# Data Update Propagation: A Method for Determining How Changes to Underlying Data Affect Cached Objects on the Web

Arun Iyengar and Jim Challenger
IBM Research
T. J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598

## 1 Introduction

Caching is often used to improve performance on the Web. One of the problems with caching Web pages is determining how cached Web pages are affected by changes to underlying data which affect the values of the pages. For example, a set of several cached Web pages may be constructed from tables belonging to a database. In this situation, a method is needed to determine which Web pages are affected by updates to the database. That way, caches can be synchronized with databases so that they do not contain stale data. Furthermore, the method should associate cached pages with parts of the database in as precise a fashion as possible. Otherwise, objects whose values have not changed may be mistakenly invalidated or updated from a cache after a database change. Such unnecessary updates to caches can increase miss rates and hurt performance.

This paper presents a method for precisely specifying data dependencies between cached objects and underlying data which are constantly changing and affect the values of cached objects. Our algorithm, which we call *Data Update Propagation (DUP),* can be generalized to hierarchies of dependencies where an object has a data dependence on other entities which in turn have dependencies on other entities. In addition, DUP provides a metric for assessing how obsolete an object is. This metric is useful when it is acceptable for a page to be slightly obsolete. Retaining slightly obsolete pages results in better performance than always maintaining current versions of pages. At some point, however, an obsolete page must be replaced, and DUP provides a method for determining when a page should be replaced.

DUP has been implemented in IBM's DynamicWeb Cache which is particularly well suited for caching dynamic Web pages [3, 2]. The DynamicWeb Cache is being used at several real Web sites and is being incorporated into IBM's net.Data software product.

## 2 The Basic Data Update Propagation (DUP) Algorithm

The basic approach is to maintain correspondences between *objects* which are defined as items which may be cached and *underlying data* which periodically change and affect the values of objects. Although an entity may be both an object as well as underlying data, objects and underlying data could also be different, which would mean that underlying data are not cacheable. For example, the DynamicWeb Cache typically only caches dynamic HTML pages. The objects are thus HTML pages. Such objects are often dependent on the contents of databases. In this situation, the databases would constitute underlying data. If a cache is only storing HTML pages and not portions of the database, the underlying data and objects are disjoint.

Storage for a cache is managed by a long-running process known as the *cache manager.* A cache manager maintains data dependence information between objects and underlying data. When the cache manager becomes aware of a change to underlying data, it queries the dependence information which it has stored in order to determine which cached objects are affected. Cache managers use dependency information to determine which objects should be invalidated as a result of changes to underlying data.

An application program is responsible for communicating data dependencies between underlying data and objects to the cache manager. Such dependencies can be represented by a directed graph known as an *object dependence graph,* wherein a vertex usually represents an object or underlying data. An edge from a vertex $v$ to another vertex $u$ denoted $(v, u)$ indicates that a change to $v$ also affects $u$. Node $v$ is known as the *source* of the edge, while $u$ is known as the *target* of the edge. For example, if node $go2$ in Figure 1 changes, then nodes $go5$ and $go6$ also change. By transitivity, $go7$ also changes. Edges may optionally have weights associated with them which indicate the importance of data dependencies. In Figure 1, the data dependence from $go1$ to $go5$ is more important than the data dependence from $go2$ to $go5$ because the former edge has a weight which is 5 times the weight of the latter edge.

In many cases we have encountered, the object dependence graph is a *simple object dependence graph* having the following characteristics:

- Each vertex representing underlying data is a leaf node.
- Each vertex representing an object does not have an outgoing edge.
- All vertices in the graph correspond to underlying data (leaf nodes) or objects (nodes with no outgoing edges).
- None of the edges have weights associated with them.

