

CC68K (XML Databases). Duration: 1h30.

2nd exam – 27th of October 2005 – Inside XML databases

All paper documents are authorised. Answer either in English or in Spanish. Each answer **must be justified**. All the questions numbered with integers are independent and can thus be answered in any order. Marks are approximative and may change.

1. Numbering schemes (/15)

1.1. A variant of the Dewey numbering scheme (/10)

As the Dewey numbering, Path Identifiers (PIDs) encode complete root paths as sequences of offsets representing relative positions among children. The difference is that instead of using UTF-8, the number of bits used to encode a given offset for a node B are stored in the DataGuide node corresponding to the parent node of B. In the following, we suppose that an element is represented by a tuple (Document-ID, PID, DataGuide node ID) and that XML documents are only composed of elements (no other node type). For all the questions related to the cost of an operation, we assume that a lookup in the DataGuide is much more faster than a lookup in the database.

a) Show that the numbering scheme is sufficient to reconstruct the whole XML structure of any document.

We only have to prove that the scheme is equivalent to Dewey, which is trivial since looking at the dataguide (from the root to the parent of the corresponding node in the dataguide) gives us the exact

b) Let `<a> <c></c> <d><d><e><f></d><c> <c><c><c></c>`; `<a><c>`; `<c><d>` be three XML documents stored in the database. Draw the extended DataGuide (consider that the DataGuide is a *tree* with the following constraint: all root elements have the same parent in this DataGuide) and the node representations of all the elements of these documents. For the sequence of offsets, use either a decimal or a binary notation. What is the minimum size for the representation of this database (without considering the storage of the DataGuide).

For the dataguide (between parenthesis are the number of bits needed for encoding, and curly braces are used for the dataguide structure),

```
Root (0) {
  a(2) { /* path /a elements can have up to 3 children => 2 bits */
    b(2) { /* /a/b elements can have up to 3 children => 2 etc. */
      c(0),
      d(1) { e(0), f(0) }
    },
    c(1) { c(0) }
  },
  b(1) { c(0), d(0) }
}
```

The numbering is similar to Dewey except that (1) the root element is encoded with 0 bit (2) the offsets start from 0 (for instance, `/a[1]/b[2]/d[1]` is encoded as (1,0) or in binary 0100).

The maximum size is given by the graph (looking at the maximum number of bits for a leaf): it is 5 (for

a – b – d – e for instance) + the maximum ID of a DataGuide node (10 => 4 bits); in total 9 bits are enough for this database. There can be 11 different PIDs (=> 4 bits). There are three documents (2 bits). The size of the database is thus (9 bits + 4 bits + 2 bits) * 18 node

c) What are the possible reconstructions that can be made from one element representation?

Ancestors, nth child, siblings, ancestor siblings just by looking at the DataGuide (to know the number of bits). Contrarily to Dewey, it is not possible to compute the representation of any element without knowing its sequence of node labels.

d) Let the different decisions be: parent, ancestor, child, nth child, following, following-sibling, preceding, preceding-sibling. What are all the decisions that can be made from two elements representations? For each decision that cannot be done, cite a numbering scheme that allows it (if it exists).

ancestor (and hence parent, child, descendant) can be made by looking at the dataguide: if the node is an ancestor (or a parent) in the DataGuide, then it is also a parent/ancestor in the XML document.

A next/previous sibling decision can be taken easily: given two representations r1 and r2, just look at the corresponding parent node DGP in the DataGuide. If it is not the same, then the two nodes are not siblings. If it is the same, take the bit number N associated to DGP and compare the last N bits of r1 and r2; if last(N,r1) > (resp. <) last(N,r2) then r1 is the next (resp. previous) sibling of r2

Following (a is following b if a is after in document order AND b is not an ancestor of a)/preceding decisions can be made using the same process as previously. Let dg1 and dg2 be the corresponding DG nodes of elements 1 and 2. Let A be the first common ancestor of dg1 and dg2. Let b be the cumulated number of bits of nodes from the root of the DG to A. If the first b bits of r1 gives a superior (resp. inferior) number to the first b bits of r2, then node 1 is following (resp. preceding) b2.

e) Propose an enhancement of the DataGuide so that the ancestor/descendant decision is faster.

using the pre-size scheme for the DataGuide allows to retrieve any descendant just by looking at the [pre,pre+size] interval.

f) What kind of queries can be partially answered by a simple consultation of the DataGuide – that is the queries for which we can pre-compute the PIDs so that all the elements in the database can be searched by a simple lookup (« SELECT ... FROM xmldatabase WHERE PID in (X)? »).

