Nr.: FIN-16-2008

A First View to the H2O Storage Structure – The Marriage of the TID-Concept with the XML-File Structure

Klaus Benecke

Arbeitsgruppe Theoretische Informatik

Technical Report

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Impressum (§ 10 MDStV):

Herausgeber:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Der Dekan

Verantwortlich für diese Ausgabe:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Klaus Benecke
Postfach 4120
39016 Magdeburg
E-Mail: benecke@iws.cs.uni-magdeburg.de

http://www.cs.uni-magdeburg.de/Preprints.html

Auflage: 120

Redaktionsschluss: 17.12.2008

Herstellung: Dezernat Allgemeine Angelegenheiten,
Sachgebiet Reproduktion

Bezug: Universitätsbibliothek/Hochschulschriften- und
Tauschstelle

A First View to the H2O Storage Structure – The Marriage of the TID-Concept with the XML-File Structure

Klaus Benecke

IWS/FIN, Otto-von-Guericke-Universität Magdeburg
Postfach 4120
39016 Magdeburg, Germany, Sachsen-Anhalt
benecke@iws.cs.uni-magdeburg.de

Abstract. This paper describes a universal storage structure, which is settled between relational and hierarchical storage. It aims to marry XML files with the TID concept. We support at first glance only nesting level of two. Therefore records are in general greater than in relational model (clustering of data with different types), but smaller than in a hierarchical model (XML). Besides arbitrary primary data also indexes can be stored. Value- and text-indexes can be expressed by H2O structures and parent- and path-indexes are not necessary, because the access to subtuples goes always through the head record (parent node). The H2O storage structure aims to be a base for a universal, simple, and lean information system with our query language ottoQL at top.

1 Introduction

XML documents allow an adequate representation of many kinds of information, tabular and non-tabular documents, but they have the disadvantage, that we cannot insert or delete a (sub-) record into the file. Such an update would require a shifting of all succeeding elements or the whole document has to be rewritten. If indexes on such an XML file exist then all indexes have to be recreated. H2O files are intended to replace XML files, if these files contain a very large number of records and if we need a direct access to (a part) of these records. H2O files are as similar as possible to XML files, but differ in following points:

1. An H2O file is a set with a key, consisting of zero or more elementary fields.
2. Beside lists in an H2O record may occur sets or bags as repeating groups
3. If an H2O record is larger than a page (several physical blocks) then the record is split into two or more sections and a mini-directory is generated in the first page of the record.
4. The elements of set- and bag- repeating groups are sorted by the first fields.
5. To allow more efficient processing a fixed field length will be allowed in later versions.

H2O files agree with XML-files in the following points:

1. An H2O table is stored in the actual version in exactly one file.
2. Metadata are contained in the file (in a metadata record).

3. An H2O file is transferable like a XML-file from one computer to another (in the internet) and easily editable by an ordinary editor (at least from experts).
4. In the actual version a record is up to the root tag an (small) XML document. Later tag data will be saved in some situations.

In the second section we define H2O tables in a narrower sense and how arbitrary XML documents can be mapped to H2O tables. The following section 3 contains the description of the H2O storage structure is in detail. In section 4 several kinds of indexes are presented. It becomes clear that H2O indexes can be used in many kinds of applications. We think that they can replace B-trees and inverted files. Some details of metadata are presented in the next short section 5. Possible extensions of the file concept are described in section 6. We demonstrate that the query optimization can be realized at top (ottoQL- (compare [5])) level only by two examples. Finally, we consider related work and a summary.

2 H2O Tables

Definition: An ***H2O table in the narrower sense*** is a table, whose proper hierarchical levels have maximal order (depth) of two and which contains in the first level a key, which consists only of elementary fields. Optional values (?-collections) are not considered as proper hierarchical levels such that they can occur in arbitrary depth.
Because H2O tables are more general than flat relations, it is easy to represent an arbitrary table by one or more H2O tables. We consider only one simple example:
D: M(A1, A2, M(B1, B2, M(C), M(D1, D2)), M(E1, E2)) # M abbreviates set
can be replaced by
H1.h2o: M(A1, A2, M(E1, E2)) and
H2.h2o: M(B1, B2, A1, M(C), M(D1, D2)),
If we assume that A1 is key in D and {B1, B2} contains a local key in D. Nevertheless the table D can be expressed by an ottoQL-view:

```
(aus H1.h2o              # aus: from
rename A1 by ANEU1
ext H2.h2o at A2         # ext: extension
mit B1:: ANEU1=A1)       # selection
```

If the table D has no keys, then they have to be generated by the system (OID's).
If we want to represent an n:m-relationship between entity set A and entity set B by H2O tables, then 2 H2O tables are needed:

```
DA.h2o: M(A1, A2,..., M(B1, C1, C2, ...)),
DB.h2o: M(B1, B2,...)
```

Here, C1, C2, … are the attributes of the relationship, and B1 is the key of entity set B. The table DA.h2o represents the "join" between A and the "relationship file".