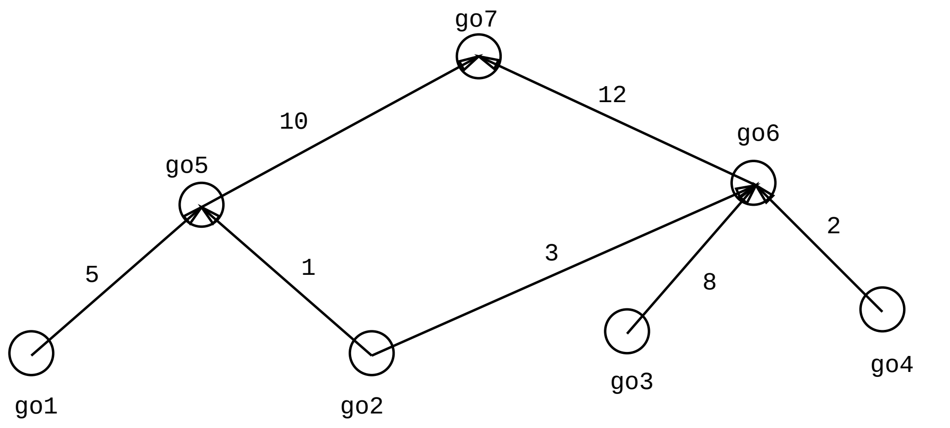

Figure 1: An object dependence graph. Weights are correlated with the importance of data dependencies.

Figure 2 depicts a simple object dependence graph. We first show how DUP works for simple object dependence graphs. We will subsequently show how DUP can be generalized to any object dependence graph.

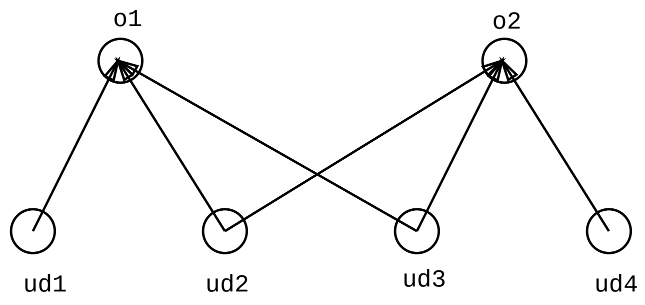

Figure 2: A simple object dependence graph.

The application program must determine an appropriate correspondence between underlying data and vertices of the object dependence graph $G$. For example, a vertex corresponding to underlying data may represent a database table. Another vertex corresponding to underlying data may represent portions of several database tables. There are no restrictions on how underlying data may be correlated with nodes of G. The application program has freedom to pick the most logical and/or efficient system.

Each object has a string *obj_id* known as the object ID which identifies the object. Similarly, each node representing underlying data has a string *ud_id* known as the underlying data ID which identifies it. An application program informs the cache manager that an object has a dependency on underlying data via an

API function:

$$add\_dependency(obj\_id, ud\_id)$$

Whenever underlying data corresponding to a node in G changes, the application program notifies the cache manager via an API function:

$$underlying\_data\_has\_changed(ud\_id)$$

The cache manager then invalidates all cached objects having dependencies on *ud_id*. Referring to Figure 2,

$$underlying\_data\_has\_changed(ud4)$$

would cause $o2$ to be invalidated. The function call:

$$underlying\_data\_has\_changed(ud2)$$

would cause both $o1$ and $o2$ to be invalidated.

## 2.1 Implementation

This section describes how simple object dependence graphs are implemented in the DynamicWeb cache. Each cache has a cache directory maintained by the cache manager. The cache directory contains information about each object in the cache. Directory information for each object with one or more dependencies on underlying data includes the object ID and an *incoming adjacency list* containing all underlying data ID's corresponding to underlying data which affect the value of the object. Figure 3 depicts the incoming adjacency lists corresponding to the graph in Figure 2.

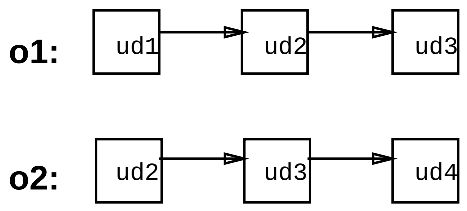

Figure 3: Incoming adjacency lists corresponding to the graph in Figure 2.