Any query that do not contain a predicate other than a positional predicate.

g) What are the properties of the XML database that may affect the storage size? Give an example of a database for which this numbering scheme is adapted. Give an example of a database for which it is not and propose one or more numbering schemes which are more well suited..

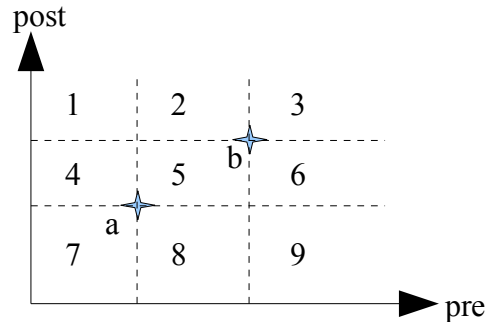
The schema is bad if the nodes that share the same simple path (implies same DataGuide node) have a number of children that varies a lot. In this case, one case use the pre-size scheme. Very deep documents are also not really suited for this scheme.

h) Propose an enhancement of this numbering scheme if the DTD is known.

If we know the DTD, we can avoid to store the offset of an element for which we know the position. For example, if we have <!ELEMENT a (b,c,d,e,f)> then it is not necessary to store the offset of b, c, d, e and f as it is fixed (0-4).

1.2. Pre/post and descendant axis (/5)

For an element x , $\text{pre}(x)$ and $\text{post}(x)$ denote respectively the pre-order and the post-order number of x .



a) With respect to elements a and b , characterise each of the 9 regions of the above graph (pre-post plane).

Recall that the space is partitioned into 9 regions for an element (which are ancestor, following, preceding, and descendants).

1 \Rightarrow common ancestors of a and b

2 (4) \Rightarrow ancestors of b (a) only and also following of a (preceding of b)

3 (7) \Rightarrow elements following (preceding) a and b

5 \Rightarrow nodes that are preceding b and following a

6 (8) \Rightarrow descendants of b (a) only

9 \Rightarrow empty (a and b cannot have a descendant in common)

b) We now want to evaluate the descendant axis of an XPath expression. In order to achieve this goal, we consider this sub-problem. Let $A=(a_1, \dots, a_n)$ and $B=(b_1, \dots, b_n)$. be two *sequences* of elements, both ordered in document order. We want to find **all** the descendants of A that are in B .

i. Without knowing anything on B , are there nodes that we do not have to consider in the set A ? If yes, characterise those nodes using the pre/post. If no, justify.

Yes: if a_i is an ancestor of a_j (that can only happen if $i < j$), then we can ignore a_j since all the descendants a_j of a_i are descendants of a_i .

ii. Without knowing anything on A , are there nodes that we do not have to consider in the set B ? If yes, characterise those nodes using the pre/post. If no, justify.

Without knowing A it is not possible since any element of B can be a descendant of A (except if b is the root of a document).

iii. For a given a_i what is the portion of the pre-post space that we need to explore if we want to find the descendants of a_i that are not a descendant of a_{i+1} ?

Let a be a_i and b be a_{i+1} . Suppose that a is not an ancestor of b (otherwise, we have the standard inequality on pre/post). This correspond to the region 8 of the graph. Formally,

x is a descendant of $a \iff \text{pre}(a) < \text{pre}(x)$ and $\text{post}(a) > \text{post}(x)$.

x is not a descendant of $b \iff \text{pre}(b) > \text{pre}(x)$ OR $\text{post}(b) < \text{post}(x)$.

Since b is not a descendant of a and that $\text{pre}(b) > \text{pre}(a)$ then $\text{post}(b) > \text{post}(a)$ from the previous inequality (with $x \rightarrow b$ and $b \rightarrow a$). As x is a descendant of a , we know that $\text{post}(a) > \text{post}(x)$ and hence that $\text{post}(b) < \text{post}(x)$ is impossible [we just eliminated the region 3].

