The Dialectic Storage Structure

Klaus Benecke

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

The dialectic file solves the following contradictions:
1. simple and complicated
2. sorted and unsorted
3. primary data and secondary data (indexes)
4. hash and tree
5. column by column and row by row
6. hierarchical und relational
7. text file and addressable
8. permanent and transferable
9. table and document
10. tabment (TABle+docuMENT) and tabments
11. formatted (SQL) und unformatted (noSQL)

Abstract. This paper describes a universal storage structure (a dialectic storage structure), which is settled between relational and hierarchical approach. It aims to marry XML files with the TID concept and row- with column-oriented storage. We support at first glance only nesting level of proper collections 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, IMS). Besides arbitrary primary data also indexes can be stored with our I-concept. Value- and full text-indexes can be expressed by DIA-file structures whereas parent- and path-indexes are not necessary, because the access to subtuples goes always through the head record (parent node). The DIA-storage structure aims to be a base for a universal, simple, and lean information system with our query language OttoQL on 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. DIA-file files are intended to replace XML files, if these files contain a very large number of records and if we need direct access to (a part) of these records. DIA-file files are as similar as possible to XML files, but differ in following points:

1. A DIA-file file in general is a set of complex records with a key, consisting of zero or more elementary fields.
2. Beside lists, in a DIA-file record sets or bags may occur as repeating groups.
3. If a DIA-file record is larger than a page (a page consists of several blocks) then the record is split into two or more sections and a mini-directory is generated in the first "B"- and "C"-pages. (Contrary to these pages "A"-pages contain short complex records or sections of large or very large records.)
4. The elements of set- and bag-repeating groups are sorted by the first fields.
5. To save storage area in the deepest collections, tag data will not be repeated. <L(X_1, X_2,…, X_n)> tags with separated values will be introduced.
6. Column oriented storage is allowed for top level columns.

DIA-file files agree with XML files in the following points:

1. A DIA-file table without column-orientation is stored at first in exactly one physical file.
2. Metadata are contained in the file (in a metadata record).
3. A DIA-file file is transferable like an XML file from one computer to another (in the internet). It contains no binary data, if it is not compressed.
4. A DIA-file record is up to a root tag a (small) XML document.

In the second section we define DIA-files in a narrower sense and sketch how high structured very large XML documents can be mapped to DIA-files. Section 3 contains the description of the DIA-file storage structure in detail. In section 4 restrictions of DIA-file-files are discussed, to simplify a first implementation. Some index structures are presented in section 5. It becomes clear that DIA-file indexes can be used in many kinds of applications. We think that they can replace B-trees and inverted files. The details of metadata are presented in section 6. Possible extensions of the file concept are described in section 7. We demonstrate that the query optimization can be realized on top (OttoQL- (compare [5])) level by five simple examples. Finally, we consider related work and a summary.

2 DIA-files

Definition: A DIA-***file in the narrower sense*** is a table, whose hierarchical levels of proper collections have maximal depth of two and which contains in the first level a key, which consists only of elementary fields (An elementary field is of string or number type). Optional values ("?", "|") and tags are not considered as proper hierarchical levels such that they can occur in arbitrary depth.
DIA-files are more general than flat relations, but in general not as high structured as XML files. It is in general possible to replace a large XML file by one or more DIA-file tables and/or XML documents. XML files are sufficient, if the corresponding files are relatively small. We consider only one simple example:
Dat.xml: L(A1, A2, L(B1, B2, L(C), L(D1, D2)), L(E1, E2)) # L abbreviates list

can be replaced by
H1.xml: L(A1, A2, L(E1, E2)) and
H2.dia: M(B1, B2, A1, L(C), M(D1, D2)) # M abbreviates "Menge" = set
Here, we assume that *A1* is a key in *Dat.xml* and *{B1, B2}* and *{D1, D2}* contain a local key in *Dat.xml*. Nevertheless the table *Dat.xml* can be expressed by an *OttoQL*-view:

```
    aus doc("H1.xml")                                # aus: from
    ext {aus doc("H2.dia") ; mit A1=A1'}  at A2      # ext: extension; mit: selection
```

Here, "{", "}" are the brackets for a subprogram, A1' is A1 one level outside the {}-brackets. If the table *Dat.xml* has no corresponding keys, then they have to be generated by the system (OIDs). But we assume that very large data collections have nearly always keys. Because *H1.xml* contains only fields of higher levels it is relatively small compared to the DIA-file file. Therefore, XML-files should be processed in main memory contrary to the very large DIA-file-files.
If we want to represent an n:m-relationship between large entity sets *A* and *B* by DIA-file tables then two DIA-file tables are needed:

DatA.dia: M(A1, A2,..., M(B1, C1, C2, ...)),

DatB.dia: M(B1, B2,...)

Here, *C1, C2*, … are the attributes of the relationship, whereas *A1, B1* are the keys of entity sets *A* and *B*, respectively. The table *DatA.dia* represents the "join" between *A* and the "relationship file" **C**: M(A1, B1, C1, C2, ...).

3 The DIA-file Storage Structure

We represented each DIA-file table without column oriented storage directly by one file to keep the first version as simple as possible. The storage structure will be introduced stepwise. For a complex record three 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 record).
3. The mini-directory of a complex record is larger than a page (very large record).