The cache manager also maintains a hash table containing pointers to *outgoing adjacency lists* for nodes in $G$ corresponding to underlying data. The hash table is indexed by underlying data ID's. Each outgoing adjacency list contains the object ID's of objects whose values depend on the underlying data represented by the underlying data ID. Figure 4 depicts the hash table corresponding to the graph in Figure 2.

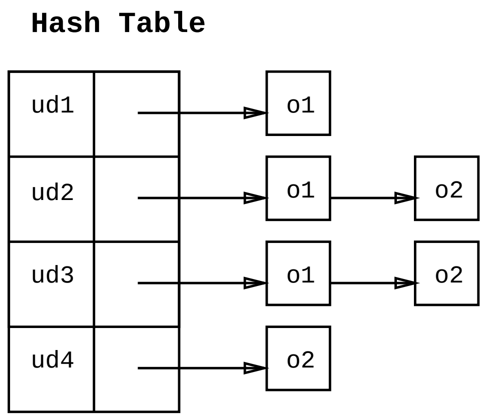

Figure 4: The hash table corresponding to the graph in Figure 2.

An invocation of the API function:

$$add\_dependency(obj\_id, ud\_id)$$

adds a new edge to $G$ by adding *obj_id* to the outgoing adjacency list for *ud_id* and *ud_id* to the incoming adjacency list for *obj_id*.

An invocation of the API function:

$$underlying\_data\_has\_changed(ud\_id)$$

is implemented by invalidating all objects on the outgoing adjacency list for *ud_id*.

It is sometimes desirable to delete nodes from $G$. For example, all dependencies on an underlying data node could become obsolete in which case the node would no longer be needed. Similarly, an object could go away which would make its node in $G$ unnecessary. Object nodes are removed from $G$ by removing the object ID from outgoing adjacency lists for all nodes on the incoming adjacency list for the object and removing the incoming adjacency list for the object. Underlying data nodes are removed from $G$ by removing the underlying data ID from incoming adjacency lists for all nodes on the outgoing adjacency list for the underlying data node and removing the hash table entry for the underlying data node.

## 3 Generalizing DUP to Arbitrary Object Dependence Graphs

### 3.1 Overview

We now present the generalized DUP algorithm which provides a number of enhancements over the version just presented:

- The generalized DUP algorithm is applicable when the object dependence graph is not simple. In general, a node may have both incoming and outgoing edges as in Figure 1. It is also possible for a graph to have cycles. A cycle would imply that a change to any node in the cycle would result in a change to all nodes comprising the cycle.

  It is not necessary for a node to correspond to an object or underlying data. A node which does not correspond to an object or to underlying data is said to represent a *virtual object.* The purpose of a virtual object is usually to propagate change information to nodes representing real objects.

- In some cases, it is acceptable for cached objects to be slightly out of date. Retaining out of date objects in a cache can be cheaper than always updating or invalidating objects after they change. At some point, an object will become highly obsolete and will have to be invalidated or updated in the cache. A quantitative method is needed for determining when a cached object has become highly obsolete. The generalized DUP algorithm provides a metric for determining how obsolete a cached object is. In order to enable the metric, edges of the object dependence graph should have weights which are correlated with the importance of dependencies such as in Figure 1.

- A cache manager might be managing multiple caches. In this situation, the generalized DUP algorithm allows a single object dependence graph to be applied to multiple caches. Different caches may concurrently be storing different versions of the same object.

- The generalized DUP algorithm provides a metric for quantitatively assessing how similar two versions of the same object are. This is particularly useful when different versions of the same object are being stored in different caches.

