Self-Maintaining Web Pages - An Overview

M. Schrefl, E. Kapsammer¹, W. Retschitzegger¹, B. Pröll²

Advanced Computing Research Centre
University of South Australia
{ek | wr}@ifs.uni-linz.ac.at, bproell@faw.uni-linz.ac.at

Abstract

Data-intensive web-based information systems usually employ database systems to store the contents forming the basis for web page construction. Generating web pages on the fly, especially in peak times, can lead to severe performance problems. Thus, pre-generation of web pages has been suggested to be ready for prime time, allowing to reliably deliver several hundred pre-generated pages per second. Maintaining the consistency of these web pages with respect to changes within the database in an efficient way, however, represents a major challenge. This paper presents a novel approach for "self maintaining" web pages that is, different to previous approaches, characterized by a simple (and thus, easy to maintain) database-to-web page mapping and very low page re-generation costs. This is achieved by utilizing fragmentation techniques from distributed databases, by allocating parameterized fragments to web page classes (rather than individual fragments to single web pages), and using the Extensible Markup Language (XML) as an intermediate layer between the database and the final web pages.

1. Introduction

For huge web-based information systems, which are characterized by a large number of web pages and frequent accesses, the employment of database systems (DBS) to store their content turned out to be worthwhile [1], [4], [13]. One example reported in [14] is the Electronic Commerce (EC) framework for web-based tourism information systems TIScover with more than 400 database tables, 400.000 web pages, and millions of pageviews per month. In such huge systems, generating all pages on the fly (by accessing the database, retrieving the corresponding data, and generating a page only after it is requested by a client [1], [4], [6]) comes at the cost of a high performance penalty. Pre-generation of web pages has been proposed as an alternative to on the fly generation: Web pages are materialized with data from the database and stored in the file system. Once the database content changes, affected web pages are re-generated using update processes [14] or database triggers [16].

A straightforward approach for maintaining pre-generated web pages has been presented in [14] for TIScover. Every time an underlying database entity (usually a tuple of a relation) changes, a meta database linking database entities to pre-generated web pages is inspected, and all web pages presenting that entity are re-generated. With this behavior, this approach as well as previous approaches by other research teams [15], [16] are essentially triggered pull-based approaches. Only the "trigger" is pushed, data are pulled.

Practice has shown, however, that such a straightforward approach suffers from (1) high costs for maintaining the mapping knowledge between database entities and web pages, since this mapping is done at the level of individual entities and web pages - and (2) high costs of re-generation, since every time a database entity changes, each page representing that entity is regenerated from scratch, thereby also pulling unchanged database entities.

The self-maintaining web pages approach overcomes both of these problems. It simplifies the mapping between database entities and web pages by employing distributed database concepts and class level mappings between relation fragments and web pages (rather than instance level mappings). And it uses an incremental push based data delivery approach together with XML [19] technology to reduce page re-generation costs at the web server significantly. In particular, the main characteristics of this approach are:

¹ Research performed at UniSA during leave from IFS, Austria.
² Research performed at UniSA during leave from FAW, Austria.
The mapping of database entities to web sites is - for the purpose of conceptual design - considered as a distributed database design problem. Relations containing the content of the whole web site are split into fragments that are allocated to web pages, whereby a single fragment may be allocated to one or more pages (cf. Figure 1). Going beyond distributed database design, fragments can be parameterized and organized into fragment classes that are mapped to page classes.

Page classes are described by a content schema and possibly several page representation schemata. Using XML technology, the content of a page is expressed in XML [19] and its representations are generated by XSL, including a possible restructuring using XSLT [20]. A canonical, generic mapping is used to represent a fragment on a web page such that basically no relation-specific mapping knowledge must be maintained. How tuples of fragments are presented or nested on a web page is expressed by meta knowledge and a generic XSL stylesheet.

An insert, update, or modification of a database relation is pushed to the relation fragments (using database triggers [9]) and pushed to the web pages representing that tuple. As the affected XML elements can be inferred from the generic mapping, the DOM API [18] can be used to update just these XML elements without any need to re-generate whole pages or pulling other data.