Independently of the size of a complex record the address of a complex record consists of a page address (in cases 2 and 3 the addresses of the first page) and a slot number. By the slot number the position of the complex record in the page is expressed (in cases 2 and 3 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 DIA-file file sorted. The sorting and identification of the complex records of the DIA-file file will be realized by an additional primary index, which is also a DIA-file file, as we show later.

If a small record grows and there is not enough free space in the page, then the record is written in a new page with enough free space. Because updating indexes would require too much time, 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 by the new address, such that we need also only two accesses to read 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 such 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 or very large records, is called A-page.

Sections of large and very large records are addressed in the same way as small records - by TIDs. 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; the offset refers to the beginning of the section, which contains the first byte of the 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

We shall see later, that we will consider a head record of a complex record also as repeating group subtuple. 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 (item 1) are needed for a quick access to subtuples and later for a saving of tag data. The offsets in item 1 are calculated from the beginning of the complex record or the beginning of the corresponding section in the case of a large record. In later generalizations a subtuple is allowed to contain collections. The corresponding subtuples are not supported by the offsets of item 1 and not considered in the minidirectory. Therefore all subtuples are stored in depth first strategy.

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 either be divided or it, or parts of 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 + slot number (position)

page foot = constant part + variable part
constant part = pages status (A, B, C, 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
+ page number of the next C-page (only in B- and C-pages)
C: page following a B- or C-page; containing the minidirectory of the complex record (repeating group numbers + keys + TID's of sections of the very large record); B→C→C→…→C: the last C page is allowed to contain also primary data (the first section)
M: starting page for a large meta record
F: completely empty page

variable part = TID-entries (status A)
mini-directory (status B, C, 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 or very 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)

TID-entries are separated by blanks, but RecordKind and Offset are directly concatenated.

mini-directory-entry: RGNO: number of the repeating group of the first subtuple of the section
+ key (set, bag), sequence number (list) of the first subtuple (it is possible that the key of a set consists of several fields)
+ TID of the section of the large or very large record

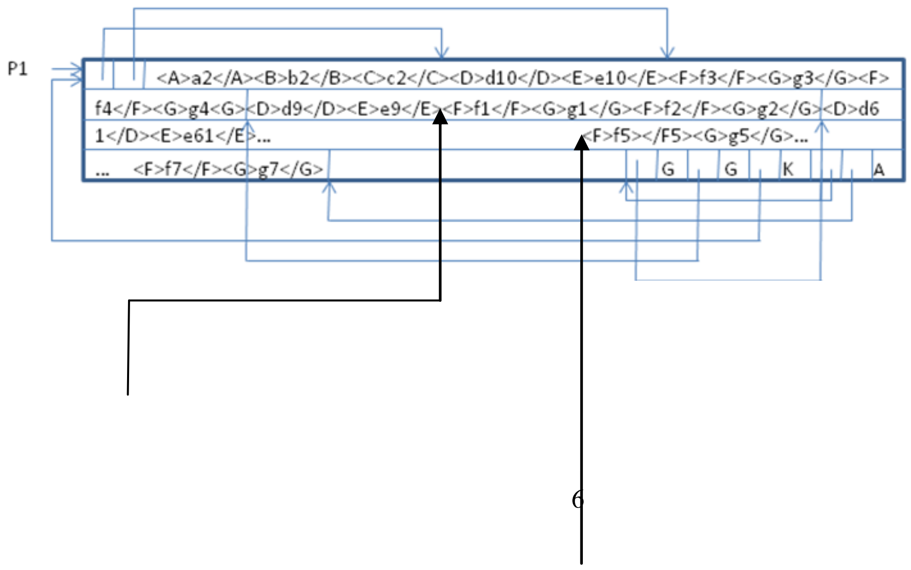
Now we can define for the page foot schemes:
Foot of an A-page:
PageStatus, FreeOffSet1, FreeOffsetLast, FreePageSucc,
FreePagePred, L(RecordKind,OffSet)
Foot of a B-, C-, or M-page:
PageStatus, FreeOffSet1, FreeOffsetLast, FreePageSucc, FreePagePred, Next_C,
L(RGNO, L(KeyValue), SectionTID)

A (RGNO, L(KeyValue), SectionTID)-tuple is a minidirectory entry. These entries will be separated by ";" and RGNO, KeyValue, and SectionTID will be separated by ",". Page number and slot number will also be separated by ",". For illustration purposes we consider in figure 1 an example of a DIA-file table in a 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. A DIA-file table in narrower sense with cuts in tabment representation

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


P1
<A>a2</A><B>b2</B><C>c2</C><D>d10</D><E>e10</E><F>f3</F><G>g3</G><F>
f4</F><G>g4<G><D>d9</D><E>e9</E><F>f1</F><G>g1</G><F>f2</F><G>g2</G><D>d6
1</D><E>e61</E>...
<F>f5></F5><G>g5</G>...
... <F>f7</F><G>g7</G>
G
G
K
A