- In some situations, a cache $c1$ may contain obsolete versions of an entire set $S$ of objects as long as all objects are consistent with each other. Whenever a new version of an object $o1$ in $S$ is added to $c1$, any other object from $S$ in $c1$ which is inconsistent with $o1$ must be updated or invalidated. Under our definition, two objects are consistent with each other if either:

  1. Both of them are current.
  2. At some point in the past, both objects were current.

  The generalized DUP algorithm allows an application program to specify that a set of objects should be consistent with each other. The object manager subsequently minimizes the number of updates or invalidations which are required without violating consistency constraints. Without DUP's feature for satisfying consistency constraints, the most obvious method for ensuring consistency would be to always make sure that objects contained in $c1$ from $S$ are always current. DUP often results in fewer updates and invalidations and hence better performance.

- Web pages often have dependencies on parts of relational database tables. If any change to a table $T1$ also causes a Web page $p1$ to change, then an underlying data node representing $T1$ with a dependency to $p1$ should be created in $G$. If, on the other hand, only some changes to $T1$ affect $p1$, this approach might invalidate and/or update $p1$ more frequently than is needed. A method for specifying dependencies from a portion of $T1$ to $p1$ is needed. The generalized DUP algorithm provides a concise method for doing so which doesn't require a new node to be created for every element of $T1$ which affects $p1$.

  The generalized DUP algorithm also provides a concise method for an application program to notify a cache manager that part of a relational database table has changed without explicitly enumerating each element of the table which has changed.

When an application program notifies a cache manager that underlying data has changed, the cache manager identifies all objects which are affected by finding all nodes $N$ reachable from the nodes corresponding to the underlying data which has changed. $N$ can be found using graph traversal techniques similar to depth-first search or breadth-first search. The degree to which a version $o1_1$ of an object $o1$ is obsolete is determined from the sum of the weights of edges terminating in $o1$ from nodes $n2$ for which $o1_1$ is consistent with the latest version of $n2$. If this sum falls below a threshold value, $o1_1$ is highly obsolete and should be invalidated or replaced with a more recent version.

## 3.2 Data Maintained by the Generalized DUP Algorithm

Each vertex of G has an ID field representing it. For simplicity, we will assume that object nodes, underlying data nodes, and virtual graph object nodes share the same ID space. The cache manager stores information about each vertex in G in a *vertex information block (VIB)*. The cache manager also maintains a hash table which is indexed by vertex ID's and contains pointers to VIB's. That way, the cache manager can locate the VIB corresponding to a vertex ID in constant time.

A VIB for a node $n1$ has the following fields:

- *ID* : A string identifying the object, underlying data, or virtual object corresponding to $n1$.

- *update_num* : The number of updates the cache manager is aware of which have been performed on $n1$. The cache manager assigns version numbers to different version numbers of the same object. The version number for a particular version of an object is the *update_num* field at a time the version of the object was current.

- *timestamp* : Represents the time of the last change to $n1$ which the cache manager is aware of. While clocks can be used for determining timestamps, the preferred method for determining the current timestamp is the number of times an application program has notified the cache manager of changes to underlying data.

- *cache_list* : If $n1$ corresponds to an object, a list identifying all caches containing the object.
- *incoming_dep* : The incoming adjacency list for $n1$. Each element of *incoming_dep* corresponds to an edge terminating in $n1$ and contains two components:
  1. The $ID$ of the node which is the source of the edge.
  2. The weight of the edge.
- *outgoing_dep* : The outgoing adjacency list for $n1$. It consists of $ID$ fields for each node $n2$ for which a vertex from $n1$ terminating in $n2$ exists.
- *sum_weight* : The sum of the weights of all edges terminating in $n1$.
- *threshold_weight* : If $n1$ corresponds to an object $o1$, this quantity is used to determine if a version $o1_1$ of the object is highly obsolete. Version $o1_1$ is highly obsolete if the sum of the weights of edges terminating in $n1$ from nodes $n2$ such that $o1_1$ is current with respect to $n2$ falls below *threshold_weight*. An object is current with respect to a node in G if the object is current or if no updates have been made to the node since the object became noncurrent. Highly obsolete versions should be invalidated or replaced with a more recent version.
- *consistency_set* : If $n1$ corresponds to an object, a pointer to the *maximal consistency set* containing $n1$. A consistency set $C$ is a set of objects such that for any two objects $o1 \in C$ and $o2 \in C$, if versions of $o1$ and $o2$ are concurrently in the same cache, then the two versions must be consistent. A maximal consistency set is a consistency set which is not a proper subset of any consistency set.
- *latest_object* : If $n1$ corresponds to an object, a pointer to the latest cached version of the object which the cache manager is aware of. Caches use this pointer to obtain recent versions of objects from other caches. This avoids the overhead of having to calculate the objects from scratch.