3 The H2O Storage Structure

We represented each H2O table directly by one file to keep the first version sufficient simple. The storage structure will be introduced stepwise. For a complex record two possibilities exist:

1. The complex record is smaller than the page size (small record).
2. The complex record is larger than the page size (large complex record).

It is shown that there is no great restriction of generality, if we assume that the mini-directory of a complex record in case 2 is smaller than the page size. Independently of the size of a complex record the address of a complex record consists of a page address (in case 2 the address of the first page) and a slot number, by which the position of the complex record in the page is expressed (in case 2 the slot number is always one, if the record has not been moved). A complex record cannot be inserted in an arbitrary page, such that we cannot keep the H2O file sorted. The sorting and identification of the complex records of the H2O file will be realized by an additional primary index, which is also an H2O file, shown later.

If a small record grows that there is not enough free space in the page then the record is written in a new page with enough free space. Updating indexes would require too much time, therefore the address of the record is not changed and the TID of the new address is written on the old position of the record. In this case an additional access to the record is required. If the record has to change its position once more, then it will be “deleted” at its real position and its address on its first position will be overwritten, such that we need also only two accesses to get the record.

If the small record grows further and becomes greater than the page size, then the first section of the record is written in an empty page, which is then considered as a so called B-page (page of status B). A B-page contains beside the first section of the record also a page foot with information of each further section of the complex record that we have a direct access to each subtuple of the large complex record, if the key respectively sequence number of the subtuple is given. The remaining sections of a large record are contained in A-pages. A page, which contains several small records or sections of large records, is called A-page. Sections of large records are addressed in the same way as small records - by TID’s. Each complex record no matter whether it is smaller or larger than a page is stored in the following order:

1. an offset for the beginning of the second, third,.., last repeating group
2. first subtuple of the first repeating group
3. …
4. last subtuple of the first repeating group
5. first subtuple of the second repeating group
6. …
7. last subtuple of the second repeating group
8. …
9. last subtuple of the last repeating group

As we shall see later, we will consider the head record of a complex record also as repeating group subtuple(s). The subtuples of each proper repeating group will be sorted by the key (set, bag) or stored in the input order (list). The data mentioned in the first line are needed for a quick access to subtuples and later for a saving of tag

data. The offset is calculated from the beginning of the complex record or the beginning of the corresponding section in the case of a large record.

A record, which starts in a B-page, is in a stable phase. It can grow further without changing its starting position. Each further subtuple will be put on the position, which corresponds to the ordering and succeeding subtuples of the corresponding section will be moved forward. If we insert in a page without enough storage space then the section can be stored in a page with enough free space or it can be divided or it can be merged with the preceding or succeeding section.

Besides, these "primary data", which length is varying, the pages contain also management data of varying length. The primary data are located at the beginning of the page and the management data at the end (page foot) to hold shifting of data in pages low. In detail the entries have the following structure:

TID = page number + position number (slot number)
page foot = constant part + variable part
constant part = pages status (A, B, M, or F)
+ first free byte in page (FreeOffSet1)
+ last free byte in page (FreeOffSetLast)
+ page number of the next page with the same free space degree
+ page number of the preceding page with the same free space degree
M: starting page for a large meta record
F: completely empty page
variable part = TID-entries (status A)
mini-directory (status B and M (see above))
no entries (status F)
TID-entry = RecordKind ('K': complete small record
'k': complete relocated small record
'G': section of a large record
'T': TID to the real position of the record
'R': rudiment
'M': meta data record
'm': complete relocated small metadata record)
+ OffSet (begin of the record (section) in the page)
mini-directory: RGno: Number of the repeating group of the first subtuple of the section
+ key (set, bag), sequence number (list) of the first subtuple
+ TID of the section of the large record

Now we can define for the page foot schemes:

Foot of an A-page:
PageStatus, FreeOffSet1, FreeOffsetLast, FreePageSucc, FreePagePred, L(RecordKind,OffSet)

Foot of a B- or M-page:
PageStatus, FreeOffSet1, FreeOffsetLast, FreePageSucc, FreePagePred, L(RGnr, L(KeyValues), SectionTID)

For illustration purposes we consider in figure 1 an example of an H2O table in narrower sense. This table consists of one small and two large records. The first

record is cut between the subtuple d8 and d9 and the third record between d20 and d21 and d60 and d61. The "M" in the head of the tabment stands for set.