As an alternative to using fragments to partition the database content for distribution to web pages, views could be used and, in fact, have been suggested for mapping databases to the web [2], [5], [7], [10], [15], [17]. In principle, a fragment can also be seen as a kind of view. The main difference is that, whereas a view groups data from one or more relations and optionally performs complex operations on their attributes, a fragment simply is a partition of a single relation fulfilling a specific criterion. Although views are more expressive, fragments provide a clearer and more intuitive structuring of the base data, thereby easing the transformation of queries and updates specified on relations to corresponding operations on affected fragments. In many cases web pages primarily present either basic data from the database without further computation or aggregate data of a fragment. Thus, it is a very reasonable design decision to handle these cases in a clear and intuitive way using fragmentation and push-based data delivery and to handle the remaining few web pages of a web site by on-the-fly generation or a triggered pull-based approach.

### 2.1 Simple Fragmentation

To decompose relations into fragments, three different kinds of fragmentation, namely horizontal fragmentation, comprising primary and derived fragmentation, and vertical fragmentation as well as combinations thereof, called mixed or hybrid fragmentation, have been proposed [3], [12]. In the following, we will focus on horizontal fragmentation only. Primary fragmentation builds subsets of the tuples (rows) of a relation or fragment, called the fragmentation base, by applying a selection predicate. Derived fragmentation takes into account that in some cases the fragmentation of a relation cannot be based on a property of its own attributes, but is derived from another relation, called the derivation base. Notice that in our context the three correctness rules for fragmentation in DDBS [3], namely completeness, reconstruction, and disjointness, do not apply. Completeness and reconstruction are irrelevant since a physical image comprising the entire relation is always kept in the
database, and disjointness has already been disputed by Meghini [11].

Consider the fragmentation schema shown in Figure 2. Accommodations are fragmented according to two criteria, the town and the category. For each town, one primary fragment is defined for luxury accommodations (categories 3*, 4*, and 5*) and one for economy accommodations (categories 1*, 2*, and 3*).

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>email</th>
<th>kind</th>
<th>category</th>
<th>town</th>
<th>activities</th>
<th>Room rates</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>Biedermeier</td>
<td>b&amp;b</td>
<td>5*</td>
<td>Vienna</td>
<td></td>
<td></td>
<td>Luxury</td>
<td></td>
</tr>
<tr>
<td>02</td>
<td>Berghof</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>03</td>
<td>Hotel zum Post</td>
<td>hotel</td>
<td>3*</td>
<td>Vienna</td>
<td></td>
<td></td>
<td>Economy</td>
<td></td>
</tr>
<tr>
<td>04</td>
<td>Theater Hotel</td>
<td>hotel</td>
<td>4*</td>
<td>Salzburg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>05</td>
<td>Hotel Mozart</td>
<td>hotel</td>
<td>3*</td>
<td>Salzburg</td>
<td></td>
<td></td>
<td>Economy</td>
<td></td>
</tr>
<tr>
<td>06</td>
<td>Gasthof Post</td>
<td>hotel</td>
<td>2*</td>
<td>Salzburg</td>
<td></td>
<td></td>
<td>Economy</td>
<td></td>
</tr>
<tr>
<td>07</td>
<td>Hotel zum Hirschen</td>
<td>hotel</td>
<td>4*</td>
<td>Innsbruck</td>
<td></td>
<td></td>
<td>Luxury</td>
<td></td>
</tr>
<tr>
<td>08</td>
<td>Pension Alpinrose</td>
<td>hotel</td>
<td>2*</td>
<td>Innsbruck</td>
<td></td>
<td></td>
<td>Economy</td>
<td></td>
</tr>
<tr>
<td>09</td>
<td>Berghof</td>
<td>b&amp;b</td>
<td>1*</td>
<td>Vienna</td>
<td></td>
<td></td>
<td>Economy</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2. Primary and Derived Fragments

Room rates are fragmented according to the accommodation to which they belong into derived fragments, using the fragments of relation Accommodation as derivation bases. Thus, there exists a corresponding fragment of relation RoomRates (e.g., LuxuryRates_Vienna) for each fragment of Accommodation (e.g., LuxuryAcc_Vienna).