The structure of $G$ is stored in the $ID$, *incoming_dep*, *outgoing_dep*, *sum_weight*, and *threshold_weight* fields in VIB's. Application programs specify the structure of $G$ via API functions which modify these fields.

Each cached version of an object has an *object information block (OIB)* associated with it. Each cache stores OIB's for all objects contained in the cache in a directory. Pointers to OIB's are maintained in a hash table indexed by object ID's. That way, the cache manager can locate an OIB corresponding to an object in a specific cache in constant time.

An OIB for an object $o1$ has the following fields:

- *ID* : A string identifying $o1$.
- *version_num* : Different versions of the same object have different *version_num* fields. The *version_num* field for a particular version of $o1$ is equal to the *update_num* field of the corresponding VIB for $o1$ at a time when the version of $o1$ was current.
- *timestamp* : represents the time at which the cached version of $o1$ became current.
- *actual_sum_weight* : the sum of the weights of all edges to $o1$ from a node $n2$ such that the cached version of $o1$ is consistent with the current version of $n2$.
- *dep_list* : The incoming adjacency list for $o1$. Each element of *dep_list* corresponds to an edge from a node in the graph $n2$ terminating in the node corresponding to $o1$ and contains the following components:
    1. The *ID* for $n2$.
    2. *weight_act* : A number representing how consistent the current version of $n2$ is with the cached version of $o1$. Our algorithm uses values of 0 (totally inconsistent) or the weight of the corresponding edge in the graph (totally consistent). A straightforward extension is to allow values in between these two extremes to represent degrees of inconsistency.
    3. *consistent_version_num* : the *update_num* field in the VIB for $n2$ at the time the cached version of $o1$ was current.

## 3.3 Adding Objects to Caches

When the current version of an object $o1$ is added to a cache $c1$, the cache manager ensures that an OIB for $o1$ exists in the directory for $c1$. This may require creating a new OIB for $o1$. The *version_num*, *timestamp*, and *actual_sum_weight* fields of the OIB are set to the *update_num*, *timestamp*, and *sum_weight* fields respectively of the corresponding VIB fields. The *dep_list* field in the OIB is copied from the *incoming_dep* field in the VIB. For each node $n2$ on the *dep_list* for $o1$, the *consistent_version_num* field is set to the *update_num* field in the VIB for $n2$. A pointer to $c1$ is added to the *cache_list* field of the VIB for $o1$ if $o1$ was not previously contained in $c1$.

Noncurrent versions of objects may be copied from one cache to another. If so, the OIB for the object is also copied to the new cache.

## 3.4 Propagating Changes to Underlying Data

Whenever underlying data changes, the application program informs the cache manager of the changes via an API function call. Let *changed_node_list* be a list of all nodes in $G$ corresponding to underlying data which has changed. The cache manager must traverse all edges reachable from a node in *changed_node_list* in order to correctly propagate changes to all cached objects.

The cache manager maintains a counter *num_updates* for the number of updates which it is aware of. This counter is incremented whenever the cache manager is informed of new updates to underlying data. In response to such an update, all nodes on *changed_node_list* are visited first. For each such node $n1$, the cache manager increments the *update_num* field in the VIB for $n1$ by 1. The *timestamp* field in the VIB is set to *num_updates*. This indicates that $n1$ has been visited during the current graph traversal. If $n1$ corresponds to an object, the *cache_list* field in the VIB is traversed in order to notify all caches containing the object that the object has changed. Each cache containing the object can then invalidate its version or obtain a more recent version of the object.