<< M(	A,	B,	C,	M(	D,	E), M(	F,	G))::
	a1	b1	c1		d1	e1	f1	g1
					d2	e2	f2	g2
					…			
					d8	e8		
					d9	e9		
	a2	b2	c2		d10	e10	f3	g3
							f4	g4
	a3	b3	c3		d11	e11	f5	g5
							f6	g6
					…		f7	g7
					d20	e20		
					d21	e21		
					…			
					d60	e60		
					d61	e61		
					…			
					d70	e70		>>

Fig. 1. An H2O table in narrower sense with cuts in tabment representation

Figure 2 shows the storage structure of the H2O table of figure 1 of scheme M(A,B,C,M(D,E),M(F,G)). Here is D local key in M(D, E). The double linked free storage chains and the metadata are not included, because of reasons of clarity. In figure 2 it is visible that the page feet are stored in a reverse order compared to the above scheme.

Because we store a record as a small XML-file, we can allow more general schemes in our H2O storage concept. This can be illustrated by three examples:

M(A1, A2, M(B1, B2, B3), C1?, C2, M(D1, M(D2, D3)), E1, (E2? | B(F1, F2?)), L(G1, G2, L(G3 | G4)))

Here, we have 8 "repeating groups" with the following number of keys:

1: A1, A2: 0 (Because we have only one A-subtuple)
2: M(B1, B2, B3): 2 ((B1,B2) is assumed to compose the local key)
3: C1?, C2: 0 (because C-values occur at most once)
4: M(D1, M(D2, D3)): 1 (D1 is assumed to compose the key of the outer M)
5: E1: 0
6: E2?: 0
7: B(F1, F2?): 1 (for bags we set the number of "keys" by definition to one (F1))
8: L(G1, G2, L(G3 | G4)): 1 (The sequence number of the outer list)

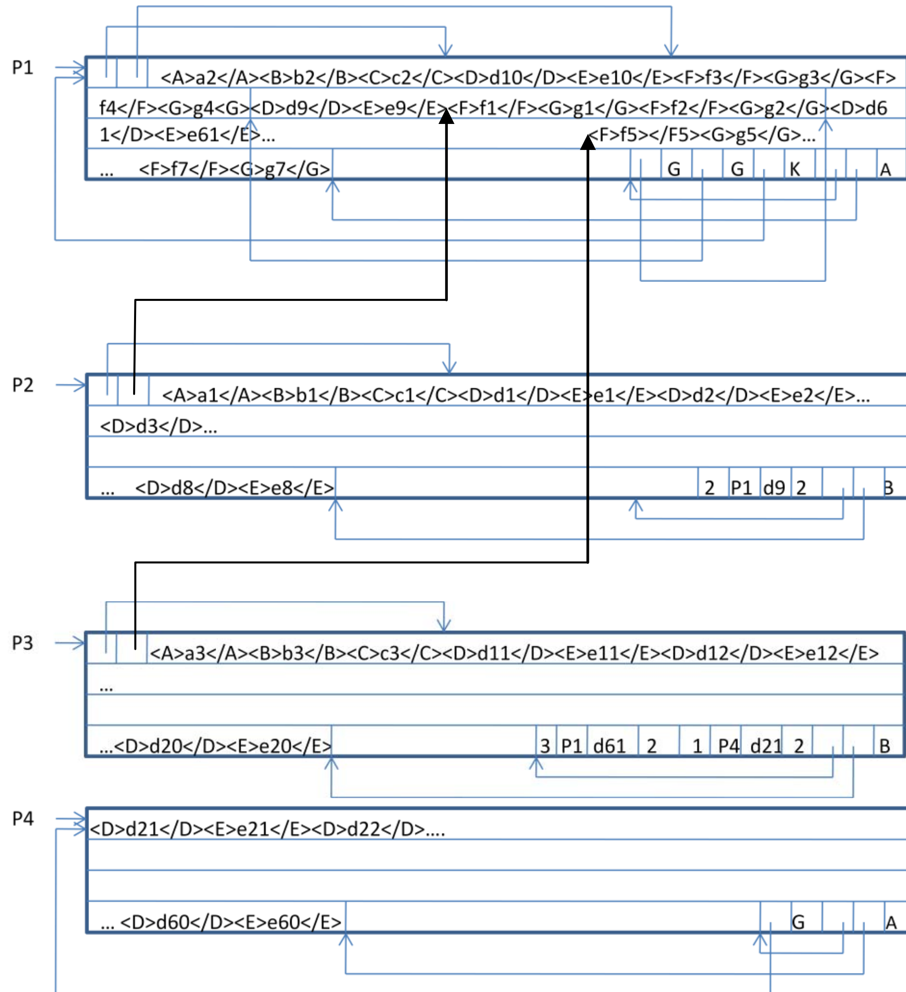

Fig. 2. H2O storage structure of the table of figure 1 without metadata

An H2O table of the type
    M(M(A, B))
contains only two “repeating groups”:
    1: “”: 0 (it exists no field at first level)
    2: M(A, B): 1 (A is assumed to be the key)
Let us consider the third, a recursive example:
    DOC: M(CHAPTER)
    CHAPTER: CTITLE, L(SECTION)
    SECTION: STITLE, L(TEXT | P | IMAGE | SECTION)
    P: TEXT
    …

