Key Differences between HTTP/1.0 and HTTP/1.1[1]

Dina Zeliger, Advanced Topics in Database Theory

Basically there are quite a lot of changes and the paper discusses them thoroughly. I am going to talk about the differences that sound more interesting to me.

Introduction

By any reasonable standard, the HTTP/1.0 protocol has been stunningly successful. As a measure of its popularity, HTTP accounted for about 75% of Internet backbone traffic in a (somewhat) recent study.

HTTP/1.0 evolved from the original ``0.9'' version of HTTP. The process leading to HTTP/1.0 involved significant debate, but never produced a formal specification. The HTTP Working Group (HTTP-WG) of the Internet Engineering Task Force (IETF) produced a document (RFC1945) that described the ``common usage'' of HTTP/1.0, but did not attempt to create a formal standard out of the many variant implementations. Instead, over a period of roughly four years, the HTTP-WG developed an improved protocol, known as HTTP/1.1.

The HTTP/1.1 specification is almost three times as long as RFC1945, reflecting an increase in complexity, clarity, and specificity. Even so, numerous rules are implied by the HTTP/1.1 specification, rather than being explicitly stated.

We'll now discuss some areas in which there have been significant changes.

Extensibility

The HTTP/1.1 effort assumed, from the outset, that compatibility with the installed base of HTTP implementations was mandatory. Because the HTTP/1.1 effort took over four years, and generated numerous interim draft documents, many implementers deployed systems using the "HTTP/1.1" protocol version before the final version of the specification was finished. This created another compatibility problem: the final version had to be substantially compatible with these pseudo-HTTP/1.1 versions, even if the interim drafts turned out to have errors in them.

[1] Based on a the paper *Key Differences between HTTP/1.0 and HTTP/1.1* by **Balachander Krishnamurthy**, **Jeffrey C. Mogul** and **David M. Kristol** (can be found at http://www.research.att.com/~bala/papers/h0vh1.html)

The compatibility issue also underlined the need to include, in HTTP/1.1, as much support as possible for future extensibility. That is, if a future version of HTTP were to be designed, it should not be hamstrung by any additional compatibility problems.

Note that HTTP has always specified that if an implementation receives a header that it does not understand, it must ignore the header. This rule allows a multitude of extensions without any change to the protocol version, although it does not by itself support all possible extensions.

Version numbers

In many cases the version number in an HTTP message can be used to deduce the capabilities of the sender. A companion document to the HTTP specification clearly specified the ground rules for the use and interpretation of HTTP version numbers.

The version number in an HTTP message refers to the hop-by-hop sender of the message, not the end-to-end sender. Thus the message's version number is directly useful in determining hop-by-hop message-level capabilities, but not very useful in determining end-to-end capabilities. For this reason, as well as to support debugging, HTTP/1.1 defines a `Via` header that describes the path followed by a forwarded message. The path information includes the HTTP version numbers of all senders along the path and is recorded by each successive recipient. Note that only the last of multiple consecutive HTTP/1.0 senders will be listed, because HTTP/1.0 proxies will not add information to the `Via` header.)

Upgrading to other protocols

In order to ease the deployment of incompatible future protocols, HTTP/1.1 includes the new `Upgrade request` header. By sending the `Upgrade` header, a client can inform a server of the set of protocols it supports as an alternate means of communication. The server may choose to switch protocols, but this is not mandatory.

Caching

Caching is effective because a few resources are requested often by many users, or repeatedly by a given user. Caches are employed in most Web browsers and in many proxy servers. Caching improves user-perceived latency by eliminating the network communication with the origin server. Caching also reduces bandwidth consumption, by avoiding the transmission of unnecessary network packets. Reduced bandwidth consumption also indirectly reduces latency for un-cached interactions, by reducing network congestion. Finally, caching can reduce the load on origin servers (and on intermediate proxies), further improving latency for un-cached interactions.

One risk with caching is that the caching mechanism might not be “semantically transparent”: that is, it might return a response different from what would be returned by direct communication with the origin server. While some applications can tolerate non-transparent responses, many Web applications (electronic commerce, for example) cannot.

Caching in HTTP/1.0

HTTP/1.0 provided a simple caching mechanism. An origin server may mark a response, using the `Expires` header, with a time until which a cache could return the response without violating semantic transparency. Further, a cache may check the current validity of a response using what is known as a *conditional request*: it may include an `If-Modified-Since` header in a request for the resource, specifying the value given in the cached response's `Last-Modified` header. The server may then either respond with a 304 (Not Modified) status code, implying that the cache entry is valid, or it may send a normal 200 (OK) response to replace the cache entry.

HTTP/1.0 also included a mechanism, the `Pragma: no-cache` header, for the client to indicate that a request should not be satisfied from a cache.

The HTTP/1.0 caching mechanism worked moderately well, but it had many conceptual shortcomings. It did not allow either origin servers or clients to give full and explicit instructions to caches; therefore, it depended on a body of heuristics that were not well-specified. This led to two problems: incorrect caching of some responses that should not have been cached, and failure to cache some responses that could have been cached. The former causes semantic problems; the latter causes performance problems.

Caching in HTTP/1.1

In HTTP/1.1 terminology, a cache entry is *fresh* until it reaches its expiration time, at which point it becomes *stale*. A cache need not discard a stale entry, but it normally must revalidate it with the origin server before returning it in response to a subsequent request. However, the protocol allows both origin servers and end-user clients to override this basic rule.

In HTTP/1.0, a cache revalidated an entry using the `If-Modified-Since` header. This header uses absolute timestamps with one-second resolution, which could lead to caching errors either because of clock synchronization errors, or because of lack of resolution. Therefore, HTTP/1.1 introduces the more general concept of an opaque cache validator string, known as an *entity tag*. If two responses for the same resource have the same entity tag, then they must (by specification) be identical. Because an entity tag is opaque, the origin server may use any information it deems necessary to construct it (such as a fine-grained timestamp or an internal database pointer), as long as it meets the uniqueness requirement. Clients may compare entity tags for equality, but cannot otherwise manipulate them. HTTP/1.1 servers attach entity tags to responses using the `ETag` header.

HTTP/1.1 includes a number of new conditional request-headers, in addition to `If-Modified-Since`. The most basic is `If-None-Match`, which allows a client to present one or more entity tags from its cache entries for a resource. If none of these matches the resource's current entity tag value, the server returns a normal response; otherwise, it may return a 304 (Not Modified) response with an `ETag` header that indicates which cache entry is currently valid. Note that this mechanism allows the server to cycle through a set of possible responses, while the `If-Modified-Since` mechanism only generates a cache hit if the most recent response is valid.

HTTP/1.1 also adds new conditional headers called `If-Unmodified-Since` and `If-Match`, creating other forms of preconditions on requests.

The Cache-Control header

In order to make caching requirements more explicit, HTTP/1.1 adds the new `Cache-Control` header, allowing an extensible set of cache-control directives to be transmitted in both requests and responses. The set defined by HTTP/1.1 is quite large, so we concentrate on several notable members.

Because the absolute timestamps in the HTTP/1.0 `Expires` header can lead to failures in the presence of clock skew, HTTP/1.1 can use relative expiration times, via the `max-age` directive. (It also introduces an Age header, so that caches can indicate how long a response has been sitting in caches along the way.)

Because some users have privacy requirements that limit caching beyond the need for semantic transparency, the `private` and `no-store` directives allow servers and clients to prevent the storage of some or all of a response. However, this does not guarantee privacy; only cryptographic mechanisms can provide true privacy.

Some proxies transform responses (for example, to reduce image complexity before transmission over a slow link), but because some responses cannot be blindly transformed without losing information, the `no-transform` directive may be used to prevent transformations.

Bandwidth optimization

Network bandwidth is almost always limited. HTTP/1.0 wastes bandwidth in several ways that HTTP/1.1 addresses. A typical example is a server's sending an entire (large) resource when the client only needs a small part of it. There was no way in HTTP/1.0 to request partial objects. Also, it is possible for bandwidth to be wasted in the forward direction: if a HTTP/1.0 server could not accept large requests, it would return an error code after bandwidth had already been consumed. What was missing was the ability to negotiate with a server and to ensure its ability to handle such requests before sending them.

Range requests

A client may need only part of a resource. For example, it may want to display just the beginning of a long document, or it may want to continue downloading a file after a transfer was terminated in mid-stream. HTTP/1.1 *range requests* allow a client to request portions of a resource. While the range mechanism is extensible to other units (such as chapters of a document, or frames of a movie), HTTP/1.1 supports only ranges of bytes. A client makes a range request by including the Range header in its request, specifying one or more contiguous ranges of bytes. The server can either ignore the `Range` header, or it can return one or more ranges in the response.