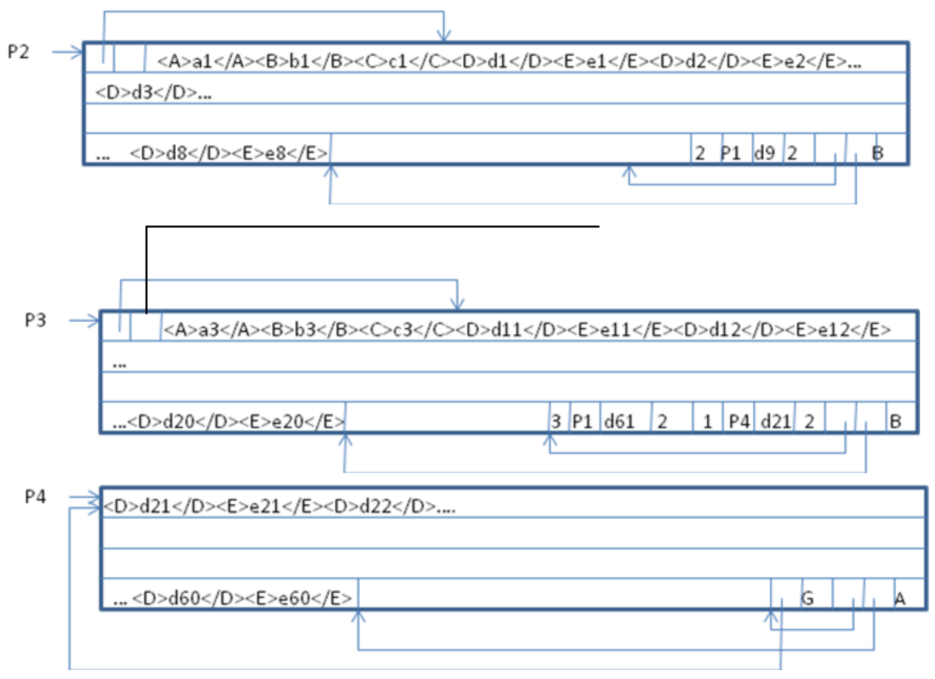

P2
<A>a1</A><B>b1</B><C>c1</C><D>d1</D><E>e1</E><D>d2</D><E>e2</E>...
<D>d3</D>...
... <D>d8</D><E>e8</E>
2 P1 d9 2
B
P3
<A>a3</A><B>b3</B><C>c3</C><D>d11</D><E>e11</E><D>d12</D><E>e12</E>
...
...<D>d20</D><E>e20</E>
3 P1 d61 2 1 P4 d21 2
B
P4
<D>d21</D><E>e21</E><D>d22</D>....
... <D>d60</D><E>e60</E>
G
A

Fig. 2. DIA-file storage structure of the table of figure 1 without metadata

Because we store a record as a small XML file, we can allow relatively general schemes (optional values + choice) in our DIA-file storage concept. This can be illustrated by an example of more complex type and an example, which is important for indexes:

M(A1, A2, M(B1, B2, B3), C1, C2?, M(D1, M(D2, D3)), E1, (E2 | E3), B(F1, F2?)), L(G1, G2, L(G3 | G4)))

Here, we have 7 "repeating groups" with the following number of keys:

1. A1, A2: 0 (Because we have only one (A1, A2)-subtuple)
2. M(B1, B2, B3): 2 ((B1, B2) is assumed to compose the local key)
3. C1, C2?: 0 (because C1 - and C2 -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, (E2 | E3): 0
6. B(F1, F2?): 1 (for bags we set the number of "keys" by definition to one (F1))
7. L(G1, G2, L(G3 | G4)): 1 (The sequence number of the outer list)

A DIA-file table of the type

M(M(A1, B1))

contains only two "repeating groups":

1. "": 0 (No field exists at first level.)
2. M(A1, B1): 1 (A1 is assumed to be the key)

We simply want to 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). Therefore 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 be written in the following way:

K001 K200 G400 K430 … (forward notation)

This means that the first (short) record has size 199 bytes, the second 200 bytes, the section of a large record a length of 30 bytes, etc.

A repeating group L(RGNO, L(Keyvalues), SectionTID)) from a foot of a B-page or C-page of the above first scheme with seven repeating groups looks for example as follows:

2,b1,b2,23,2;5,5666,6;7,301,4666,8; etc.

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 seventh repeating group (list). Such sequences of characters can be interpreted, because we know how many key fields each repeating group contains. Therefore a corresponding entry can be found by a modified binary search. If a DIA-file 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 a certain percentage of the page size (30 or 40?). Then the number of sections is restricted and calculable and the mini-directory will not be too large and estimates for the page size for a given maximal record size can be computed.

4 Restrictions of the DIA-file Storage Structure

We should consider at first only XML documents of the following type:

$M(Key_1, Key_2,\ldots,Key_n, F_1, K_1(S_1)\,, F_2, K_2(S_2), \ldots\,, F_n, K_n(S_n))$

where each field of the scheme is of type TEXT, ZAHL, or PZAHL. Key_1, …, Key_n, compose the key of the outmost set. F_i (i=1,…, n)contains no proper collection symbol (M, B, L), but „?“ and „|“ are allowed within Fi.

$K_i \in \{K_1,K_2,\ldots,K_n\}$ is either M, B , L, M-, or B-.