2.2 Parameterized Fragmentation

The simple fragmentation approach from distributed databases is rather inflexible and not fully adequate for its use in the web setting. If relations are split into logical fragments for presentation on web pages, the number of fragments will typically be large with each fragment being relatively small. Problems arise if the addition of a tuple gives rise to new fragments or if a relation should be partitioned alternatively according to some new criterion. For example, if a new town together with its accommodations is inserted into the database, a new primary fragment of Accommodation and a derived fragment of RoomRates, as well as the fragmentation knowledge for mapping these fragments to web pages must be specified. Even worse, if for each town a web page with all motels of that town should be created to cater for motorists, in the case of the 2,000 towns stored by TIScover, 2,000 fragments need to be specified manually.

This inflexibility can be addressed by introducing parameterized fragments. Fragments of the same kind are collected into fragment classes. A fragment class $F \langle L \rangle(X)$ with fragment attributes $X=X_1 \ldots X_n$ and fragment parameters $L=L_1 \ldots L_m$ collects fragments with attributes $X$, where one fragment $f \langle l \rangle$ exists for each value $l$ in the parameter domain, $\text{dom}(L)=\text{dom}(L_1) \times \text{dom}(L_2) \ldots \times (L_m)$. The parameter domain of each fragment parameter $P \in L$, $\text{dom}(P)$, is linked to the set of the values of the primary key $A$ of a reference relation $r$, i.e., $\text{dom}(P)=\pi_A(r)$. (Note that for simplicity of presentation, we assume that $P$ and $A$ are not compound; an extension to compound keys is straightforward.) If a new value is added to the parameter domain (by inserting a new tuple to its reference relation), new fragments with that parameter value are implicitly created as well. Thus, the fragmentation schema becomes independent from changes in the parameter domain.

Considering our example given in Figure 2, the fragments holding luxury accommodations in different towns are collected into fragment class LuxuryAcc<town> and the fragments on economy accommodations into fragment class EconomyAcc<town> as shown in Figure 3.

Figure 3. Primary and Derived Fragment Classes

If a new town "Graz" is added to the parameter domain town, two new fragments, LuxuryAcc<Graz> and EconomyAcc<Graz> are added as well. Notice that the reference relation containing all towns is not shown. Correspondingly, derived fragments of relation RoomRates are collected into fragment classes LuxuryRates<town> and EconomyRates<town>. Notice that these fragments have with "town" a parameter which is not an attribute of relation RoomRates. A derived fragment class "inherits" the parameters of the derivation base, in our example LuxuryAcc<town> and EconomyAcc<town>, respectively.
Non-parameterized fragments and relations can be considered special cases of parameterized fragments. A non-parameterized fragment is defined by a fragment class with no parameter and which possesses this very fragment as single instance. For convenience, we assume that for each relation \( r(X) \), a fragment class \( R<>(X) \) that comprises relation \( r \) as sole fragment exists. Parameterized fragment classes can be defined recursively upon previously defined fragment classes. They are specified as follows:

A primary fragment class \( G<K>(X) \) where \( X=X_1\ldots X_n \) \((n>1)\) are fragment attributes and where \( L \) and \( K=K_1\ldots K_m \) \((m \geq 0, K \subseteq X)\) are fragment parameters, is specified by a base fragment class \( F<L>(X) \) \(|L|\geq 20\) and a fragmentation predicate \( p \) over \( X \).

The fragment class \( G<K>(X) \) then comprises a fragment \( g(lk) \) for each \( lk \in \text{dom}(LK) \) and \( g(lk) = \{ t(X) \mid \exists f<l> \in F<L>; t \in f<l> \land t[K]=k \land p(t) \}\).