If a response contains a range, rather than the entire resource, it carries the 206 (Partial Content) status code. This code prevents HTTP/1.0 proxy caches from accidentally treating the response as a full one, and then using it as a cached response to a subsequent request. In a range response, the `Content-Range` header indicates the offset and length of the returned range.

Range requests can be used in a variety of ways:

1. To read the initial part of an image, to determine its geometry and therefore do page layout without loading the entire image
2. To complete a response transfer that was interrupted (either by the user or by a network failure)
3. To read the tail of a growing object.

Expect and 100 (Continue)

Some HTTP requests (for example, the PUT or POST methods) carry request bodies, which may be arbitrarily long. If, the server is not willing to accept the request, perhaps because of an authentication failure, it would be a waste of bandwidth to transmit such a large request body.

HTTP/1.1 includes a new status code, 100 (Continue), to inform the client that the request body should be transmitted. When this mechanism is used, the client first sends its request headers, and then waits for a response. If the response is an error code, such as 401 (Unauthorized), indicating that the server does not need to read the request body, the request is terminated. If the response is 100 (Continue), the client can then send the request body, knowing that the server will accept it.

However, HTTP/1.0 clients do not understand the 100 (Continue) response. Therefore, in order to trigger the use of this mechanism, the client sends the new `Expect` header, with a value of `100-continue`.

Because not all servers use this mechanism (the `Expect` header is a relatively late addition to HTTP/1.1, and early “HTTP/1.1” servers did not implement it), the client must not wait indefinitely for a 100 (Continue) response before sending its request body. HTTP/1.1 specifies a number of somewhat complex rules to avoid either infinite waits or wasted bandwidth.

Compression

One well-known way to conserve bandwidth is through the use of data compression. While most image formats (GIF, JPEG, MPEG) are pre-compressed, many other data types used in the Web are not. One study showed that aggressive use of additional compression could save almost 40% of the bytes sent via HTTP. While HTTP/1.0 included some support for compression, it did not provide adequate mechanisms for negotiating the use of compression, or for distinguishing between end-to-end and hop-by-hop compression.

HTTP/1.1 makes a distinction between content-codings, which are end-to-end encodings that might be inherent in the native format of a resource, and transfer-codings, which are always hop-by-hop. Compression can be done either as a content-coding or as a transfer-coding.

HTTP/1.0 includes the `Content-Encoding` header, which indicates the end-to-end content-coding(s) used for a message; HTTP/1.1 adds the `Transfer-Encoding` header, which indicates the hop-by-hop transfer-coding(s) used for a message.

HTTP/1.1 (unlike HTTP/1.0) carefully specifies the `Accept-Encoding` header, used by a client to indicate what content-codings it can handle, and which ones it prefers. HTTP/1.1 also includes the TE header, which allows the client to indicate which transfer-codings are acceptable, and which are preferred.

Network connection management

HTTP almost always uses TCP as its transport protocol. The original HTTP design used a new TCP connection for each request, so each request incurred the cost of setting up a new TCP connection. Since most Web interactions are short this was highly inefficient.

Web pages frequently have embedded images, sometimes many of them, and each image is retrieved via a separate HTTP request. The use of a new TCP connection for each image retrieval serializes the display of the entire page on the connection-setup latencies for all of the requests.

To resolve these problems, Padmanabhan and Mogul recommended the use of *persistent connections* and the *pipelining* of requests on a persistent connection.

The `Connection` header

Given the use of intermediate proxies, HTTP makes a distinction between the end-to-end path taken by a message, and the actual hop-by-hop connection between two HTTP implementations.

HTTP/1.1 introduces the concept of `hop-by-hop` headers: message headers that apply only to a given connection, and not to the entire path. The use of `hop-by-hop` headers creates a potential problem: if such a header were to be forwarded by a naive proxy, it might mislead the recipient.

Therefore, HTTP/1.1 includes the `Connection` header. This header lists all of the `hop-by-hop` headers in a message, telling the recipient that these headers must be removed from that message before it is forwarded. This extensible mechanism allows the future introduction of new `hop-by-hop` headers; the sender need not know whether the recipient understands a new header in order to prevent the recipient from forwarding the header.

Because HTTP/1.0 proxies do not understand the `Connection` header, however, HTTP/1.1 imposes an additional rule. If a `Connection` header is received in an HTTP/1.0 message, then it must have been incorrectly forwarded by an HTTP/1.0 proxy. Therefore, all of the headers it lists were also incorrectly forwarded, and must be ignored.

The `Connection` header may also list *connection-tokens*, which are not headers but rather per-connection Boolean flags. For example, HTTP/1.1 defines the token `close` to permit the peer to indicate that it does not want to use a persistent connection. Again, the `Connection` header mechanism prevents these tokens from being forwarded.

Persistent Connections

HTTP/1.0, in its documented form, made no provision for persistent connections. Some HTTP/1.0 implementations, however, use a `Keep-Alive` header to request that a connection persist. This design did not interoperate with intermediate proxies; HTTP/1.1 specifies a more general solution.

In recognition of their desirable properties, HTTP/1.1 makes persistent connections the default. HTTP/1.1 clients, servers, and proxies assume that a connection will be kept open after the transmission of a request and its response. The protocol does allow an implementation to close a connection at any time, in order to manage its resources, although it is best to do so only after the end of a response.

Because an implementation may prefer not to use persistent connections if it cannot efficiently scale to large numbers of connections or may want to cleanly terminate one for resource-management reasons, the protocol permits it to send a `Connection: close` header to inform the recipient that the connection will not be reused.

Pipelining

Although HTTP/1.1 encourages the transmission of multiple requests over a single TCP connection, each request must still be sent in one contiguous message, and a server must send responses (on a given connection) in the order that it received the corresponding requests. However, a client need not wait to receive the response for one request before sending another request on the same connection. In fact, a client could send an arbitrarily large number of requests over a TCP connection before receiving any of the responses. This practice, known as pipelining, can greatly improve performance. It avoids the need to wait for network round-trips, and it makes the best possible use of the TCP protocol.

Message transmission

HTTP messages may carry a body of arbitrary length. The recipient of a message needs to know where the message ends. The sender can use the `Content-Length` header, which gives the length of the body. However, many responses are generated dynamically. Without buffering the entire response (which would add latency), the server cannot know how long it will be and cannot send a `Content-Length` header.

When not using persistent connections, the solution is simple: the server closes the connection. This option is available in HTTP/1.1, but it defeats the performance advantages of persistent connections.

The Chunked transfer-coding

HTTP/1.1 resolves the problem of delimiting message bodies by introducing the `Chunked` transfer-coding. The sender breaks the message body into chunks of arbitrary length, and each chunk is sent with its length prepended; it marks the end of the message with a zero-length chunk. The sender uses the `Transfer-Encoding: chunked` header to signal the use of chunking.

This mechanism allows the sender to buffer small pieces of the message, instead of the entire message, without adding much complexity or overhead. All HTTP/1.1 implementations must be able to receive chunked messages.

The Chunked transfer-coding solves another problem, not related to performance. In HTTP/1.0, if the sender does not include a `Content-Length` header, the recipient cannot tell if the message has been truncated due to transmission problems. This ambiguity leads to errors, especially when truncated responses are stored in caches.

Trailers

Chunking solves another problem related to sender-side message buffering. Some header fields, such as `Content-MD5` (a cryptographic checksum over the message body), cannot be computed until after the message body is generated. In HTTP/1.0, the use of such header fields required the sender to buffer the entire message.

In HTTP/1.1, a chunked message may include a *trailer* after the final chunk. A trailer is simply a set of one or more header fields. By placing them at the end of the message, the sender allows itself to compute them after generating the message body.

The sender alerts the recipient to the presence of message trailers by including a `Trailer` header, which lists the set of headers deferred until the trailer. This alert, for example, allows a browser to avoid displaying a prefix of the response before it has received authentication information carried in a trailer.

HTTP/1.1 imposes certain conditions on the use of trailers, to prevent certain kinds of interoperability failure. For example, if a server sends a lengthy message with a trailer to an HTTP/1.1 proxy that is forwarding the response to an HTTP/1.0 client, the proxy must either buffer the entire message or drop the trailer. Rather than insist that proxies buffer arbitrarily long messages, which would be infeasible, the protocol sets rules that should prevent any critical information in the trailer (such as authentication information) from being lost because of this problem. Specifically, a server cannot send a trailer unless either the information it contains is purely optional, or the client has sent a `TE: trailers header`, indicating that it is willing to receive trailers (and, implicitly, to buffer the entire response if it is forwarding the message to an HTTP/1.0 client).