For K_i=M the key consists of the first atomic fields. Further, we assume that all fields have different names. Therefore the following example situations are not allowed:

1. Atomic fields in level zero

UNI, M(STID, NAME,… FAC,… , L(HOBBY),…)

Solution: UNI in an additional (small) XML file: UNI, M(FAC, DEAN,…) or

UNI in the COMMENT-field of the metadata of the file

2. An M-repeating group has a composed key field:

M(CLASS,…, M(NAME,…)…)

NAME=(LAST, FIRST)

Solution: replace NAME by LASTNAME, FIRSTNAME
3. Tags around collections:
M(NAME, …, PHYSICS, MATHS)
PHYSICS: L(MARK)
MATHS: L(MARK)
Solution: M(NAME,…, L(PHYSICS), L(MATHS))
4. Tags around tuples of collections
M(STID,…, M(EXAM),…)
EXAM: COURSE, MARK, DATE
Solution: M(STID,…, M(COURSE, MARK, DATE),…)
5. Recursive lists
BOOK: L(TITLE, L(SECTION))   SECTION: TITLE, L(SECTION)
Solution: not a DIA-file- but an XML file
But a file of books is larger than a file of one book such that it should be represented by a DIA-file file. BOOKS: M(ISBN, A_DATE, ABSTRACT, L(TITLE, L(SECTION)))
PARTS: M(PART) PART: PART_NO, NAME, M(PART, OCCUR_NO))
Solution:
PARTS2: M(PART_NO, NAME, M(DIRECT_SUBPART_NO, OCCUR_NO))
6. Keys with "?" or "|"
The following repeating groups are not allowed:
L(A1?, A2?) , L(A1 | A2), and L(TEXT | Y), because the start of an element is not well defined.
7. Any
We do not allow ". . ." (Any) in the DTD of a DIA-file file, but ". ." (Any_flat); therefore each proper collection appears with a first field in the given scheme. The following example is nevertheless possible:
M(STID,NAME, . . M(COURSE, . . L(MARK,. .)), M(HOBBY, . .))

5 Indexes

We shall see in this section that it is possible to define several different kinds of indexes with DIA-file files. This has the advantage that the basic functions for retrieval and update have to be implemented only once and OttoQL programs can be used at physical level, too.

5.1 Row-Oriented Indexes

The DIA-file indexes, introduced in this subsection, can be compared with B-trees. But for larger files there is less redundancy than in B-trees. Let's start with two simple tables:

STUDENT.dia: M(STID, NAME, FIRSTNAME?, FAC,
REGISTER, LOC, SCHOLARSHIP,
M(COURSE, MARK), L(HOBBY))

FACUL.dia: M(FAC, DEAN, STUDCOUNT)

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

I1.idx.dia: M(STID, STUDENT_TID)

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

I2.idx.dia: M(M(STID, STUDENT_TID)),

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

I3.idx.dia: M(FAC, M(STUDENT_TID)),

which contains each FAC-value only once and additionally a primary index for this index. This primary index is necessary, because we want direct access to the FAC-elements of I3.idx.dia.

I4.idx.dia: M(M(FAC, I3_TID))

If we want all students of a given faculty, we at first look with the help of I4.idx.dia to the corresponding I3.idx.dia address and then have direct access to the corresponding STUDENT_TID repeating group of the I3.idx.dia file.

Now, we see that I3.idx.dia and FACUL.dia have both FAC as a key. So we could theoretically extend either FACUL by the corresponding address repeating group, or I3 by DEAN and STUDCOUNT:

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

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

I5.idx.dia: 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.dia:

I6.idx.dia: M(STID1, M(STID2, STUDENT_TID))

I7.idx.dia: M(M(STID1, I6_TID))

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

```
aus  doc(“STUDENT.dia”)                    # “aus” is German and means „from”
ext  STID1:= STID subtext (1, 5)           # ext: extension by a new column
ext  STID2:= STID subtext (6, 3)           # STID is assumed to consist of 8 digits
gib  M(STID1, M(STID2, STUDENT_TID)) #  “gib”: similar to restruct from [1]
```

Here, STUDENT.dia is extended by the virtual column STUDENT_TID:

STUDENT.dia: M(STUDENT_TID, STID, NAME, FAC, LOC,
SCHOLARSHIP, M(COURSE, MARK), L(HOBBY))

The repeating groups in I6.idx.dia and I7.idx.dia are much smaller than in I2.idx.dia. Therefore, for a very large STUDENT-file a simple index file has to be replaced by a two or even three step index. In general the question, whether an index has to be realized by one or two DIA-file files can be answered by a rough estimation, if the mini-directory fits in one or very few pages 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) (Here, ^ is the concatenation). 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 into two or more files has also an additional advantage. Assume we have 10,000 students. Then it may be a good approximation to have 100 STID1-values. Therefore, I6.idx.dia, along with I7.idx.dia, contains only 200 STID1 values. But in I2.idx.dia we have 10,000 STID1-values. This is a considerable saving of storage space, compared to I2 or to B-trees.

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

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

If we want to find all results of examinations for a given course, we can access the given primary index of COURSE.dia for the corresponding record and find all corresponding STUDENT_TIDs. With each student address we access the corresponding STUDENT-record. Now we can take the COURSE-value once more to have direct access to each COURSE-subtuple with the help of the minidirectory (if we have a large or very large record). We see that we do not need an additional address for the (COURSE, MARK)-subtuple. It should be remarked that a COURSE-index for STUDENT.dia 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.dia, we have at first the possibility to include the COURSE in the index:

I8.idx.dia: M(MARK, M(COURSE, STUDENT_TID))

If a mark is given, 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, we do not save page accesses. Therefore a smaller index without course would suffice in this application probably:

I9.idx.dia: M(MARK, B(STUDENT_TID)) or

I9b.idx.dia: M(MARK, M(STUDENT_TID, MARK_CNT))