A derived fragment class \( G<LK>(X) \) with fragment attributes \( x \) and fragment parameters \( LK \) is specified by a base fragment class \( F<L>(X) \) and a derivation fragment class \( H<K>(Y) \) with at least one join attribute, i.e., \( X \cap Y \neq \{\} \). Note that for simplicity only, we assume that fragments are joined using the natural join, an extension to theta joins is straightforward.

The fragment class \( G<LK>(X) \) then comprises a fragment \( g(lk) \) for each \( lk \in \text{dom}(LK) \) and \( g(lk) = \{ t(X) \mid \exists f<l> \in F<L>; t \in f<l> \land \exists h<k> \in H<K>, \exists s \in h<k>: t \in f<l> \land t[K]=k \land p(t) \}\).

Parameterization of fragments provides significant advantages. First, the criterion for partitioning the global relation does no longer have to be specified for each fragment, thus reducing the number of fragment specifications. Second, insertions (or deletions) over a reference relation of a fragmentation parameter leads, without user intervention, to the generation (or deletion) of fragments parameterized with the inserted (or deleted) key values.

3. Page Schema

Like similar objects in object-oriented design are collected into object classes, pages of the same kind are collected into page classes. A page class is specified by a content schema (defining what kind of data a page contains and how these data are linked) and one or several presentation schemata (defining alternative ways of organizing and displaying a page's data). Page classes comprising more than one page are defined with one or several parameters, where the parameter values uniquely identify a page of the page class.

3.1 Page Content Schema

Bringing database content to the web, fragments are allocated to web pages. Instead of defining this allocation for individual fragments and individual pages, fragment classes are mapped to one or several page classes. The content schema of a page class defines among others, which fragment classes are mapped to this page class. A distinguished fragment class, called the foundation fragment class, must possess the same parameters as the page class; the remaining fragment classes must have the same or a subset of these parameters. A page class \( P<l> \) with foundation fragment class \( P<l> \) comprises exactly one page \( p<l> \) for each fragment \( f<l> \) of \( P<l> \). A fragment \( g<k> \) of a fragment class \( G<K> \) that is mapped to a page class \( P<l> \) \((K \subseteq L)\) is allocated to each page \( p<l> \) of \( P<l> \) where \( L[K]=k \).

For example, the left part of Figure 4 shows the page content schema of page classes \( P_LuxuryAcc<town> \) and \( P_RoomCategories<> \).
The common structure of these documents is defined by two generic XML DTDs, pageContentSchema.dtd and pageContent.dtd, that are independent of a particular page class. The benefits of this generic solution are threefold:

(1) There is no need to have a distinct DTD for each page class. Specific information about a certain page class is stored by the XML document describing the content schema of that class.

(2) Modifications of the relational schema, the fragmentation schema, or the page schema do not require to change a DTD, but only require to modify XML documents.

(3) The generic DTDs enable the definition of a generic XSL stylesheet for generating default page layouts.

For the following description of the generic DTDs and the corresponding XML documents, the reader's familiarity with XML at the level of [8] is assumed. The reader not interested in these details may proceed with Section 4.

The DTD pageContentSchema.dtd depicted in Figure 5 describes the page content schema. It consists of three basic element types, fragmentClass, internalPageRef, and externalPageRef.

Figure 5. Generic DTD for Page Content Schema

|<?xml version = '1.0'?>
|<!-- file: pageContentSchema.dtd -->
|<!ELEMENT pageContentSchema (fragmentClass*, internalPageRef*, externalPageRef*)>
|<!ATTLIST pageContentSchema pageClass CDATA #REQUIRED> |

Figure 6 depicts an example of an XML document that conforms to the DTD of Figure 5 and represents the content schema of page class P_LuxuryAcc<town>.

Figure 6. Page Content Schema for P_LuxuryAcc<town>

The DTD pageContent.dtd depicted in Figure 7 describes the content of a page (pageContent) and refers to the DTD describing the page content schema (pageContentSchema) by an external XML entity reference [19]. The page content consists of a set of fragments which are mapped basically one-to-one into page elements. Each fragment (fragment) consists of a set of tuples (tuple), which again consists of a set of attributes (attribute).