By replacement of corresponding tags we get the "H2O"-scheme of this document
DOC': M(CTITLE, L(SECTION))
Here, CTITLE is the key of the document and we have only 2 "repeating groups":
1. CTITLE: 0 (key consists of zero fields)
2. L(SECTION): 1 (key is the sequence number of the list)
In general we replace tags as long as possible by its schemes, except that we do not replace names of elementary types as TEXT,… and further we do not replace names in proper inner repeating groups of the modified scheme. That means for the last example that we would have to replace in the scheme of DOC' CTITLE if it would not be of TEXT type, and we do not have to replace SECTION, because it is already in a proper repeating group.

We want simply store page numbers and slot numbers by variable length sequences of digits. Therefore small numbers require a little storage area and the number of pages and the size of the pages are unrestricted from the point of view of our storage concept. An offset-value is always smaller than the page size. Therefore we will use for the offsets as many digits as the page size requires (e.g. 4 digits for page size 4096). Because of this some functions are more easily to implement and a direct access to a corresponding TID-entry of an A-page is possible. The repeating group L(RecordKind, OffSet) from a foot of an A-page can written in the following way:
K001 K200 G400 K430 … (forward notation)
This means that the first (short) record has size 199 byte, the second 200 byte, the section of a large record a length of 30 byte, etc..

A repeating group L(RGno, L(Keyvalues), SectionTID)) from a foot of a B-page of the above type with eight repeating groups looks as follows:
2 b1 b2 23 2 | 5 5666 6 | 8 301 4666 8 | etc.
The symbol "|" is not a part of the repeating group; it is the delimiter of a mini-directory entry. Here, it is assumed that the first section is in the B-page; the second section starts with a (B1, B2, B3)-subtuple with key (b1, b2) in page 23 as second entry. The third section is in page 5666 on position 6 and starts with the fifth repeating group, which has zero keys fields. The forth section is in page 4666 at eighth position and starts with the 301th subtuple of eighth repeating group (list). Such sequences of character can be interpreted, because we know, how many key fields each repeating group contains. If an H2O table contains only one proper repeating group then the repeating group number can be omitted in later versions.

The insertion and deletion of a subtuple into large records should be implemented in a way such that the sections (the last and first excluded) are larger than 30 (or 40?) percent of the page size. Because of this, the mini-directory will not be too large and estimates for the page size for a given maximal record size can be computed.

4 Indexes

Let's start with two simple tables:
STUDENT.h2o: M(STID, NAME, FAC, LOC, M(COURSE, MARK))
FACUL.h2o: M(FAC, DEAN, STUDCOUNT)

As a first step we could define a primary index for STUDENT.h2o in the following way:

I1.h2o: M(STID, STUDENT_TID)

The tuples in this H2O index are not sorted. Therefore most people would not consider such a file as an index. But because this index is much smaller than the original student file corresponding queries could be realized more efficient, than without this index. On the other hand, it is in general necessary to scan the half index, if we look for an address (TID) of a student, if his identifier (STID) is given. We do not have direct access. Therefore it is much better to replace I1.h2o by a file

I2.h2o: M(M(STID, STUDENT_TID))

which contains the same set of pairs, but one level deeper. I2.h2o can contain only one complex record, because the outermost set of an H2O table is forced to have a key. This key has to consist of zero fields, which can have only one value. Therefore I2.h2o contains exactly one complex record. If now the repeating group is larger than a page then we have direct access to the section, which contains the desired student address. This is due to the fact that the pairs in I2.h2o are sorted by STID contrary to I1.h2o. In the same way we can define a FAC-Index for the FACUL-file. If we want to create a FAC-index for STUDENT.h2o, we could define at first an index of the following type

I3.h2o: M(FAC, M(STUDENT_TID))

which contains each FAC-value only once and additionally a primary index for this index I3.h2o:

I4.h2o: M(M(FAC, I3_TID))

Now, we see that I3 and FACUL.h2o have a similar structure. So we could extend either FACUL by the corresponding address repeating group or I3 by DEAN and STUDCOUNT:

FACULTY.h2o: M(FAC, DEAN, STUDCOUNT, M(STUDENT_TID))

Now we need only one FAC-index for both files:

I5.h2o: M(M(FAC, FACULTY_TID))

Here, the FAC-Index for STUDENT consists of two files. In the same way it is possible to create a two stage STID- index for STUDENT.h2o:

I6.h2o: M(STID1, M(STID2, STUDENT_TID))