In I9.idx.dia 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 mini-directories in STUDENT.dia the COURSE in I8.idx.dia has the function of a subtuple address. Therefore by indexes of type I8.idx.dia 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 processing costs, if we store STUDENT in the above DIA-file file. If we would store the data in a relational model then we have to define a further relational file M(<u>STID, COURSE</u>, 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.idx.dia. 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.dia and FACUL.dia, which contain only primary data, then the FAC index can be defined better in the following way:

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

5.2 Column-Oriented Storage

Row oriented storage is too slow for OLAP applications. The DIA-file storage structure allows a storage column by column, too. We consider our student file once more:
STUDENT.dia: M(<u>STID</u>, NAME, FIRSTNAME?, FAC, REGISTER, LOC,
SCHOLARSHIP, M(COURSE, MARK), L(HOBBY))

Because *exam*- and *hobby*-data are stored successively, we can consider this already as a restricted column oriented storage. It can be expected that aggregations per student are efficiently to realize. But we can take this file further apart:

STUDENT_i.dia: M(STUDENT_TID, FIRSTNAME?,
M(COURSE, MARK), L(HOBBY))
# STUDENT_TID is a virtual column
STUDENT_STID.col.dia: M(L(STID))
STUDENT_NAME.col.dia: M(L(NAME))
STUDENT_FAC.col.dia: M(L(FAC))
STUDENT_REGISTER.col.dia: M(L(REGISTER)
STUDENT_LOC.col.dia: M(L(LOC))
STUDENT_SCHOLARSHIP.col.dia: M(L(SCHOLARSHIP))
STUDENT_PAGENO.col.dia: M(L(PAGENO))
STUDENT_SLOTNO.col.dia: M(L(SLOTNO))
STUDENT_COURSE1.idx.dia: M(<u>COURSE</u>, M(STUDENT_i_TID))
STUDENT_COURSE2.idx.dia: M(M(COURSE, STUDCOURSE1_TID))

We denote by STUDENT_POS the virtual position number of the outermost collection. All above “col” files use this position number for their inner lists. We will see that we can store all inner lists nearly without tag data.

If for example, we want all names of students from “Hadmersleben”, then we apply the condition LOC=”Hadmersleben” to STUDENT_LOC.col.dia and get all corresponding position numbers. With these position numbers, we find in STUDENT_NAME.col.dia all corresponding names.

If an original file without repeating groups and without optional fields is given, we can omit the last 4 files and the STUDENT_i.dia file contains only metadata.

In the above STUDENT example, it is visible that we in general define top-level “col” files for non optional attributes X only.

We shall see that it is possible to combine two or more of the above files to one file. *STUDENT_PAGE_NO.col.dia* and *STUDENT_SLOT_NO.col.dia* can be replaced by

STUDENT_TID.col.dia: M(L(PAGENO, SLOTNO)) for example.

Further, we should test a redundant storage of certain “col” files. That means for certain queries it may be advantageously that the data of the column are stored in the “i” file, too.

5.3 Hybrid Indexes

Let us consider the file STUDENT_FAC.col.dia (type: M(L(FAC))) again. It is clear that each faculty appears repeatedly for every student. We have in general to scan the whole col-file, if we want all position numbers of a given faculty. We can improve the situation, if we replace the col-file by the two following files:

STUDENT_FAC.col2.dia: M(FAC, M(STUDENT_POS))

I12.idx.dia: M(M(FAC, STUDENT_FAC.col2_TID))

Now each faculty is stored only twice and all student positions of a faculty can be computed very fast over I12.

If a position number is given and we want the corresponding faculty, then we have to jump from faculty to faculty and look whether the position number is in the corresponding repeating group. If a repeating group consists of several sections, then we have to look only in one section. The disadvantage of this approach is that if we want to delete a student with position number *x* then we have to subtract *1* from all position numbers greater than *x*. Insertions can be realized efficiently, because we have to add only *1* to the greatest position number of the metadata of the given file.

It is also possible to store the fields of a repeating group column-wise. In the above example we could omit from STUDENT_i.dia the columns COURSE and MARK and add the relative small files

COURSE.col: M(STID, L(COURSE)) and

MARK.col: M(STID, L(MARK)).

For both files STID-indexes are needed or they are taken as a whole into main memory.

5.4 Few Remarks on Geometric Data

Here, we want to show that is possible to integrate non-geometric with geometric data. We consider at first Germany with its borders:

GERMANY.dia: M(<u>COUNTY</u>, M(<u>TOWN</u>, CITIZEN, CENTER), L(BORDERPOINT))
CENTER: X,Y
BORDERPOINT: X,Y

Assume we want to cluster the towns and villages of Europe with respect to their positions and we want a fast access to neighbors of a given town. Then we can for example extract from each X and Y coordinate the integers X10 and Y10, which result from X and Y, respectively, in the following way:

EUROPE.dia: M(<u>X10, Y10</u>, M(<u>X, Y</u>, TOWN, CITIZEN, COUNTY,…))
X10:= integer(X div 10)
Y10:= integer(Y div 10)

Especially the following indexes are useful:

XY_EUROPE1.idx.dia: M(X10, M(Y10, EUROPE_TID))
XY_EUROPE2.idx.idx.dia: M(M(X10, XY_EUROPE1_TID))

It is evident that there exist less (X10, Y10)-entries than (X, Y)-entries such that an index for (X10, Y10)- or (X50, Y50, …) is much smaller than an index for (X,Y)-pairs. Nevertheless we have direct access for each (X, Y)-value. With the help of XY_EUROPE2 and XY_EUROPE1 we find the corresponding TID and then we can find the desired subtuple by a minidirectory, if the corresponding complex record is larger than a page. Two possibilities exist for indexes of further fields:

TOWN_EUROPE1.idx.dia: M(M(TOWN, EUROPE_TID)) or
TOWN_EOROPE2.idx.dia: M(M(TOWN, X, Y, EUROPE_TID))

It is clear once more that each of both possibilities can be replaced by a two or three file index. The second version is useful only, if the complex records are very often larger than a page.