Internet address conservation

Companies and organizations use URLs to advertise themselves and their products and services. When a URL appears in a medium other than the Web itself, people seem to prefer "pure hostname" URLs; i.e., URLs without any path syntax following the hostname. These are often known as "vanity URLs", but in spite of the implied disparagement, it's unlikely that non-purist users will abandon this practice, which has led to the continuing creation of huge numbers of hostnames.

IP addresses are widely perceived as a scarce resource. The Domain Name System (DNS) allows multiple host names to be bound to the same IP address. Unfortunately, because the original designers of HTTP did not anticipate the "success disaster" they were enabling, HTTP/1.0 requests do not pass the hostname part of the request URL. For example, if a user makes a request for the resource at URL http://example1.org/home.html, the browser sends a message with the *Request-Line*

```
GET /home.html HTTP/1.0
```

to the server at example1.org. This prevents the binding of another HTTP server hostname, such as exampleB.org to the same IP address, because the server receiving such a message cannot tell which server the message is meant for. Thus, the proliferation of vanity URLs causes a proliferation of IP address allocations.

The Internet Engineering Steering Group (IESG), which manages the IETF process, insisted that HTTP/1.1 take steps to improve conservation of IP addresses. Since HTTP/1.1 had to interoperate with HTTP/1.0, it could not change the format of the Request-Line to include the server hostname. Instead, HTTP/1.1 requires requests to include a `Host` header that carries the hostname. This converts the example above to:

```
GET /home.html HTTP/1.1
Host: example1.org
```

If the URL references a port other than the default (TCP port 80), this is also given in the Host header.

Clearly, since HTTP/1.0 clients will not send Host headers, HTTP/1.1 servers cannot simply reject all messages without them. However, the HTTP/1.1 specification requires that an HTTP/1.1 server must reject any HTTP/1.1 message that does not contain a Host header.

Error notification

HTTP/1.0 defined a relatively small set of sixteen status codes, including the normal 200 (OK) code. Experience revealed the need for finer resolution in error reporting.

The Warning header

HTTP status codes indicate the success or failure of a request. For a successful response, the status code cannot provide additional advisory information, in part because the placement of the status code in the *Status-Line*, instead of in a header field, prevents the use of multiple status codes.

HTTP/1.1 introduces a `Warning` header, which may carry any number of subsidiary status indications. The intent is to allow a sender to advise the recipient that something may be unsatisfactory about an ostensibly successful response.

HTTP/1.1 defines an initial set of `Warning` codes, mostly related to the actions of caches along the response path. For example, a `Warning` can mark a response as having been returned by a cache during disconnected operation, when it is not possible to validate the cache entry with the origin server.

The `Warning` codes are divided into two classes, based on the first digit of the 3-digit code. One class of warnings must be deleted after a successful revalidation of a cache entry; the other class must be retained with a revalidated cache entry. Because this distinction is made based on the first digit of the code, rather than through an exhaustive listing of the codes, it is extensible to `Warning` codes defined in the future.

Other new status codes

There are 24 new status codes in HTTP/1.1. Two of the more notable additions include:

- 409 (Conflict), returned when a request would conflict with the current state of the resource. For example, a PUT request might violate a versioning policy.
- 410 (Gone), used when a resource has been removed permanently from a server, and to aid in the deletion of any links to the resource.

Most of the other new status codes are minor extensions.

Security, integrity, and authentication

In recent years, the IETF has heightened its sensitivity to issues of privacy and security. One special concern has been the elimination of passwords transmitted “in the clear”. This increased emphasis has manifested itself in the HTTP/1.1 specification.

Digest access authentication

HTTP/1.0 provides a challenge-response access control mechanism, *Basic authentication*. The origin server responds to a request for which it needs authentication with a `WWW-`

`Authenticate` header that identifies the authentication *scheme* (in this case, “Basic”) and *realm*. (The realm value allows a server to partition sets of resources into “protection spaces”, each with its own authorization database.)

The client (user agent) typically queries the user for a username and password for the realm, and then repeats the original request, this time including an `Authorization` header that contains the username and password. Assuming these credentials are acceptable to it, the origin server responds by sending the expected content. A client may continue to send the same credentials for other resources in the same realm on the same server, thus eliminating the extra overhead of the challenge and response.

A serious flaw in Basic authentication is that the username and password in the credentials are unencrypted and therefore vulnerable to network snooping. The credentials also have no time dependency, so they could be collected at leisure and used long after they were collected. *Digest access authentication* provides a simple mechanism that uses the same framework as Basic authentication while eliminating many of its flaws.

The message flow in Digest access authentication mirrors that of Basic and uses the same headers, but with a scheme of “Digest”. The server's challenge in Digest access authentication uses a nonce (one-time) value, among other information. To respond successfully, a client must compute a checksum (MD5, by default) of the username, password, nonce, HTTP method of the request, and the requested URI. Not only is the password no longer unencrypted, but the given response is correct only for a single resource and method. Thus, an attacker that can snoop on the network could only replay the request, the response for which he has already seen. Unlike with Basic authentication, obtaining these credentials does not provide access to other resources.

As with Basic authentication, the client may make further requests to the same realm and include Digest credentials, computed with the appropriate request method and request-URI. However, the origin server's nonce value may be time-dependent. The server can reject the credentials by saying the response used a stale nonce and by providing a new one. The client can then recompute its credentials without needing to ask the user for username and password again.

In addition to the straightforward authentication capability, Digest access authentication offers two other features: support for third-party authentication servers, and a limited message integrity feature (through the `Authentication-Info` header).

Proxy authentication

Some proxy servers provide service only to properly authenticated clients. This prevents, for example, other clients from stealing bandwidth from an unsuspecting proxy.

To support proxy authentication, HTTP/1.1 introduces the `Proxy-Authenticate` and `Proxy-Authorization` headers. They play the same role as the `WWW-Authenticate` and

Authorization headers in HTTP/1.0, except that the new headers are hop-by-hop, rather than end-to-end. Proxy authentication may use either of the Digest or Basic authentication schemes, but the former is preferred.

A proxy server sends the client a Proxy-Authenticate header, containing a challenge, in a 407 (Proxy Authentication Required) response. The client then repeats the initial request, but adds a Proxy-Authorization header that contains credentials appropriate to the challenge. After successful proxy authentication, a client typically sends the same Proxy-Authorization header to the proxy with each subsequent request, rather than wait to be challenged again.

Protecting the privacy of URIs

The URI of a resource often represents information that some users may view as private. Users may prefer not to have it widely known that they have visited certain sites.

The Referer header in a request provides the server with the URI of the resource from which the request-URI was obtained. This gives the server information about the user's previous page-view. To protect against unexpected privacy violations, the HTTP/1.1 specification takes pains to discourage sending the Referer header inappropriately; for example, when a user enters a URL from the keyboard, the application should not send a Referer header describing the currently-visible page, nor should a client send the Referer header in an insecure request if the referring page had been transferred securely.

State management

HTTP requests are stateless. That is, from a server's perspective, each request can ordinarily be treated as independent of any other. For Web applications, however, state can sometimes be useful.

Netscape introduced “cookies” in version 1.1 of their browser as a state management mechanism. The IETF subsequently standardized cookies in RFC2109. The basic cookie mechanism is simple. An origin server sends an arbitrary piece of (state) information to the client in its response. The client is responsible for saving the information and returning it with its next request to the origin server. RFC2109 and Netscape's original specification relax this model so that a cookie can be returned to any of a collection of related servers, rather than just to one. The specifications also restricts for which URIs on a given server the cookie may be returned. A server may assign a lifetime to a cookie, after which it is no longer used.

Cookies have both privacy and security implications. Because their content is arbitrary, cookies may contain sensitive application-dependent information. For example, they could contain credit card numbers, user names and passwords, or other personal information. Applications that send such information over unencrypted connections leave it vulnerable to snooping, and cookies stored at a client system might reveal sensitive information to another user of (or intruder into) that client.

RFC2109 proved to be controversial, primarily because of restrictions that were introduced to protect privacy. Probably the most controversial of these has to do with “unverifiable transactions” and “third-party cookies”. Consider this scenario.

1. The user visits `http://www.example1.com/home.html`
2. The returned page contains an IMG (image) tag with a reference to `http://ad.example.com/adv1.gif`, an advertisement
3. The user's browser automatically requests the image. The response includes a cookie from `ad.example.com`.
4. The user visits `http://www.exampleB.com/home.html`
5. The returned page contains an IMG tag with a reference to `http://ad.example.com/adv2.gif`.
6. The user's browser automatically requests the image, sending the previously received cookie to `ad.example.com` in the process. The response includes a new cookie from `ad.example.com`

Privacy advocates, and others, worried that:

- The user receives, in step 3, a (“third-party”) cookie from `ad.example.com`, a site she didn't even know she was going to visit (an “unverifiable transaction”)
- The first cookie gets returned to `ad.example.com` in the second image request

If a `Referer` header is sent with each of the image requests to `ad.example.com`, then that site can begin to accumulate a profile of the user's interests from the sites she visited, here `http://www.example1.com/home.html` and `http://www.exampleB.com/home.html`. Such an advertising site could potentially select advertisements that are likely to be interesting to her. While that profiling process is relatively benign in isolation, it could become more personal if the profile can also be tied to a specific real person, not just a persona. For example, this might happen if the user goes through some kind of registration at `www.example1.com`.