I7.h2o: M(M(STID1, I6_TID))

Here STID is divided in STID1 and STID2 (STID=STID1^STID2)(^: concatenation). It is clear that I6.h2o (and I7.h2o) can be expressed by an ottoQL-program:

```
aus  STUDENT2.h2o                   # aus: from
ext  STID1:= STID subtext (1, 4)    # ext: extension by a new column
ext  STID2:= STID subtext (5, 4)    # STID is assumed to consist of 8 byte
gib  M(STID1, M(STID2, STUDENT_TID)) #  gib: similar to restrict from [1]
```

Here, STUDENT2.h2o, is the same as STUDENT.h2o, but extended by the TID:

STUDENT2.h2o: M(STUDENT_TID, STID, NAME, FAC, LOC,
M(COURSE, MARK))

The repeating groups in I6.h2o and I7.h2o are much smaller than in I2.h2o. Therefore a two or even three steps index can be used for much greater files than a single index. In general the question, whether an index has to be realized by one or two H2O files can be answered by a rough estimation, if the index data fit in one complex record or not. If not, the key has to be divided into two or more substrings. If it is possible then

this division should be made by a point of view of content (EMP_ID=BIRTHDATE ^ REST). Often the key consists of two or more fields. Then this division of the whole key into fields can be taken for the division of the indexes. The division of an index to two or more files has also an additional advantage. Assume we have 10000 students. Then it is a good approximation to have 100 STID1-values. Therefore, I6.h2o along with I7.h2o contains only 200 STID1 values. But in I2.h2o we have 10000 STD1-values. This is a considerable saving of storage space, compared to I2 and with B-trees.

Up to now we considered only indexes of the top level of an H2O file. For fields of repeating groups we can also create indexes. An index for COURSE could be integrated into a file COUR.h2o: M(COURSE, PROF) because it can be presupposed that a primary index for COURSE exists:

COURSE.h2o: M(COURSE, PROF, M(STUDENT_TID))

If we want to find all results of examinations for a given course then we can access the given primary index of COURSE.h2o to the corresponding record and we find all corresponding STUDENT_TID's. With each student address we access to the corresponding STUDENT-record. Now we can take the COURSE-value once more to have direct access to each COURSE-subtuple. We see that we do not need an additional address for the (COURSE, MARK)-repeating group. It should be remarked that a COURSE index for STUDENT.h2o is not needed at all, if a (STID, COURSE)-pair is given. With a STID-index we find the corresponding student, and with the mini-directory (if it exists) we find the corresponding (COURSE, MARK)-subtuple. That means, it is not necessary to compute the intersection of the addresses of the given STID with the set of addresses of the given COURSE.

If we want to create a MARK index for STUDENT.h2o then we have at first the possibility to include the COURSE in the index:

I8.h2o: M(MARK, M(COURSE, STUDENT_TID))

If a mark is given then we find corresponding student addresses and courses. With the address we find the student and with the course the subtuple. But if the STUDENT-records are smaller than a page then we do not save page accesses. Therefore a smaller index without course would suffice in this application surely:

I9.h2o: M(MARK, B(STUDENT_TID))

In I9.h2o the repeating group should be a bag, because in the case of deletion of a (COURSE, MARK)-subtuple we could not always delete the corresponding STUDENT_TID of a corresponding set. Because of the mini-directory in I8.h2o the COURSE has the function of a subtuple address. Therefore by indexes of type I8.h2o we have all manipulation possibilities which are given by hierarchical addresses.

Finally we have two remarks. First, we save a lot of storage space and computing power, if we store STUDENT in the above H2O file. If we would store the data in a relational model then we have to define a further relational file M(STID, COURSE, MARK). For this file a STID-index is necessary. If a student has in average 20 courses, then this index is 20 times larger than I2.h2o. Further, two STID-indexes are needed. It is important to remark, that it is also possible to free primary files completely from index data as in the relational model. If we have for example STUDENT.h2o and FACUL.h2o, which contain only primary data, then the FAC index can be defined also in the following way:

I10.h2o: M(FAC, FACUL_TID, M(STUDENT_TID))
I11.h2o: M(M(FAC, I10_TID))