A set of polygons can be stored for example in the following way:

TOWN_BORDERS: M(TOWN, COUNTY, XMAX, XMIN, YMAX, YMIN, L(X, Y))

The two maxima and minima describe the smallest rectangle, containing all values of the border L(X, Y). Therefore we need surely instead of an (X, Y)-index only indexes for the maxima and minima. These indexes are again very much smaller than an (X, Y)-index.

5.5 A Remark on a Modified SSBM-Benchmark

Fact table:
LINEORDER.dia:
M(<u>ORDERKEY</u>, CUSTKEY, ORDTOTALPRICE, ORDERDATE, ORDERPRIORITY,
SHIPPRIORITY, M(<u>LINENUMBER</u>, PARTKEY, SUPPKEY, QUANTITY,
EXTENDEDPRICE, DISCOUNT, REVENUE,
SUPPLYCOST, TAX, COMMITDATE, SHIPMODE))

Keys are underlined.

The Dialectic Storage Structure

Dimension tables:

PART.xml: M(<u>PARTKEY</u>, NAME, MFGR, CATEGORY, BRAND1, COLOR, TYPE, SIZE, CONTAINER)

SUPPLIER.xml: M(REGION, M(NATION, M(CITY, M(<u>SUPPKEY</u>, NAME, ADDRESS, PHONE))))

CUSTOMER.xml: M(CREGION, M(CNATION, M(CCITY, M(<u>CUSTKEY</u>, NAME, ADDRESS, PHONE, MKTSEGMENT))))

DATE.xml: M(YEAR, M(MONTH, YEARMONTHNUM, YEARMONTH, MONTHNUMINYEAR, M(<u>DATEKEY</u>, DATE, DAYOFWEEK, DAYNUMINWEEK, DAYNUMINMONTH, DAYNUMINYEAR, WEEKNUMINYEAR, SELLINGSEASON, LASTDAYINWEEKFL, LASTDAYINMONTHFL, HOLIDAYFL, WEEKDAYFL)))

Here, *LINEORDER.dia* materializes the join between *Orders* and *LineItem* without denormalization. More correct, this is the materialization of the ***ext*** operation. *Lineorder.dia* has the advantage compared to the flat *lineorder* file from SSBM-benchmark, that each orderkey together with *custkey,..., shippriority* appear only once for each order. If we assume that each order has in average 20 lines, then the corresponding columns in column oriented storage need only 0.05 of the storage area of the flat file, if we do not use compression. Further, because *ordtotalprice* is stored only once for each order, aggregations on this column can be applied without prior eliminating of duplicates.

Query Q1:

```
select sum(extendedprice*discount) as revenue
from lineorder as lo, date as d
where lo.orderdate = d.datekey
and d.year = 1994 and (lo.discount between 2 and 4) and lo.quantity < 24.6;

aus doc(“lineorder.dia”)
mit orderdate in {aus doc(“date.xml“); mit year=1994; gib M(datekey)} and &&
    (discount between 2 and 4) and quantity<24.6 # “between” not yet implemented
gib revenue revenue:=sum(extendedprice*discount)
or:
aus doc(“date.xml“)
mit year=1994
gib M(datekey)
, doc(“lineorder.dia”)
mit orderdate in M(datekey) and (discount between 2 and 4) and quantity<24.6
gib revenue revenue:=sum(extendedprice*discount)
or:
aus doc(“lineorder.dia”)
```

```
ext doc(“date.xml“)  at shipmode
mit orderdate=datekey and year=1994 and (discount between 2 and 4) and quantity<24.6
gib revenue revenue:=sum(extendedprice*discount)
```

Query Q2:

```
select sum(lo.revenue), d.year, p.brand1
from lineorder as lo, date as d, part as p, supplier as s
where lo.orderdate = d.datekey
and lo.partkey = p.partkey
and lo.suppkey = s.suppkey
and p.category = 'MFGR#12'
and s.region = 'AMERICA'
group by d.year, p.brand1
order by d.year, p.brand1;
```

```
aus doc(“date.xml“)
ext lo:=doc(“lineorder.dia”) at weekdayfl
ext p:=doc(“part.xml”) at shipmode
ext s:=doc(“supplier.xml”) at container
mit orderdate=datekey and p/partkey=lo/partkey and lo/suppkey=s/suppkey and &&
      p/category=”MFGR#12” and s/region=”AMERICA”
gib M(year, M(brand1, su)) su:=sum(revenue)
or:
aus doc(“lineorder.dia”)
mit partkey in {aus doc(“part.xml”); mit category=”MFGR#12”; gib M(partkey)} and &&
     suppkey in {aus doc(“supplier.xml”); mit region =”AMERICA”; gib M(suppkey)}
replace partkey by  {aus doc(“part.xml”); mit partkey=partkey’; gib brand1}
replace orderdate by {aus doc(“date.xml”); mit datekey=orderdate’; gib year}
gib  M(year, M(brand1, su)) su:=sum(revenue)
```