RFC2109 sought to limit the possible pernicious effects of cookies by requiring user agents to reject cookies that arrive from the responses to unverifiable transactions. RFC2109 further stated that user agents could be configured to accept such cookies, provided that the default was *not* to accept them. This default setting was a source of concern for advertising networks (companies that run sites like `ad.example.com` in the example) whose business model depended on cookies, and whose business blossomed in the interval between when the specification was essentially complete (July, 1996) and the time it appeared as an RFC (February, 1997). RFC2109 has undergone further refinement in response to comments, both political and technical.

Content Negotiation

Web users speak many languages and use many character sets. Some Web resources are available in several *variants* to satisfy this multiplicity. HTTP/1.0 included the notion of *content negotiation*, a mechanism by which a client can inform the server which language(s) and/or character set(s) are acceptable to the user.

Content negotiation has proved to be a contentious and confusing area. Some aspects that appeared simple at first turned out to be quite difficult to resolve. For example, although current IETF practice is to insist on explicit character set labeling in all relevant contexts, the existing HTTP practice has been to use a default character set in most contexts, but not all implementations chose the same default. The use of unlabeled defaults greatly complicates the problem of internationalizing the Web.

HTTP/1.0 provided a few features to support content negotiation, but the RFC never uses that term and devotes less than a page to the relevant protocol features. The HTTP/1.1 specification specifies these features with far greater care, and introduces a number of new concepts.

The goal of the content negotiation mechanism is to choose the best available representation of a resource. HTTP/1.1 provides two orthogonal forms of content negotiation, differing in where the choice is made:

1. In *server-driven* negotiation, the more mature form, the client sends hints about the user's preferences to the server, using headers such as `Accept-Language`, `Accept-Charset`, etc. The server then chooses the representation that best matches the preferences expressed in these headers.
2. In *agent-driven* negotiation, when the client requests a varying resource, the server replies with a 300 (Multiple Choices) response that contains a list of the available representations and a description of each representation's properties (such as its language and character set). The client (agent) then chooses one representation, either automatically or with user intervention, and resubmits the request, specifying the chosen variant.

Although the HTTP/1.1 specification reserves the `Alternates` header name for use in agent-driven negotiation, the HTTP working group never completed a specification of this header, and server-driven negotiation remains the only usable form.

Some users may speak multiple languages, but with varying degrees of fluency. Similarly, a Web resource might be available in its original language, and in several translations of varying faithfulness. HTTP introduces the use of *quality values* to express the importance or degree of acceptability of various negotiable parameters. A quality value (or *qvalue*) is a fixed-point number between 0.0 and 1.0. For example, a native speaker of English with some fluency in French, and who can impose on a Danish-speaking office-mate, might configure a browser to generate requests including

```
      Accept-Language: en, fr;q=0.5, da;q=0.1
```

Because the content-negotiation mechanism allows qvalues and wildcards, and expresses variation across many dimensions (language, character-set, content-type, and content-encoding) the automated choice of the “best available” variant can be complex and might generate unexpected outcomes

Content negotiation promises to be a fertile area for additional protocol evolution. For example, the HTTP working group recognized the utility of automatic negotiation regarding client implementation features, such as screen size, resolution, and color depth. The IETF has created the Content Negotiation working group to carry forward with work in the area.

Conclusion

HTTP/1.1 differs from HTTP/1.0 in numerous ways, both large and small. While many of these changes are clearly for the better, the protocol description has tripled in length, and many of the new features were introduced without any real experimental evaluation to back them up. The HTTP/1.1 specification also includes numerous irregularities for compatibility with the installed base of HTTP/1.0 implementations.

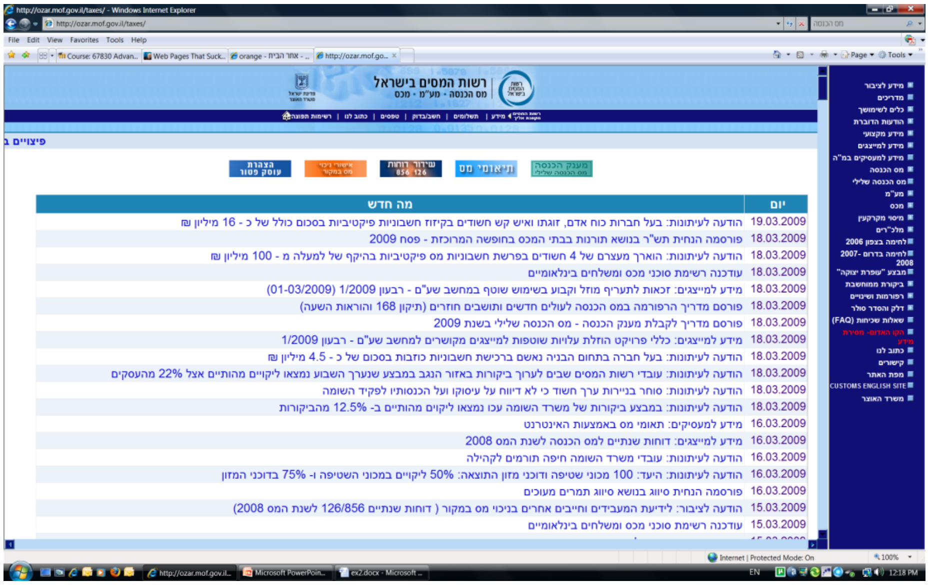
This increase in complexity complicates the job of client, server, and especially proxy cache implementers. It has already led to unexpected interactions between features, and will probably lead to others. Fortunately, the numerous provisions in HTTP/1.1 for extensibility should simplify the introduction of future modifications.

דינה זליגר

נושאים מתקדמים במסדי נתונים – תרגיל 2

שימושיות של אתרי אינטרנט

ננתח את השימוש של האתר של מס הכנסה http://ozar.mof.gov.il/taxes

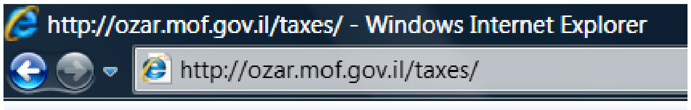


יום	מה חדש
19.03.2009	הודעה לעיתונות: בעל חברות כוח אדם, זוגתו ואיש קש חשודים בקיזוז חשבוניות פיקטיביות בסכום כולל של כ - 16 מיליון ₪
18.03.2009	פורסמה הנחית תש"ר בנושא תורנות בבתי המכס בחופשה המרוכזת - פסח 2009
18.03.2009	הודעה לעיתונות: הוארך מעצרם של 4 חשודים בפרשת חשבוניות מס פיקטיביות בהיקף של למעלה מ - 100 מיליון ₪
18.03.2009	עודכנה רשימת סוכני מכס ומשלחים בינלאומיים
18.03.2009	מידע למייצגים: זכאות לתעריף מוזל וקבוע בשימוש שוטף במחשב שע"ם - רבעון 1/2009 (01-03/2009)
18.03.2009	פורסם מדריך הרפורמה במס הכנסה לעולים חדשים ותושבים חוזרים (תיקון 168 והוראות השעה)
18.03.2009	פורסם מדריך לקבלת מענק הכנסה - מס הכנסה שלילי בשנת 2009
18.03.2009	מידע למייצגים: כללי פרויקט הוזלת עלויות שוטפות למייצגים מקושרים למחשב שע"ם - רבעון 1/2009
18.03.2009	הודעה לעיתונות: בעל חברה בתחום הבניה נאשם ברכישת חשבוניות כוזבות בסכום של כ - 4.5 מיליון ₪
18.03.2009	הודעה לעיתונות: עובדי רשות המסים שבים לערוך ביקורות באזור הנגב במבצע שנערך השבוע נמצאו ליקויים מהותיים אצל 22% מהעסקים
18.03.2009	הודעה לעיתונות: סוחר בניירות ערך חשוד כי לא דיווח על עיסוקו ועל הכנסותיו לפקיד השומה
18.03.2009	הודעה לעיתונות: במבצע ביקורות של משרד השומה עכו נמצאו ליקוים מהותיים ב- 12.5% מהביקורות
16.03.2009	מידע למעסיקים: תאומי מס באמצעות האינטרנט
16.03.2009	מידע למייצגים: דוחות שנתיים למס הכנסה לשנת המס 2008
16.03.2009	הודעה לעיתונות: עובדי משרד השומה חיפה תורמים לקהילה
16.03.2009	הודעה לעיתונות: היעד: 100 מכוני שטיפה ודוכני מזון התוצאה: 50% ליקויים במכוני השטיפה ו- 75% בדוכני המזון
16.03.2009	פורסמה הנחית סיווג בנושא סיווג תמרים מעוכים
15.03.2009	הודעה לציבור: לידיעת המעבידים וחייבים אחרים בניכוי מס במקור (דוחות שנתיים 126/856 לשנת המס 2008)
15.03.2009	עודכנה רשימת סוכני מכס ומשלחים בינלאומיים