Figure 7. Generic DTD for Page Content

|<?xml version = '1.0'?>
|<!-- file: pageContent.dtd -->
|<!ENTITY % pageContentSchema SYSTEM "pageContentSchema.dtd"> |

Figure 8 shows an example of an XML document conforming to the DTD of Figure 7, page P_LuxuryAcc<Vienna> of page class P_LuxuryAcc<town>. The XML document serves as a container for the allocated fragments LuxuryAcc<Vienna> and LuxuryRates<Vienna>. In addition, it refers to the meta

---

3 Note, that for presentation convenience we use the symbols “<” and “>” to express parameterization, although not allowed within XML attribute values and file names.

---

References within and between individual pages during the formatting process (cf. Section 3.2).

The XML document serves as a container for the allocated fragments LuxuryAcc<Vienna> and LuxuryRates<Vienna>. In addition, it refers to the meta
information about its page class again by means of an external XML entity reference.

```xml
<?xml version = '1.0'?>
<!DOCTYPE page SYSTEM "pageContent.dtd" [ !ENTITY P_LuxuryAcc<Vienna>contentSchema SYSTEM "p_LuxuryAcc<Vienna>_contentSchema.xml" ]>
<page>
    &P_LuxuryAcc<Vienna>contentSchema;
    <pageContent>
        <fragment id = "LuxuryAcc<Vienna>">
            <tuple id = "LuxuryAcc<Vienna>:01">
                <attribute name = "id">01</attribute>
                <attribute name = "name">Biedermeier</attribute>
            </tuple>
        </fragment>
        <fragment id = "LuxuryRates<Vienna>"/>
            <tuple id = "LuxuryRates<Vienna>:01.Single">...
                <attribute name = "accomID">01</attribute>
                ...
            </tuple>
        </fragment>
    </pageContent>
</page>
```

Figure 8. Page Content for p_LuxuryAcc<Vienna>

3.2 Page Presentation Schema

The page content schema described in the previous section does not define any presentational issues such as nesting of fragments or visual properties. This is covered by the page presentation schema. For example, the page presentation schema of page class p_LuxuryAcc<town> expresses that for each accommodation the applicable room rates should be presented by a nested list (cf. the right part of Figure 4).

Using XML, the nesting structure of a page presentation schema is represented by an XML document, which is read together with a "flat" page by an XSL stylesheet to restructure the page accordingly.

4. Incremental Push-Based Data Delivery

The initial generation of fragments and pages from the specification of fragment classes and page classes is easy. We do not discuss it further here. Previous approaches for maintaining pre-generated web pages [14], [15], [16] used a triggered pull-based approach in which changes to database relations triggered the re-generation of affected web pages. Only the trigger is pushed, data are pulled.

The self-maintaining web pages-approach pushes modifications incrementally from their origin within the database by means of database triggers [9] and the DOM API [18] to affected web pages. In this section, we describe how changes to a relation are translated into corresponding changes of fragments and web pages.

4.1 Determining Modifications on Fragments

For the determination of affected fragments and their necessary modifications we build on algorithms proposed in the area of DDBS. Meghini and Thanos [11] presented algorithms for propagating relation updates (tuple insertion, tuple deletion, and tuple modification) to fragments. They used overlapping fragments, as we do, but they restricted themselves to primary fragmentation and did not cover derived fragments, recursive fragmentation, or parameterization. Due to lack of space and since tuple modification is the most complex operation, we discuss only this operation here.

The algorithm for the translation of a modification of a tuple \( t \) to \( t' \) of fragment \( f<i> \) (which corresponds to a relation \( r \) when originally invoked, but which is a "true" fragment when some relation update has been propagated to it) to a set \( U \) of equivalent operations on fragments of \( f<i> \) is given in Figure 9. The algorithm assumes that the primary key and the fragmentation parameters do not change: A change of a primary key is handled by a tuple deletion and a tuple insertion. A change of a fragment parameter of a tuple causes the tuple to be moved to a different parameterized fragment, which is reflected by deletion and insertion.