5 Metadata

The metadata of all H2O files have the same schemes. They will be stored as a complex record with the TID (1 1). This is in general the first record in the first page. Therefore, the metadata can grow and shrink, if we create or delete an index on the corresponding file. Insertions of new columns in the file, which would cause a growing of metadata, we will not consider. If the metadata record (M-record) is greater than a page then the first section of the record is in an M-page. If the M-record is smaller than a page, then it is in an A-page.
Scheme of metadata:
FIRST_F30, FIRST_F70, FIRST_F100, FIRST_FREE, KEYCNT1,
L(TAG, TYPOS), M(SETNO, KEYCNT2), M(INDEXNAME)
FIRST_F30: page number of the first F30-page (page with more than 30% free storage area)
FIRST_F70: page number of the first F70-page
FIRST_F100: page number of the first F100 page
FIRST_FREE: page number of the first free page; if a bulk of pages a reserved at once; this is to avoid scattering pages on the secondary storage area; these pages are not included in the F100 free page chain.
KEYCNT1: count of key fields of the outermost set (we presuppose that the key fields are always the first elementary fields)
L(TAG, TYPOS): DTD (Document Type Definition) of the H2O file; the first TAG is TABMENT
SETNO: number of a set repeating group
KEYCNT2: Number of the atomic fields, which compose the key of the set

It is clear that only the topmost sets are counted. Lists and bags are not considered because they do not contain proper keys (for bags we take the first field and for lists the sequence number). The set number has to be transformed by the system to the repeating group number (RGno).

Example: M(A1, A2, L(B1), M(C1, C2, M(D1, D2)), B(E2), M(F1, F2, F3)):

SETNO,	KEYCNT2	RGno	
1	2	3	(C1, C2)
2	1	5	(F1)

INDEXNAME: Name of an H2O file, which contains addresses to the given file; indexes of the second level are not included
An H2O file begins always with a file header at the beginning of the first page. This header is of type:
VERSION, PAGESIZE, PAGECNT, ENCODING

It does not belong to the metadata. Therefore, it cannot be transferred, if the metadata record grows too much.

As remarked above the metadata record has always the address “<B>1</B><S>1</S>”. This holds for index files of type M(M(A, D_TID)) too. Besides, the metadata record, the file contains only one (complex) record. For this record we need an entry address. This has to be “<B>1</B><S>2</S>”. We will guarantee this, by choosing the page size always as large such that the M-record, which is indeed small in this case fits in the first page. Then we can start to insert the (A, D_TID)-subtuples also in the first page. Therefore, the record can hold always the address (1 2). This address remains unchanged, also if the record grows over the page size.

6 Extensions

Firstly we will store the complex records like small XML-files, eventually without root. This allows a quicker implementation. Further, we have the advantage that empty optional values do not require storage space and that each field can be nearly arbitrarily long. But a subtuple is not allowed to be longer than a page. The storage of H2O records as XML-files allows also a generalization of H2O tables. The number of nesting level of two is not so hard to keep in practice. We will allow for example a file of type

PUPIL.h2o: M(NAME, FIRSTNAME, FATHER, MOTHER, CLASS,
M(SUBJECT, TEACHER, L(MARK)))

as an H2O file.

A subtuple is of type (SUBJECT, TEACHER, L(MARK)), where SUBJECT is the key and the mini-directories would contain only SUBJECT-data. A cut will be done only between complete subtuples.

An arbitrary index, like

I12.h2o: M(CLASS, M(PUPIL_TID))

requires a lot of tag data (CLASS, PUPIL_TID, B, S), we plan to avoid the tags in later versions. We could separate number data by an empty space. Index manipulations like the intersection of address sets will be more efficient.

Furthermore, we plan to allow TEXT-data of fixed length. Then we can save storage area for tag data in different cases and XML or HTML-files can be stored as lists of text of length 1 (TEXT1).

DOCS.h2o: M(DOCID, L(AUTHOR), ABSTRACT, …, DTD, DOC)
DOC: XML
ABSTRACT: XML
XML: L(TEXT1)
…

An XML document is stored simply as a list of text of length one. Therefore, the XML document can be cut at arbitrary position and it contains no additional tags of the H2O level. Furthermore, we have direct access to an arbitrary part of the document, and we can insert or delete parts of a document without writing the whole document.

Further, we can define a full text index **I13.h2o** by the following "gib-aus-mit"-construct (SELECT-FROM-WHERE-construct):

```
aus      DOCS.h2o                      # aus: from
replace  DOC by worte(DOC)             # transfers each DOC in a list of words
gib      M(WORT, M(DOCS_TID, C)) && # WORT: word
         C:=count(WORT)
```

Once more an index

I14.h2o: M(M(WORT, I13_TID))

should be defined. If we are looking now for all documents, which contain the words "Hadmersleben" and "Magdeburg", then we can find across I14.h2o the complex records with keys "Hadmersleben" and "Magdeburg". The corresponding DOC_TIDS can be considered as posting lists. We have to intersect both lists. If a list is greater than a page, then the corresponding mini-directory entries can be considered as skip pointers (compare [14]). Therefore the intersection of two TID-lists can be computed very efficiently. It is evident, that this "skip pointer principle" can be used for two different indexes on one H2O table.

7 First Steps to Query Optimization with H2O Files for ottoQL

Because of space limitations we consider only three examples of ottoQl-programs with the corresponding translated ottoQL-programs on the student file with an index. It becomes clear that we plan firstly only query optimization techniques at logical (ottoQL) level.