__בעיה 0:__ בעיית כיעור כללי!!!

__תיקון__: להחליף את האנשים שאחראים על האתר!

__בעיה 1__: הכותרת של העמוד אינה מכילה תוכן מועיל, אלא פשוט את הכתובת של האתר– לא טוב אם משתמשים בסימניות..

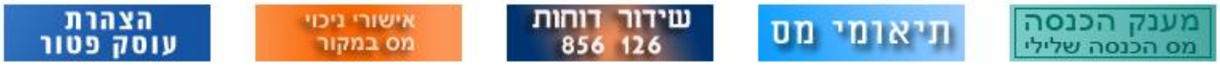

http://ozar.mof.gov.il/taxes/ - Windows Internet Explorer
http://ozar.mof.gov.il/taxes/

__תיקון__: לשנות את הכותרת של העמוד למשהו אינפורמטיבי; למשל "רשות המסים בישראל".

__בעיה 2__: אין אפשרות חיפוש באתר (או לפחות אני לא מצאתי אפשרות כזאת...)

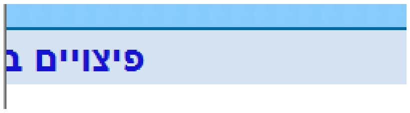
__תיקון__: להוסיף יכולת חיפוש (טפסים, הודעות, מידע וכו')

<u>בעיה 3</u>: הלינקים עושים כאב ראש. לא רואים את זה בתמונה אבל הם גם מרצדים צבעים שונים. הגופנים לא תואמים (לזכותם ייאמר שמאחורי הלינקים מופיע טקסט אז לפחות זה נגיש לעיוורים)



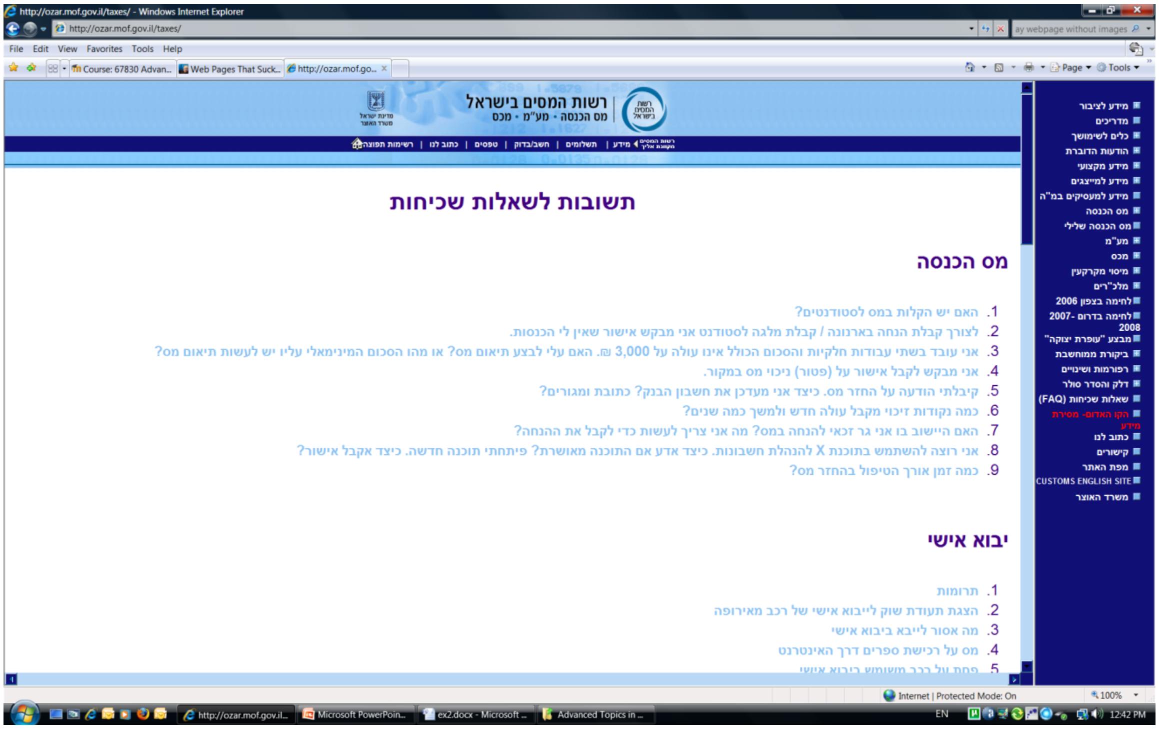
<u>תיקון</u>: לתת לכל הלינקים עיצוב זהה עם רקע אחיד ולא משתנה

<u>בעיה 4</u>: marquee עם לינקים – לוקח המון זמן עד שהם מגיעים וגם אפשר לפספס ואז צריך לחכות שובץ ממש מעצבן!!!



<u>תיקון</u>: לבטל את ה-marquee לאלתר ולכתוב אל הלינקים בצורה מסודרת כך שלא יברחו לשום מקום..

<u>בעיה 5</u>: לינקים בצבעים לא קריאים. קשה מאוד לקרוא את התכלת על הרקע הלבן מה גם שלינקים לא נראים ככה באופן סטנדרטי...



תשובות לשאלות שכיחות

מס הכנסה

1. האם יש הקלות במס לסטודנטים?
2. לצורך קבלת הנחה בארנונה / קבלת מלגה לסטודנט אני מבקש אישור שאין לי הכנסות.
3. אני עובד בשתי עבודות חלקיות והסכום הכולל אינו עולה על 3,000 ₪. האם עלי לבצע תיאום מס? או מהו הסכום המינימאלי עליו יש לעשות תיאום מס?
4. אני מבקש לקבל אישור על (פטור) ניכוי מס במקור.
5. קיבלתי הודעה על החזר מס. כיצד אני מעדכן את חשבון הבנק? כתובת ומגורים?
6. כמה נקודות זיכוי מקבל עולה חדש ולמשך כמה שנים?
7. האם היישוב בו אני גר זכאי להנחה במס? מה אני צריך לעשות כדי לקבל את ההנחה?
8. אני רוצה להשתמש בתוכנת X להנהלת חשבונות. כיצד אדע אם התוכנה מאושרת? פיתחתי תוכנה חדשה. כיצד אקבל אישור?
9. כמה זמן אורך הטיפול בהחזר מס?

יבוא אישי

1. תרומות
2. הצגת תעודת שוק לייבוא אישי של רכב מאירופה
3. מה אסור לייבא ביבוא אישי
4. מס על רכישת ספרים דרך האינטרנט
5. פחת על רכב משומש ביבוא אישי

<u>תיקון</u>: הטקסט צריך להיות בצבע כהה יותר.

Dina Zeliger

ADVANCED TOPICS IN DB THEORY – EXERCISE 3

Compressed Tries

WHAT IS A TRIE?

A **trie**, or **prefix tree**, is an ordered tree data structure that is used to store an associative array where they keys are usually strings. Unlike a binary search tree, no node in a trie stores the key associated with that node. Instead, its position in the tree shows what key is associated with it. All the descendents of a node have a common prefix of the string associated with that node, and the root is associated with the empty string. Strings are usually considered null-terminated, and thus no string can be the prefix of another. Note that without the null-termination, a string CAN be the suffix of another string (i.e. "the girl" and "the girl ate").

Though it is most common, tries need not be keyed by character strings. The same algorithms can be easily adapted to serve similar functions of ordered lists of any construct.

ADVANTAGES AND DISADVANTAGES

- Generally, when the total set of stored keys is very sparse within their representation space, the trie data structure is quite wasteful in storage.
- Comparing to a regular BST, tries have several advantages:
    - Looking up a key of length m takes $O(m)$ time at worst case. A balanced BST performs $O(\log n)$ comparisons where n is the number of elements in the tree. Thus, in the worst case, a BST takes $O(m \log n)$ time.
    - Tries can require less space when they contain a large number of short strings, because the keys are not stored explicitly and nodes are shared between keys with common initial subsequences.
    - Tries help with longest-prefix matching[1], where we wish to find the key sharing the longest possible prefix of characters all unique.
- Comparing to hash tables:
    - Advantages:
        - Looking up data in a trie is faster in the worst case, $O(m)$ time, compared to an imperfect hash table. An imperfect hash table can have key collisions. The worst-case lookup speed in an imperfect hash table is $O(n)$ time, but far more typically is O(1), with O(m) time spent evaluating the hash.
        - There are no collisions of different keys in a trie.