The algorithm comprises five steps, whereby step (1) initializes the output set \( U \) and step (5) returns \( U \). The remaining steps are responsible for determining the equivalent operations on fragments depending on \( f<i> \). These are in step (2) fragments of primary fragment classes having the fragment class of \( f<i> \), \( F<L> \), as fragmentation base; in step (3), fragments of derived fragment classes having fragment class \( F<L> \) as fragmentation base; and in step (4) fragments of derived fragment classes having \( F<L> \) as derivation base.

Concerning step (2), notice that tuple \( t \) and tuple \( t' \) may each belong only to one fragment \( g \) (\( g<\ell'[LK]> \) and \( g<\ell'[LK]> \), respectively) of a primary fragment class \( G<L> \) of \( F<L> \). If the modification does not affect the fragment parameter \( \ell \) (2.1), the algorithm corresponds to the algorithm known from Meghini and Thanos [11]. Otherwise (2.2), \( t \) does no longer belong to the parameterized fragment \( g<\ell'\ell>LK> \) but to fragment \( g<\ell'\ell>LK> \), into which the modified tuple \( t' \) has to be inserted if it qualifies for it (i.e., the predicate \( p \) holds over \( t' \)) and if it is not already part of it.
Algorithm: Translating a relation/fragment modification

**Input:** The operation "modify(f<l>, t, t')" over fragment base F<l>

**Output:** The set U of the equivalent operations on fragments of f<l>.

**Precondition:** t ∈ f<l>, t[L] = t'[L], primary key unchanged

**Method:**
1. Set U to empty;
2. For each primary fragment class G<k>L> with fragmentation base F<l> and selection predicate p:
   2.0 Let in (2): k=t[K] and k'=t'[K].
   2.1 If k=k' then
      2.1.1 If p(t) ∧ p(t') then add to U "modify(g<k>L>, t, t')";
   2.2 If k≠k' then
      2.2.1 Add to U "delete(g<k>L>, t)";
      2.2.2 If p(t) ∧ ¬ p(t') then add to U "insert(g<k>L>, t', t)";
3. For each derived fragment class G<k>L> with fragmentation base F<k>L> and derivation base H<k>L> with join attributes J:
   3.1 If t[J]=t'[J] then for each g<k>L> ∈ G<k>L> s.t. t ∈ g<k>L> add to U "modify(g<k>L>, t, t')"
   3.2 If t[J]≠t'[J] then
      3.2.1 For each h<k>L> ∈ H<k>L> s.t. exists s ∈ h<k>L> with s[J]=t[J] and not exists s' ∈ h<k>L> with
      s'[J]=t'[J] add to U "delete(g<k>L>, t)";
      3.2.2 For each h<k>L> ∈ H<k>L> s.t. exists s ∈ h<k>L> with
      s[J]=t'[J] and t' ∉ g<k>L> add to U "insert(g<k>L>, t', t)"
4. For each derived fragment class G<k>L> with fragmentation base F<k>L> and derivation base F<k>L> with join attributes J:
   4.1 If t[J]=t'[J] then no operation;
   4.2 If t[J]≠t'[J] then
      4.2.1 If not exists v ∈ f<l>: v≠t and v[J]=t[J] then for each g<k>L> ∈ G<k>L>, each s ∈ g<k>L>
      s.t. s[J]=t[J] add to U "delete(g<k>L>, s)";
   4.2.2 If not exists v ∈ f<l>: v≠t and v[J]=t'[J] then for each h<k>L> ∈ H<k>L>, each s ∈ h<k>L>
      s.t. s[J]=t'[J] add to U "insert(g<k>L>, s)";
5. Output U.