STUDENT.h2o: M(STID, NAME, FAC, LOC, M(COURSE, MARK))

I2.h2o: M(M(STID, STUDENT_TID)),

Program 1: Find the student with number 1234!

```
aus doc("STUDENT.h2o")
mit STID="1234"                  # mit: where: selection
```

has to be transferred to

```
aus doc("I2.h2o") value_subtid (<B>1</B><S>2</S><STID>1234</STID>)
gib M(STUDENT_TID)   # gib: similar to restruct from [1]
replace STUDENT_TID by (doc("STUDENT.h2o") &&
        value_tid STUDENT_TID)
```

Program 2: Give three students with numbers 1234, 5678 and 9101!

```
aus STUDENT.h2o
mit STID in L["1234"; "5678"; "9101"]
```

has to be transferred to

```
aus doc("I2.h2o")
mit STID in L["1234"; "5678"; "9101"]            # sequential scan
gib M(STUDENT_TID)

replace STUDENT_TID by (doc("STUDENT.h2o") &&
        value_tid STUDENT_TID)
```

or 3 times the first two rows of program 1 and union the three STUDENT_TID's and applying the following replace.

Program 3: Give all computer science students from Magdeburg!

```
aus     doc("STUDENT.h2o")
mit     LOC="Magdeburg"
mit     FAC="Computer Science"
```

Firstly we assume that no FAC-index, but a two-level LOC-index exists:
LOC1.h2o: M(M(LOC, LOC2_TID))
LOC2.h2o: M(LOC, M(STUDENT_TID))

```
doc("LOC1.h2o") value_subtid &&
     (<B>1</B><S>2</S><LOC>Magdeburg</LOC>)
ext     (doc("LOC2.h2o") value_tid LOC2_TID) at LOC2_TID
gib     M(STUDENT_TID)
replace STUDENT_TID by (doc("STUDENT.h2o") value_tid
        STUDENT_TID)
mit     FAC="Computer Science"
```

If we have additionally a FAC-index we compute for each condition a set of student addresses, intersect them in "skip pointer manner" as mentioned above and replace the addresses by its values.

8 Related Work

The storage structures of AIM/P and DASDBMS [12], [8], [9] are very similar to each other. A collection is stored with the help of a pointer array. Contrary to our approach mini-directories are used also in the case of a small complex record. Therefore, especially small subtuples need additional storage area for pointers. A pointer to a segment consists in general of all addresses of superordinated segments. In our approach a TID-address of the whole record is sufficient in most cases. This is because our records are in general not deep structured and much smaller than the records, which had the developers from AIM/P and DASDBMS in mind. Our records are forced to have a key contrary to AIM/P and DASDBMS.

In [6] and [15] it is tried to melt XML and relational technology. XML-documents are stored as values of an XML column. Corresponding considerations are not discussed in detail in this paper. But contrary to both approaches we do not have in mind to allow very large XML documents as row values. We think it is better to store these large documents as one independent H2O file or a view of H2O files. Further, we will use for our system only one language (ottoQL ([5])) and not SQL/XML and XQuery.

In [10] four types of indexes are presented. Value and text indexes can directly represented by H2O structures as described above. Link indexes we do not need, because the access to a subtuple goes always over the parent (head) row. We shall consider path indexes in future work.

In [11] a general method for storing and updating XML data is described. There, proxy and helper nodes are used, when a new node is inserted into a page, which is too small for the grown record. Our "helper nodes" are foreign keys, which are inserted already at design phase of the database. That means we split a deep structured document into several "tables" such that each record will be sufficient small. If we introduce corresponding keys then also the order of the documents can be maintained.

9 Summary

1. The described storage structure for H2O files contains with a minimal set of concepts components of sequential (XML-files), relational, hierarchical and network (object-oriented) storage. The access to subtuples is represented either by physical neighborhood (XML-files) or by access from head record via mini-directory (hierarchical systems). Address fields (TID's) could allow a cross linking of files (network, object-oriented). Address-fields (repeating groups) do not carry information (relational systems), therefore we can exclude them from end user view, but they are useful for computer scientists. Each H2O file can be used independently from other files (relational ideas).
2. The file concept allows a clustered storage of records of different types. These clusters are stable after updates. Contrary to the relational model the clusters are visible already at end user level. Therefore, our whole system will be simpler (no additional physical level).
3. Subtuples are stored sorted. Therefore, and because of a small mini-directory we have direct access to subtuples. The sorting of subtuples in indexes allows a sorted access to corresponding tuples. The sorting and structuring of data can later be used in query optimization (e.g. with stroke compare [3]).
4. The addressing concept is relatively simple. We use only one type of addresses (TID's). Because of the ordering of subtuples we need only a sparse heterogeneous index (mini-directory) for large complex records. Nevertheless index manipulations can be realized unrestrictedly. The sparse mini-directory allows a storage of very short subtuples (address sets, sequences of points, etc.). The soft addressing allows a transfer of subtuples over page borders. This will simplify the implementation of update operations.
5. (External) indexes (full text indexes included) can be implemented also by H2O files or by repeating groups in other files. Therefore, all functions, which will be implemented for primary H2O files, can be applied to indexes. H2O indexes replace B trees and inverted files.
6. Metadata do not require a separate file concept. They are stored as a structured tuple of fixed type with address (1 1). In other implementations all metadata of a database could be stored in one or more H2O files.