[1] **Longest prefix match** is an algorithm used by routers in Internet Protocol networking to select an entry from a routing table. Because each entry in a routing table may specify a network, one destination address may match more than one routing table entry. The most specific table entry – the one with the highest subnet mask – is called the longest prefix match. It is called this because it is also the entry where the largest number of leading address bits in the table entry match those of the destination address.

        - Buckets in a trie which are analogous to hash table buckets that store key collisions are only necessary if a single key is associated with more than one value.
        - There is no need to provide a hash function or to change hash functions as more keys are added to a trie.
        - A trie can provide an alphabetical ordering of the entries by key.
    - Disadvantages:
        - Tries can be slower in some cases than hash tables for looking up data, especially if the data is directly accessed on a hard disk drive or some other secondary storage device where the random access time is high compared to main memory.
        - It is not easy to represent all keys as strings, such as floating point numbers, which can have multiple string representations for the same floating point number, e.g. 1, 1.0, 1.00, +1.0, etc.

Overcoming the Disadvantages – Compressed Tries

When the trie is mostly static, i.e. all insertions or deletions of keys from a prefilled trie are disabled and only lookups are needed, and when the trie nodes are not keyed by node specific data (or if the node's data is common) it is possible to compress the trie representation. This application is typically used for compressing lookup tables when the total set of stored keys is very sparse within their representation space. Another option is to do updating in batches, whenever the batch size exceeds a threshold.

Whereas each field in a trie has to be large enough to hold a pointer, each field in a C-trie is represented by a single bit. If the k-th bit of a node N on level l is set, it indicates that one or more keys pass through this node N and have as $(l+1)$-th substring $x_{l+1} = k$.

The structure of a node N in level l of a C-trie is as follows:

- U – 1-bit field. If $U = 0$, then the node is internal and B, K, C are as follows. Otherwise, the node is a leaf and B, K, C contain the suffix $x_{l+1} \dots x_k$ of the key $x = x_1 \dots x_k$. Note that the prefix does not have to be stored since it is implicitly defined by the path from the root of the C-trie to the leaf.
- B – 1-bit field. B is set to 1 if one of the keys passing through the node terminates at level l and thus corresponds to a blank field in a trie which contains a key.
- K – $(m-1)$-bit field, each corresponding to a field in a node of a trie. m is the number of characters plus the terminating sign.
- C – at most $\lceil \log_2 n \rceil$ bits where n is the number of keys. C is equal to the number of nonzero bits of the K fields in the nodes on level l to the left of the node N. This number equals the number of nodes on level $l+1$ preceding the successor of the first nonzero bit in the field K of node N.

All the nodes are stored as a continuous bit string. First is the root node, which is followed by all the nodes in level 1 from left to right, and so on.

BIBLIOGRAPHY

1. http://en.wikipedia.org/wiki/Trie
2. http://en.wikipedia.org/wiki/Longest_prefix_match
3. Compressed Tries, Kurt Maly, Communications of the ACM 1976

דינה זוימר

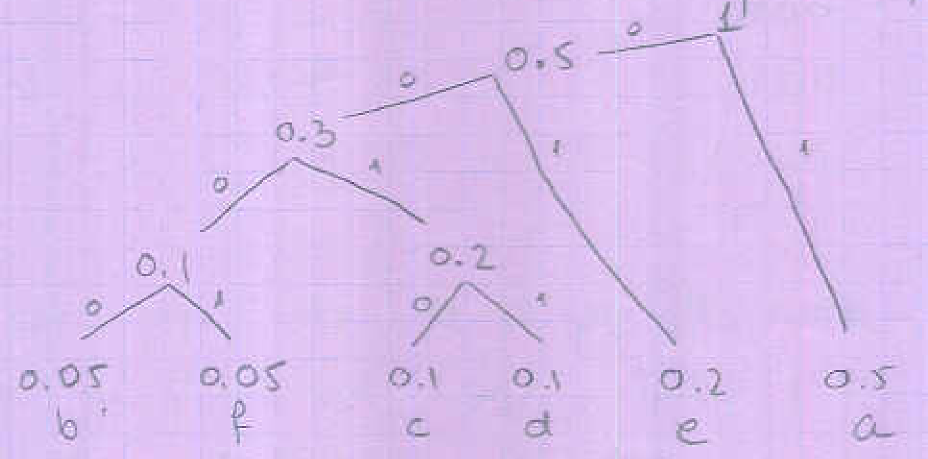
[redacted]

נושאים מתקדמים במבני נתונים - תרגיל 4

נתבונן בא"ב הבא:

character	probability
a	0.5
b	0.05
c	0.1
d	0.1
e	0.2
f	0.05

נמצא קוד האפמן עבור זה.



אז קוד האפמן יהיה

a - 1

b - 0000

c - 0010

d - 0011

e - 01

f - 0001

נמצא קוד האפמן קנוני עבור הנ"ל. קודם כל נסדר את הקוד לפי אורך (ואז לפי סדר הא"ב)

a - 1

e - 01

b - 0000

c - 0010

d - 0011

f - 0001

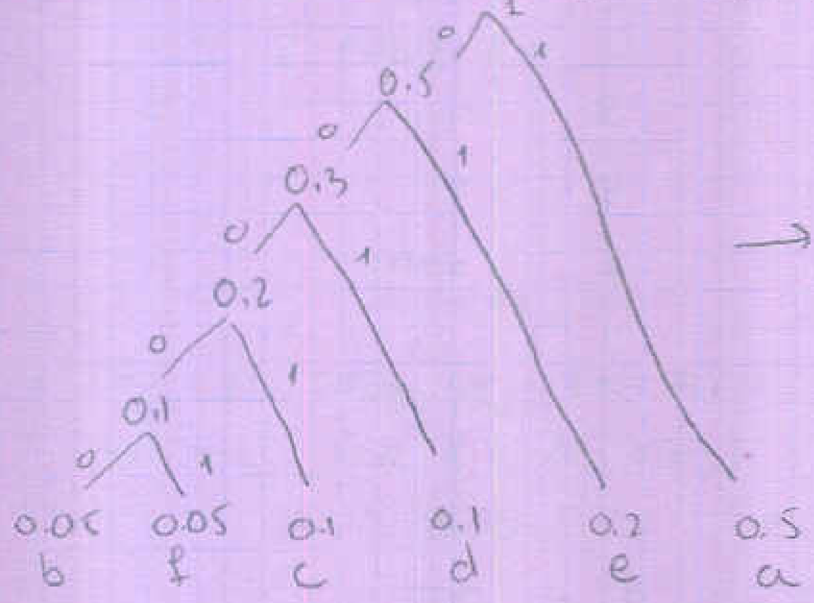
ועכשיו נהפוך את זה לקוד האפמן קנוני ע"י האלגוריתם הבא:

- הסמל הראשון ברשימה מקבל קוד באותו אורך כמו הקוד המקורי אלא שכולו אפסים.
- כל סמל עוקב מקבל את הערך הבינארי הבא
- כשמגיעים לסמל עם קוד ארוך יותר, אחרי המעבר לערך הבינארי הבא, מרפדים אותו מימין באפסים עד שמגיעים לאורך בקוד המקורי.

ככה נקבל

a	1	→	a	0
e	01		e	10
b	0000		b	1100
c	0010		c	1101
d	0011		d	1110
f	0001		f	1111

קוד האפמן קנוני אינו יחיד אם יש שני קודי האפמן (רגילים) שונים לאותו תו קידוד באורך שונה. למשל, נתבונן בדוגמה מסעיף ב. אפשר לעשות עץ אחר



→

a	1
b	00000
c	0001
d	001
e	01
f	00001

ברור שהקוד הקנוני המתאים יהיה שונה מהקוד הקודם כי האורכים של הקודים שונים!!!

דינה רלימר
[redacted]

נושאים מתקדמים במסדי נתונים - תרגיל 5

נציע מבנה נתונים לשאילת אינדקס biword
הרעיון די פשוט: במקום להתייחס לכל מילה בנפרד פשוט
נתייחס לזוגות מילים סדורים כמילה אחת - כ- key יחודי באינדקס.