Query Q3:

```
aus doc(“lineorder.dia”)
mit discount < 5
mit partkey in {aus doc(“part.xml”); mit category=”MFGR#12”; gib M(partkey)}
mit suppkey in {aus doc(“supplier.xml”); mit region =”AMERICA”; gib M(suppkey)}
gib M(suppkey, su1) su1:=sum(revenue)
replace suppkey by {aus doc(“supplier.xml”); mit suppkey=suppkey’; gib nation, city}
gib su, M(nation, su, M(city, su)) su :=sum(su1)
```

5.6 Hash Storage

We consider at first a flat file:
Students: M(STID, NAME, LOC?, FAC, GRANT?, CURR_VITAE)

Students0.dia: M(STID5, M(STID3, NAME, LOC?, FAC, GRANT?, CURR_VITAE)),
where STID = STID3++STID5
assume we have 500 000 students, then in average we have 5 students in one repeating group. That means we create at the beginning a file with 100 000 blocks for primary data. All the blocks are A-blocks. STID5 is the hash value of STID.
The repeating group is sorted by STID3.
Assume we want a student with a STID, where the STID3 repeating group contains 200 students. Then we can go through the mini directory and access directly the wanted record.
In general we can use the structure
Students.dia: M(HVALUE, M(STID, NAME, …, CURR_VITAE)) # here HVALUE is the hash value of STID.
A FAC-index: I1.dia: M(FAC, M(STID)) + I2.dia: M(M(FAC, I1_TID))
Full text index: I3.dia: M(WORT, M(STID)) + I4.dia: M(WORT, I3_TID)

Now, we consider a structured file:
Students_h: M(STID, NAME, LOC?, FAC, GRANT?, CURR_VITAE,
M(COURSE, MARK), L(HOBBY))
Students_h.dia: M(HVALUE, M(STID, NAME, LOC?, FAC, GRANT?, CURR_VITAE,
M(COURSE, MARK), L(HOBBY)))

FAC- index (see above)
COURSE-Index: I5.dia: M(COURSE, M(STID)) + I2.dia: M(M(COURSE, I5_TID))
Full text index: I6.dia: M(WORT, M(STID)) + I7.dia: M(WORT, I6_TID)
HVALUE=BLOCKNO
Assume we want a student with a STID with a hash value with 200 students. Then the students are sorted and we have direct access via mini directory to the student.
INSERT STUDENT: All TID's in reserved area have slot number 1. Insertions are realized only in blocks with this corresponding HASH VALUE. If the ITM-records are greater than a block then the new segments are inserted after all reserved blocks with hash value.

6 Metadata

The metadata of all DIA-file files have the same scheme, which is described in this section. They will be stored in one complex record. 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, will be not considered. 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, it is in an A-page.
Scheme of metadata:
COMMENT, TUPCNT, GREATEST_KEY, LANGUAGE,

KEYCNT1, L(TAG, TYPE), B(RGNO, L(SUBTUPKEY)), M(INDEXNAME)

COMMENT: user comment to the file
TUPCNT: total number of complex records of the DIA-file file; is needed directly for end-user or for column oriented manipulations
GREATEST_KEY: greatest key of all complex records; needed if the system generates the keys of the DIA-file
LANGUAGE: main language, used in the file
KEYCNT1: count of key fields of the outermost set (we presuppose that the key fields are always the first elementary fields)
L(TAG, TYPE): DTD (Document Type Definition) of the DIA-file file; the first TAG is TABMENT
L(SUBTUPKEY): atomic fields, which compose the key of the repeating group

It is clear that only the topmost collections are counted. Non-proper collections are not considered. For bags we take all atomic fields and for lists the sequence number.

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

SUBTUPKEY	RGNO	
-	1	A1, A2
POS	2	L(B1)
C1,C2	3	M(C1, C2,…)
E2	4	B(E2)
F1	5	MCT(F1,…)

Collections, which are stored with complex tags, have to be marked by the user by MCT, BCT, or LCT, where CT stands for ”Complex Tag”.
INDEXNAME: name of a DIA-file file, which contains addresses to the given file; indexes to this index are not included. These names are contained in the index file to which it refers.
A DIA-file file begins always with a file header at the beginning of the first page. This header is of type:
VERSION, PAGESIZE, PAGECNT, ENCODING, FIRST_F30, FIRST_F70, FIRST_F100, FIRST_FREE
It does not belong to the metadata. Therefore, it cannot be transferred if the metadata record grows too much.

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 are 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.

As remarked above, the metadata record has always the address “<P>1</P><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 is the first address except the metadata record address. We will set its address to “<P>1</P><S>2</S>”. If the metadata record or the primary data record is larger than a page, the first page has to contain a rudiment to the real position of the complex record at the position “<P>1</P><S>2</S>”.
It should be possible that queries are directed to metadata, too.
Example of a metadata query:
Look for all TEXT-fields of the file “rivers.dia”

```
        aus meta(“rivers.dia”)          # not yet implemented
        mit TYPE=”TEXT”
        gib M(TAG)
```

7 Extensions and Improvements for the Future

Firstly, we will store the complex records like small XML files, possibly 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 is of variable length.

7.1 DIA-file Files with Nesting of Collections Deeper than 2

The storage of DIA-file records as small XML files allows also a generalization of DIA-file tables. We will not require in all cases a collection depth of two. We will for example allow a file of type

PUPILS.dia: M(NAME, FIRSTNAME, FATHER, MOTHER, CLASS,
M(SUBJECT, TEACHER, L(MARK)))

as a DIA-file file, too, because the records of this file are not expected to grow too much.