Acknowledgement. I would like to thank R. Aretz and X. Li for starting the H2O files implementation and for valuable remarks to this paper. Further, I thank Andreas Luebke for careful reading the paper and worthwhile comments.

References

[1] S. Abiteboul, N. Bidot, „Non-First-Normal-Form Relations: An Algebra Allowing Data Restructuring", J. Comput. System Sci: 1986, pp. 361-393

[2] K. Benecke, "On hierarchical normal forms", in Proc. 1st Symposium on Mathematical Fundamentals of Database Systems, J. Biskup, et al. (Eds), MFDBS 87, LNCS, Dresden 1987, pp.10-20

[3] K. Benecke, „A Powerful Tool for Object-Oriented Manipulation", in Proc. IFIP TC2/WG 2.6 Working Conference on Object Oriented Database: Analysis, Design & Construction, Windermere, UK, pp. 95-122, North Holland 1991

[4] K. Benecke, "On Powerful and User Friendly Assignments for XML", submitted for publication

[5] K. Benecke, M. Schnabel, Internet server for *ottoQL*: http://otto.cs.uni-magdeburg.de/otto/web/index.html

[6] K. Beyer, R. J. Cochrane, V. Josifovski, J. Kleewein, G. Lapis, G. Loman, B. Lyle, F. Özcan, H. Pirahesh, N. Seemann T. Truong B. Van der Linden, B. Vickery, C. Zhang "System RX: One part Relational, One Part XML" SIGMOD 2005, June 14-16 Baltimore, Maryland, USA

[7] ed.: D. Chamberlain et al., "XML Query Use Cases", W3C Working Draft 23, March 2007, http://www.w3.org/TR/xmlquery-use-cases

[8] P. Dadam, K. Küspert, F. Andersen, H. Blanken, R. Erbe, J. Günauer, V. Lum, P. Pistor, G. Walch, „A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View on Flat Tables and Hierarchies" In ACM-SIGMOD, Proc: Int. Conf. on Management of Data, Washington D.C., pp 356-367, 1986

[9] U. Deppisch, H.-B. Paul, H.-J. Schek. „A Storage System for Complex Objects", In K. Dittrich, U. Dayal, Ed. Int. Workshop on Object-Oriented Database Systems, pp. 183-195, IEEE Computer Society Press, 1986

[10] J. McHugh, J. Widom, S. Abiteboul, Q. Luo, and A. Rajamaran, "Indexing semistructured data", Technical report, Stanford University, Computer Science Department, 1998. http://citeseer.ist.psu.edu/mchugh98indexing.html

[11] C. C. Kanne, G. Moerkotte, "Efficient storage of XML Data" Technical Report Nr. 8, Lehrstuhl für praktische Informatik III Universität Mannheim June 1999

[12] V. Lum, P. Dadam, R. Erbe, J. Günauer, P. Pistor, G. Walch, H. Werner, J. Woodfill, „Design of an Integrated DBMS to Suppoert Advanced Applications", in Blaser, P. Pistor, Ed., Datenbank-Systeme für Büro, Technik und Wissenschaft; pp. 362-381, GI-Fachtagung, Springer, Informatik-Fachberichte, Nr. 94 1985

[13] G. Moerkotte, "Building Query Compilers" (Under Construction) http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf

[14] C. D. Manning, P. Raghavan, H. Schütze, "An Introduction to information retrieval" Campridge University Press, http://nlp.stanford.edu/IR-book/html/htmledition/irbook.html

[15] S. Pal, I. Cseri, O. Seeliger, G. Schaller, L. Giakoumakis, V. Zolotov, "Indexing XML Data Stored in a Relational Database", Proc, 30th VLDB Conf. Toronto Canada 2004

[16] H.-J. Schek, P. Pistor, "Data Structures for an Integrated Data Base Management and Information Retrieval System", In Proc. 8th Int. Conf. on Very Large Data Bases, Mexico City, Mexico pp. 197-207, 1982

[17] J. D. Ullmann, “Principles of Database Systems”, Computer Science Press, Potomac, Maryland 1980