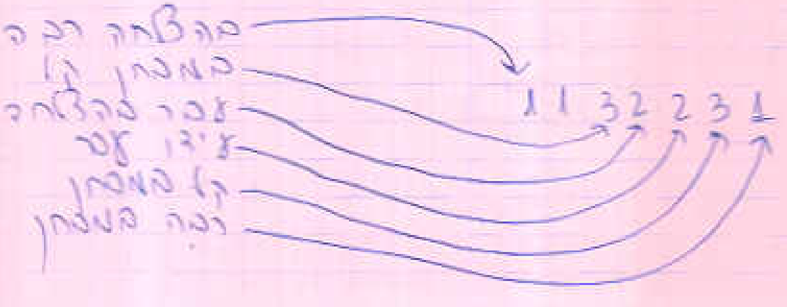
אם יש n זוגות מילים אז במבנה הנתונים יש
מערך n פוינטרים ממוין לפי א"ב של מצביעים לתוך
רשימה של מספרים שמייצגים את רשימות ההפרשים
של ה inverted-index של זוגות המילים.

נניח שהמסמכים הם
1) בהצלחה רבה במבחן
2) עידו עבר בהצלחה רבה
3) קל במבחן קל.

אז כל הזוגות השונים ממוינים הם:

זוג מילים	מסמכים	רשימת הפרשים
בהצלחה רבה	1,2	1,1
במבחן קל	3	3
עבר בהצלחה	2	2
עידו עבר	2	2
קל במבחן	3	3
רבה במבחן	1	1

מבנה הנתונים:



לא כדאי לשמור את הפוינטרים של זוגות המילים [redacted]
האפשריים כי במקרי גדול הרבה זוגות לא מופיעים וזה
זה בזבזני מאוד.

החיפוש במבנה הוא כמובן לוגריתמי במספר זוגות
המילים. עושים חיפוש בינארי ברשימת הפוינטרים
ואח"כ זה דבר תלוי באורך הרשימה. ברור שבמקרה
הגרוע בנוסף לחיפוש הבינארי צריך עוד לעבור על רשימה
שהיא באורך מספר המאמרים (אם הזוג מופיע בכל
המאמרים)

אם יש n זוגות מילים, k מאמרים
מילה ממוצעת היא באורך l תווים
פוינטר תופס p תווים
הזוג הכי נפוץ מופיע ב- m מאמרים אז

טבלת הפוינטרים: $\underbrace{p\cdot n}_{\text{הפוינטרים}} + \underbrace{n\cdot l\cdot 2}_{\text{שמירת הזוגות}}$

טבלת ההפרשים: $n\cdot k\cdot \log_2 k$

מספר הרשימות

~~מס~~ אורך הרשימה

אורך ייצוג מספר

<u>הערה</u>: זה פתרון יעיל בזמן אבל הבעיה היא שהוא לא scalable
במקרה שנרצה לעשות חיפוש של יותר ממילה או מילים שאינן
צמודות. פתרון שאפשר להרחיב שמור אינדקס של מילים
ומאמרים אבל לא רק מספרים אלא גם את המיקום של המילה
במאמר. זה יותר בזבזני במקום ובזמן השליפה במקרה של
biword, אבל זה פתרון הרבה יותר גמיש!!

דינה דומר
[redacted]

נושאים מתקדמים במסדי נתונים - תרגיל 6

האלגוריתם שמוצג במאמר מאפשר למצוא חוקי DUST
עבור אתר אינטרנט כלשהו בלי להסתכל על הדפים עצמם.
האלגוריתם עובר על רשימת הכתובות של האתר ומתמקד
במציאת חוקים מהצורה $\beta \to \alpha$. כלומר בכתובת
האחרונה α מוחלפת ב- β.
האלגוריתם מתבסס על שלושה כללי אצבע
- אם $\beta \to \alpha$ חוק אמיתי אז יהיו לו הרבה דוגמאות
- אם $\beta \to \alpha$ אינו אמיתי אז הוא יפיע בקבוצות
גדולות, כמו למשל מספרים בשמות מסמכים או
תמונות
- למסמכים דומים גדלים דומים (אם משתמשים ב-
web logs) או תקצירים דומים (אם יש לנו
תוצאות של crawl קודם).

לטענת המחברים שימוש בכללי אצבע אלה מוריד משמעותית את
הכפילויות בהצלחה.
נציין רק שאחר שהאלגוריתם מוצא רשימת מועמדים הוא
מוודא אותם ע"י דגימה קטנה של דפים שהוא מוריד מה
רשת.

Do Not Crawl in the DUST: Different URLs with Similar Text

ZIV BAR-YOSSEF, IDIT KEIDAR, URI SCHONFELD

PRESENTED BY DINA ZELIGER

DUST: A Problem

- Different URLs with Similar Text
- Examples:
  - Standard Canonization:
    - http://domain.name/index.html → http://domain.name
  - Domain names and virtual hosts
    - http://news.google.com → http://google.com/news
  - Aliases and symbolic links:
    - http://domain.name/~shuri → http://domain.name/people/shuri
  - Parameters with little affect on content
  - URL transformations:
    - http://domain.name/story_ → http://domain.name/story?id=

Why Care?

- Expensive to crawl
  - Access the same document via multiple URLs
- Forces us to shingle
  - An expensive technique used to discover similar documents
- Ranking algorithms suffer
  - References to a document split among its aliases
- Multiple identical results
  - The same document is returned several times in the search results

DUST Buster Basic Framework

- Input: URL list
- Detect likely DUST rules
- Eliminate redundant rules
- Validate DUST rules using samples:
  - Eliminate DUST rules that are “wrong”
  - Further eliminate duplicate DUST rules
- First three phases DO NOT require fetching!!

DUST Buster Heuristics

- Large support principle:
  - Likely DUST rules have lots of “evidence” supporting them
  - A pair of URLs (u,v) is an instance of rule r, if r(u) = v
  - Support(r) = all instances (u,v) of r
  - Envelope of string a in URL u is pair (p,s) s.t. u = pas
  - E(a) = all envelopes of a in URLs in URL list
  - $|\text{support}(a \rightarrow b)| = |E(a) \cap E(b)|$
- Small buckets principle:
  - Ignore evidence that supports many different rules
  - $\text{Bucket}(p,s) = \{a \mid (p,s) \in E(a)\}$
  - Often do not contain similar content

More Likeliness Evidence

- Similar texts have similar sizes
  - When using web server logs
- Similar texts have similar sketches
  - When using previous crawl

Eliminating Redundant Rules

- Rule r refines rule q if support(r) $\subseteq$ support(q)
  - "/vlsi/" → "/labs/vlsi/"
  - "/vlsi" → "/labs/vlsi"
- A substitution rule a' → b' refines rule a → b if and only if there exists an envelope (c,d) such that a'=cad and b'=cbd
  - Helps identify refinements easily
- If r refines q, remove q if supports match
  - Refinement gives the full context of the substitution

Validating rules

- Fetch random SMALL sample of pages
- Check if fraction of similar pages exceeds threshold
- Why not dismiss rule with even one bad example?
  - A 95% valid rule may still be worth it
  - Dynamic pages change often

Dina Zeliger
[redacted]

Advanced Topics in DB Theory
Hilltop Algorithm

PageRank is based on connectivity. It assumes that authoritative pages tend to point to other authoritative pages. Moreover, it is query-independent. It cannot by itself distinguish between pages that are authoritative in general and pages that are authoritative on the query topic.

This is the problem that the Hilltop algorithm is trying to solve. ~~The bene~~ Hilltop is very similar to PageRank, but the benefit of Hilltop over raw PageRank is that it is topic-sensitive, and thus it is harder to manipulate (it is easy to mislead ~~seac~~ PageRank by building link farms)

Hilltop relies on the same assumption that the number and quality of the sources referring to a page are a good measure of the page's quality. However, Hilltop only considers "expert" sources - pages that have been created with the sole purpose of directing people towards resources. In response to a query, it first computes a list of the most relevant experts on the query topic. ~~Hilltop also requires that it can easily locate at least 2 expert documents~~ Then it continues in a similar manner to PageRank. However, it also requres that it can easily locate at least 2 expert documents voting for the same Web page. Thus if a pool of experts is not available or if Hilltop cannot find a minimum of expert pages for the query topic, the results ~~will~~ it will return will be absolute zero. So HillTop is tuned for result accuracy and not query coverage. This is the main disadvantage of the algorithm.

Since the ranking phase is similar to that of PageRank (using connectivity analysis) we'll focus on the expert lookup.
The authors felt that expert pages need to be objective and diverse. Therefore, in order to find the experts one needs to detect when to sites belong to the same organization.

Two hosts are said to be affiliated if

- They share the same first 3 octets of their IP addresses. or
- The rightmost non-generic token in their

hostnames are the same. In a preprocessing step it is possible to construct a host affiliation lookup - group hosts that are considered affiliated. Then to find the experts, Hilltop crawls a search engine's database. Considering all pages with out-degree greater that some threshold k, it tests to see if these URLs point to k distinct non-affiliated hosts. Such pages are considered experts.

