenario of the previous slide, can't a user glean all the vehicles in the hidden database by specifying:

$$\begin{gathered}\texttt{Make} = \text{any}\\ \texttt{Body style} = \text{any}\\ \texttt{Price} \text{ from } 0 \text{ to } \infty\\ \texttt{Mileage} \text{ from } 0 \text{ to } \infty?\end{gathered}$$

The answer is no. This is because a site like Yahoo typically puts a limit on how many records to return. For example, the limit at Yahoo Auto is about 1000. In other words, when over 1000 vehicles satisfy the user's query, only 1000 of them are returned, together with a note like "refine your query to retrieve more vehicles".

Think

Why such a limit? Is it to prevent people from obtaining the entire hidden database?

So here comes the question: how to extract the entire hidden database behind Yahoo Autoo with the smallest number of queries?

Think

Why do we want to minimize the number of queries?

Formally, we will consider the following hidden database crawling problem. The data space is $\mathbb{N}^d$, where d is the dimensionality. Let D be a hidden database where each element is a point in the d-dimensional space.

Each query that can be issued by a user specifies an axis-parallel rectangle q in $\mathbb{N}^d$. The server returns:

- the entire $q \cap D$, if $|q \cap D| \le k$ where k is a system parameter. We say that such a query is resolved.
- arbitrary k points in $q \cap D$, otherwise. Also, in this case, the system also returns a signal to indicate that not all the results have been returned. We say that such a query overflows.

The goal is to obtain the entire D by asking queries strategically. The cost of the algorithm is defined to be the number of queries issued.

D must not have $k + 1$ points that are at the same location. Otherwise, the problem admits no solution at all (i.e., no algorithm can guarantee extracting the entire D) – think: why?

We have assumed that all attributes are numeric. In practice, an attribute can be categorical, e.g., `Make` of a vehicle. Categorical attributes require different handling, which is not required by this course.

Let us define a basic operation.

Let q be a rectangle $[a_1, b_1] \times [a_2, b_2] ... \times [a_d, b_d]$. A 3-way split at value v of dimension i generates three rectangles:

$$\begin{aligned} q_{left} &= [a_1, b_1] \times ... \times [a_i, v-1] \times ... \times [a_d, b_d] \\ q_{mid} &= [a_1, b_1] \times ... \times [v, v] \times ... \times [a_d, b_d]. \\ q_{right} &= [a_1, b_1] \times ... \times [v+1, b_i] \times ... \times [a_d, b_d]. \end{aligned}$$

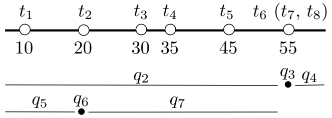
Let us first consider $d = 1$. We will solve a more general problem: given any rectangle q, our algorithm extracts the entire $q \cap D$ from the server.

Issue a query with q to the server. Let R be the set of points returned by the sever. If q is resolved, then we finish by returning R.

Consider now the case where q overflows. Sort all the points in R in ascending order, breaking ties arbitrarily. Let p be the $(k/2)$-th point in the sorted order. Count the number c of points in R that are equal to p (including p itself).

Perform a 3-way split of q at p into q_{left}, q_{mid}, and q_{right}. Recursively retrieve $q_{left} \cap D$, $q_{mid} \cap D$, and $q_{right} \cap D$.

Example.



- We issue the first query with $q_1 = (-\infty, \infty)$. Suppose that the server returns $\{t_3, t_6, t_7, t_8\}$. So the query is split into $q_2 = (-\infty, 54]$, $q_3 = [55, 55]$, and $q_4 = [56, \infty)$.
- We then issue q_2. Suppose that the server returns $\{t_1, t_2, t_4, t_5\}$. So the query is split into $q_5 = (-\infty, 19]$, $q_6 = [20, 20]$, and $q_7 = [20, 54]$.
- Next, we issue q_3, q_4, q_5, q_6, and q_7, all of which are resolved.

Lemma

The algorithm issues $O(n/k)$ queries.

Next we extend the algorithm to $d > 1$. As before, given a query rectangle q, we will return $q \cap D$.

Let $[a_1, b_1]$ be the extent of q on the first dimension.

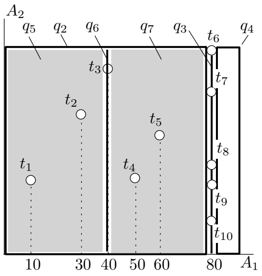
If $a_1 = b_1$, then we process q as a $(d-1)$-dimensional query in the data space $[a_1, a_1] \times \mathbb{N}^{d-1}$.

If $a_1 \neq b_1$, then issue a query with q to the server. Let R be the set of points returned by the sever. If q is resolved, then we finish by returning R.

Consider now the case where q overflows. Sort all the points of R in ascending order by their coordinates on the first dimension, breaking ties arbitrarily. Let p be the $(k/2)$-th point in the sorted order. Count the number c of points in R that are equal to p (including p itself) on the first dimension.

Perform a 3-way split of q at p into q_{left}, q_{mid}, and q_{right}. Recursively retrieve $q_{left} \cap D$, $q_{mid} \cap D$, and $q_{right} \cap D$.

Example.



- We issue the first query with $q_1 = (-\infty, \infty) \times (-\infty, \infty)$. Suppose that the server returns $\{t_4, t_7, t_8, t_9\}$. So the query is split into q_2, q_3, and q_4, as shown in the above figure.
- We then issue q_2. Suppose that the server returns $\{t_2, t_3, t_4, t_5\}$. So the query is split into q_5, q_6, and q_7.
- Next, we answer q_3 using our 1d algorithm (observe that q_3 essentially is a 1d query).
- Finally, we issue q_4, q_5, q_6, and q_7, all of which are resolved.

Lemma

The algorithm issues $O(dn/k)$ queries.