After all nodes on *changed_node_list* have been visited, the object manager must traverse all edges reachable from these nodes. It can do so using graph traversal techniques such as depth-first search or breadth-first search [1].

For each edge $(n1, n2)$ which is traversed, the cache manager determines if $n2$ has already been visited. This is true if and only if the *timestamp* field for $n2$ is equal to *num_updates*. If $n2$ has not been visited yet, its *update_num* field is incremented by 1, and its *timestamp* field is set to *num_updates*. If $n2$ corresponds to an object $o2$, all caches containing $o2$ must update the OIB for $o2$ and possibly update or invalidate their copies of $o2$. Such caches are located by traversing the *cache_list* field in the VIB for $n2$. For each such OIB, the *actual_sum_weight* field is decremented by the *weight_act* field in the OIB corresponding to the edge $(n1, n2)$. If this results in the *actual_sum_weight* field of the OIB being less than the *threshold_weight* field in the VIB, $o2$ is either invalidated from the cache or replaced with a more recent version.

If *actual_sum_weight* $>=$ *threshold_weight*, $o2$ is not replaced with a new version. Instead, the *weight_act* field in the OIB corresponding to the edge $(n1, n2)$ is set to 0.

If, on the other hand, $n2$ has already been visited, a slightly different procedure is followed. If $n2$ corresponds to an object $o2$, all caches containing $o2$ must update the OIB for $o2$ and possibly update or invalidate their copies of $o2$. For each such cache, the cache manager determines if the cache contains a current version of o2 by comparing the *version_num* field in the OIB with the *update_num* field in the VIB. If the cached version of $o2$ is current, the *consistent_version_num* component in the *dep_list* element corresponding to the edge $(n1, n2)$ is set to the *update_num* field for $n1$.

If, on the other hand, the cached version of $o2$ is not current, the *actual_sum_weight* field is decremented by the *weight_act* field in the OIB corresponding to the edge $(n1, n2)$. If this results in the *actual_sum_weight* field of the OIB being less than the *threshold_weight* field in the VIB, $o2$ is either invalidated from the cache or replaced with a more recent version.

If *actual_sum_weight* $>=$ *threshold_weight*, $o2$ is not replaced with a new version. Instead, the *weight_act* field in the OIB corresponding to the edge $(n1, n2)$ is set to 0.

### 3.5 Comparing Two Versions of the Same Object

Our algorithm provides a similarity score for assessing how similar two versions of the same object $o1_1$ and $o1_2$ are. Let $n1$ be the node in G corresponding to $o1$. The similarity score is based on the sum of the weights of incoming dependencies to $n1$ from nodes $n2$ for which the same version of $n2$ is consistent with both $o1_1$ and $o1_2$. Let *common_weight* be this sum. The similarity score is then given by the formula:

$$SS = \frac{common\_weight}{sum\_weight}$$

where *sum_weight* is obtained from the VIB for $n2$. Similarity scores range from 0 (least similar) to 1 (most similar).

In order to calculate *common_weight*, the cache manager adds up the sum of the weights of all edges $(n2, n1)$ in G such that the *consistent_version_num* component corresponding to the edge are the same in the OIB's for both $o1_1$ and $o1_2$.

## 4 Deployment of DUP at a High-Volume Web Site

## References

[1] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] J. Challenger and A. Iyengar. Distributed Cache Manager and API. Technical Report RC 21004, IBM Research Division, Yorktown Heights, NY, October 1997.

[3] A. Iyengar and J. Challenger. Improving Web Server Performance by Caching Dynamic Data. In *Proceedings of the Usenix Symposium on Internet Technologies and Systems*, December 1997.