To locate expert pages that match queries there's an inverted index that maps keywords to experts ~~on whic~~. But this is done only on indexed text contained in "key phrases" - pieces of text that qualify one or more URLs in the page.

In response to a user query, Hilltop first computes a list of N experts that are most relevant for that query. Then, it ranks results by selectively following the relevant links from these experts, and assigning an authority score to each such page.

hostnames are the same. In a preprocessing step it is possible to construct a list of affiliation, locally group hosts that are considered affiliated. Then to find the experts, Hilltop consults a search engine's database. Considering all pages with out-degree greater than some threshold k, it tests to see if these URLs point to k distinct non-affiliated hosts. Such pages are considered experts.

To build expert pages' inverted index, queries leads on search index that maps keywords to expert pages. But this is keyed only on unique text contained in "key phrases" — a piece of text that qualify one or more URLs in the page.

In response to a user query, Hilltop first computes a list of N experts that are most relevant for that query. Then it ranks results by selectively following the relevant links from these experts, and assigning an authority score to each such page.

דנה אייר
[redacted]

נושאים מתקדמים במסדי נתונים
תרגיל 9

נתבונן ב- DTD :

```
<! DOCTYPE a [
        <!ELEMENT a (d+|b)? >
        <!ELEMENT b (c,f) >
        <!ELEMENT c (a) >
        <!ELEMENT e (d) >
        <!ELEMENT d (f) >
        <!ELEMENT f (g,h) >
        <!ELEMENT g (#PCDATA) >
        <!ELEMENT h (#PCDATA)> ]>
```

נפשט באמצעות הכללים :

$$(d+ | b)? \xrightarrow{e_1+ \to e_1*} (d* | b)? \xrightarrow[(e_1|e_2)? \to e_1?, e_2?]{} (d*)?, b?$$

$$\xrightarrow[e_1*? \to e_1*]{} d*, b?$$

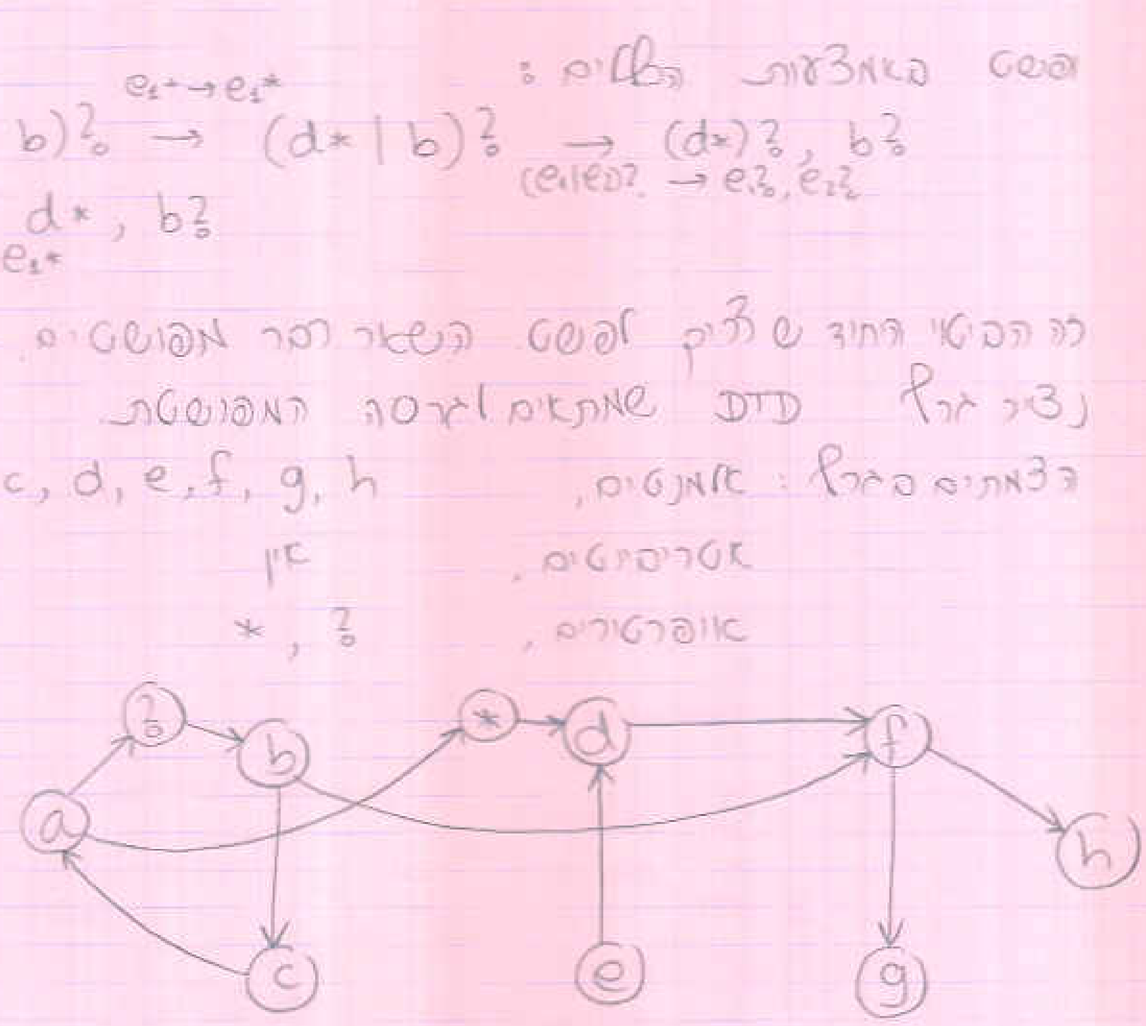
זה הביטוי היחיד שצריך לפשט. השאר כבר מפושטים.

נצייר גרף DTD שמתאים לגרסה המפושטת.

הצמתים בגרף : אלמנטים, a, b, c, d, e, f, g, h

אטריבוטים, אין

אופרטורים, *, ?



נראה את הסכמה שנוצרת ע"י ה-

Shared Inline Technique

יוצרים יחס (טבלה) עבור כל

- אלמנט עם דרגת כניסה גדולה מ-1
- אלמנט עם דרגת כניסה 0
- אלמנט מתחת ל-*
- אלמנט אחד מכל קבוצה רקורסיבית, בעם דרגת כניסה 1

כל הצמתים, של אלמנטים כאלה עם דרגת כניסה 1 (כניסה) יחס הדדיים: חוץ מאלה שמתחת ל-*

each element node x that is a separate relation inlines all the elements y that are reachable from it such that the path from x to y does not contain a node (other than x)

אלמנט	דרגת כניסה		ילדים
a	1		b?, d*
b	1		c, f
c	1		a
✓ d	2	←	f
✓ e	0	←	d
f	2	←	g, h
g	1		-
h	1		-

חוץ מזה יש מעגל אחד מגרף - a→?→b→c→a

; d - (נמצא מתחת ל-*)

כל יחס שומרים את הבאים:

- Id של הצומת
- התוכן של כל הילדים המקושרים - אלה עם דרגת כניסה 1
- עבור צמתים עם θ ע נכנסת
  - Id של ההורה
  - Code של ההורה

לכן במקרה שלנו (נקבל) את הטבלאות הבאות:

d (dID: integer,
    d.parentID: integer, d.parentCode: integer)

e (eID: integer)

f (fID: integer, f.g: string, f.h: string,
    f.parentID: integer, f.parentCode integer)

רגע נחזור לדיון עבור המעגל a→b→c→a.

הבחירה ב- a אינה טובה משום שלא ידוע כמה ילדים יכולים להיות לו.

הבחירה ב- c לא טובה כי אסור לא להיות לו אבא.

לכן נותרנו רק עם האופציה של b

b (bID: integer, ~~b.a: string b.c: string~~
    b.parentID: integer, b.parentCode integer)

נראה את הסכמה שנוצרת ע"י Hybrid Inlining Technique

כאן הגישה דומה ל- shared אלא שמטפלת גם בכללים
גם ילדים שיש להם דרגת כניסה גדולה מ- 1 והתנאי שלהם
לא רקורסיביים או אחרת)·*

d (dID: integer , d.f.g: string, d.f.h: string
    d.parentID: integer , d.parentCode: integer)
e (eID: integer , e.d.f.h: string, e.d.f.g: string)
f (fID: integer , f.g: string, f.h: string
    f.parentID: integer, f.parentCode: integer)
b (bID: integer, b.f.g: string, b.f.h: string,
    b.parentID: integer , b.parentCode: integer)