Then we have the following region:

$\text{pre}(x)$ in $] \text{pre}(a), \text{pre}(b)]$

$\text{post}(x)$ in $[0, \text{post}(a) [$

iv. Suppose that for a given i and a given j , $\text{pre}(b_j) > \text{pre}(a_i)$ and $\text{post}(b_j) > \text{post}(a_i)$. Knowing a_{i+1} , can

you characterise the region of the pre-post plane that will not contain any descendant of A?

That corresponds exactly to the graph with a and b. We know that b is not a descendant of a; furthermore, if $\text{pre}(a_{i+1}) > \text{pre}(b_j)$ then we can skip all the b nodes until $\text{pre}(b_k) > \text{pre}(a_{j+1})$.

2. Locking and concurrency (/5)

A document model specification allow the following operations: from a node, one can access its n^{th} child or its parent.

a) Among the non-semantic locking mechanisms (...2PL), what are those that do not verify the serialisability property with this new document model?

Two possible answers:

I) Conceptually, if we apply the mechanisms (locking the nodes/pointer that are at least conceptually traversed), all schemes are OK

[preferred] II) Since the document models are different, it may happen for OO2PL. It is sufficient to give an example to prove it. For OO2PL, it is easy to see that if T1 reads the second child, and that T2 inserts a child before it then we have a problem because only the pointers "2nd child" (for T1) and first child (T1) are locked.

b) Propose an adapted locking mechanism.

Depends on a) if you took the option (I), then nothing to do

If you took option (II), then you can simply adapt OO2PL.

3. Normal forms (/5)

a) Let a DTD be

```
<!ELEMENT artist (name, influence*)><!ELEMENT name (#PCDATA)><!ELEMENT influence (#PCDATA)>
```

Find all the trivial functional dependencies for this DTD.

All of the form $p \rightarrow p'$ where p' is a prefix of p (and p does not end in $\#PCDATA$) or the form $p \rightarrow p/\#PCDATA$

and:

– $/\text{artist}/\text{influence} \rightarrow / \text{artist}/\text{name}$

– $/\text{artist} \rightarrow / \text{artist}/\text{name}$

+ of course all the FDs that are build from the previous one by adding a path in the left part.

b) Would it be possible to use functional dependencies with path of the form $//\text{element-node-test}$? If no, justify. If yes, propose an extension of FD for that kind of XPath.

The important is that a path = a value; given a tree tuple, an element with a given name can appear several times in the path from the root to the node. So no it is not possible in general (but it does depend on the DTD also).

c) Prove that functional dependencies used in XNF are a generalisation of functional dependencies of relational databases.

First, we have to do a mapping relational schema to database schema. This can be done simply by mapping each attribute to a given element name.

The DTD of such a mapping will be

```
<!ELEMENT (rows)>
```

```
<!ELEMENT row (attribute1, ...) >
```

```
<!ELEMENT (attribute1, attribute2, ...) >
```

<!ELEMENT attribute1 (#PCDATA)>

...

It is then easy to convert any relation instance into XML by using a “row” element for each tuple of the table.

A FD is then converted easily:

let $a_1 \dots a_n \rightarrow b$ be a FD, then with our DTD this becomes

$/\text{attributes}/\text{row}/a_1/\#PCDATA \dots /\text{attributes}/a_n/\#PCDATA \rightarrow /\text{attributes}/b/\#PCDATA$

In order to prove that this is a generalisation, we must show that if a functional dependency holds (does not hold) in the relational world, then it holds with the XML FDs.

Suppose that $X(x_1 \dots x_n) \rightarrow Y(y_1 \dots y_p)$ holds, then for any rows r_1 and r_2 , $r_1.X = r_2.X \Rightarrow r_1.Y = r_2.Y$.

Suppose that for any tree tuples t_1 and t_2 ,

$t_1./\text{attributes}/\text{row}/x_i/\#PCDATA = t_2./\text{attributes}/\text{row}/x_i/\#PCDATA$.

By definition of a tree tuple, $t_1./\text{attributes}/\text{row}$ (the same for t_2) corresponds to one and only one row element. As a row element corresponds to a row in the table, and that all the attribute values are equal for the set X , then the values are equal for the set Y .

The same when it does not hold by saying that we can find two rows (in the table) for which $r_1.Y$ is not equal to $r_2.Y$, and that those two rows can be found in the corresponding XML file.