**Figure 9. Translating a Relation/Fragment Modification**

Considering step (3), if the modification of a tuple in f<l> does not affect the join attributes J between derived fragment class G<k>L> and derivation base H<k>L> (3.1), the fragments of derivation base H<k>L> can be ignored. But different to step (2), the parameters of fragments of G<k>L> are not only drawn from t but also from some fragment h<k>L>, such that each fragment g<k>L> of G<k>L> with x[L]=t[L] must be considered for modification. Remember that t[L] does not change. Since the fragmentation parameter x is also independent of the values of t, tuple t' cannot qualify for other fragments than t does. Hence, insertions of t to other fragments or deletions of t from other fragments need not be considered in this case. But if the modification of t does affect the join attributes J (3.2), the modified tuple may well qualify for other fragments than t. Step (3.2.1) and (3.2.2) determine the fragments from which t has to be deleted and into which fragments the modified tuple t' has to be inserted by inspecting fragments of the derivation base.

Finally, concerning step 4, no operation is required over a derived fragment (4.1) if a tuple modification in the derived fragment's derivation base does not pertain the join attribute values. Otherwise (4.2), a tuple s in a derived fragment might no longer qualify for the derived fragment since its link to a joining tuple in the derivation base is lost when the join attributes of J change (4.2.1). The link is, however, not lost when the derivation base contains another tuple v with the same join values than t. Conversely, a tuple s might newly qualify for a derived fragment if a tuple in the derivation base of the derived fragment receives a new join value (4.2.2).

The propagation of tuple insertions and tuple deletions over relations to fragments is similar. But notice that different to tuple insertions and tuple deletions, these operations might also cause the creation or deletion of parameterized fragments, if the primary key of the updated relation is linked to the parameter domain of some parameter of a fragmentation class.

**4.2 Determining Modifications on XML Pages**

After having determined the necessary modifications on a certain fragment, these modifications are pushed to all pages representing that fragment using corresponding algorithms to determine the necessary insertPageTuple, deletePageTuple, and modifyPageTuple operations. For lack of space, only the algorithm for pushing a fragment insertion is explained below and shown in Figure 10.

Algorithm: Push fragment insertion

**Input:** The operation "insert(f<l>, t)", over fragment f<l>.

**Output:** The set U of push operations over pages

**Method:**
1. Set U to empty;
2. For each p<k>L> s.t. p<k>L> ∈ µ(F<k>L>):
   2.1 For each p<k>L> ∈ p<k>L> where ∃ k[L]=l:
      Add to U "insertPageTuple (p<k>L>, f<l>, t)";
3. Output U.

**Figure 10. Algorithm for Pushing Modifications**

The mapping of fragment classes to sets of page classes is represented by the function µ: FragmentClasses → Powerset(PageClasses), where for each p<k>L> in µ(F<k>L>): k ⊆ L (cf. Section 2.2). The pages that represent fragment f<l> are pages of page classes to which the fragment class of f<l> is mapped (step 2) and whose parameter values over L coincide with the parameters of
insertPageTuple can be easily implemented using the DOM API [18] to change the affected page's contents incrementally.

The creation and deletion of fragments of a foundation class of a page class leads consequently to the creation and deletion of pages of that page class.

5. Conclusion

In this paper, we introduced a novel approach for pre-generating web pages in data-intensive web-based information systems. The self-maintaining web pages approach is characterized by a simple and easy to maintain mapping between database content and web pages. It reduces the costs for pre-generating web pages significantly by employing an incremental approach for push-based data delivery based on fragmentation techniques from distributed database systems. These techniques have been adopted and extended for the web setting by introducing parameterized fragment classes and allocating them to parameterized web page classes, thereby minimizing the amount of meta knowledge which has to be maintained. To implement the approach, XML has been used as an intermediate layer between the database and the final web pages, separating content from presentation. Together with DOM, the need to re-generate the whole web page in case of changes to the content is eliminated. Finally, the load of the web server is reduced significantly: Dynamic page construction is avoided by pre-generation and incremental maintenance, and page formatting is shifted to the client through the use of XML/XSL.

6. References