7.2 Complex Tags

If a proper collection is at deepest level (then it contains no proper collection), we will save storage area by omitting the deepest tags. In the above file, PUPILS.dia, only L(MARK) is of this type. A subtuple now for example looks as follows:

```
<SUBJECT>Maths</SUBJECT>
<TEACHER>Klose</TEACHER>
<MARK>1</MARK>
<MARK>1</MARK>
<MARK>3</MARK>
```

<MARK>2</MARK>
<MARK>2</MARK>
<MARK>1</MARK>
<MARK>3</MARK>

This can now be abbreviated in the following way:

<SUBJECT>Maths</SUBJECT>
<TEACHER>Klose</TEACHER>
<L(MARK)>1;1;3;2;2;1;3</L(MARK)>

If we consider for example the file STUDENT.dia, then a student can be stored in the following way:

<STID>1234</STID>
<NAME>Streich</NAME>
<FIRSTNAME>Joachim</FIRSTNAME>
<FAC>Sport</FAC>
<REGISTER>1971</REGISTER>
<LOC>Magdeburg</LOC>
<SCHOLARSHIP>1850</SCHOLARSHIP>
<M(COURSE, MARK)>Mathe,2;Fussball,1;Laufen,2</M(COURSE,MARK)>
<L(HOBBY)>FOOTBALL;CHESS;READING</L(HOBBY)>

A deepest collection with optional values can be stored in the same way. Non-existing values are represented by empty strings:

<L(A1,A2?)>a11,a21; a12,;a13,a23</L(A1,A2?)>

We will not use collection tags for deepest collection which contain the choice symbol:

<A1>a1</A1><B1>b1</B1><A1>a2</A1><A1>a3</A1>

cannot be abbreviated by

<L(A1 | B1>a1;b1;a2;a3</L(A1 | B1>,

because the inner data cannot be interpreted.

The index I2.idx.dia: M(M(STID, STUDENT_TID)) can also to be stored with the above convention with a minimum of tag data:

<M(STID,P,S)>1111,12,1;1113,17,5;1222,60,6;…</M(STID,P,S)>

Here, the metadata contains the information that “P,S” is the type of STUDENT_TID.

7.3 Cutting Tuples at Arbitrary Positions

It is easier to implement, if a tuple is cut only between complete subtuples. But then a subtuple (and therefore also a field) cannot be larger than a page. This restriction can be avoided, if a corresponding new section is addressed with the same entry as the preceding one but preceded with a “+”symbol in front of the repeating group number.

7.4 Representing Numbers by a Sixty Digit Base

The numbers for pages and slots are represented in a first version by variable length numbers over the alphabet {0, 1, 2, …, 9}. Because of this convention a file is not restricted to a given maximum number of pages and the number of entries (small records and sections of large and very large records) in pages is not restricted to a maximum number. But a number needs a relatively large number of bytes. This can be avoided by enlarging the alphabet. We could take an alphabet (set of digits) [A=0, B=1,…, z=57, {=58, |=59]. Then indexes and pages can carry more information and we save storage area and disk accesses. This can be considered as a special case of data compression.

7.5 Full Text Indexes

Files with long text data need full text indexes.

DOCS.dia: M(DOCID, TITLE, L(AUTHOR), ABSTRACT, …, DOC)
…
We can define a full text index **I13.idx.dia** by the following "gib-aus-mit"-construct (SELECT-FROM-WHERE-construct):

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

Once more we need an index to this index:
I14.idx.dia: M(M(WORT, I13_TID))
If we are looking now for all documents, which contain the words "Hadmersleben" and "Magdeburg", we can find across I14.idx.dia the complex records with keys "Hadmersleben" and "Magdeburg". The corresponding DOCS_TID-lists of I13.idx.dia 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 [12]). 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 DIA-file table, too.

7.6 DIA-file Files without Key of the Outmost Set

There exist situations, where the complex records have no user key. That could mean that the topmost collection symbol has to be a bag or a list? This we will not allow. We will force the system to generate corresponding keys. This can be reached by storing the largest key number in the metadata of the DIA-file file. So the system can choose a new key value, if we want to insert a new record.

7.7 Data Compression

"High level data compression" occurs very often in our proposed data structure. In a file of type *M(STID, NAME, M(COURSE, MARK)) STID*- and *NAME*-data appear only once for each student, contrary to the relation of type *M(STID, NAME, COURSE, MARK)*. In the same way *I6.idx.dia* and *I7.idx.dia* contain *STID1* only twice, compared to *I2.idx.dia,* where each *STID1* value may occur 100 times or more. Further, representing numbers by words over a 60 letter alphabet (section 7.4) is a special kind of data compression. In the same way by complex tags (section 7.2) a lot of tag data space is saved.

We believe that further data compression should be restricted to light weight data compression techniques. It should be applied to small complex records or sections of large and very large records only. But, this will not be considered in a first version.

8 A Short View to Query Optimization with DIA-file Files for OttoQL

Because of space limitations we consider only few examples of OttoQL-programs with the corresponding translated OttoQL-programs on the student file with an index. We plan firstly only query optimization techniques at DIA-file and XML file level. Because indexes are DIA-file-files, too, this can be considered as unification of logical and physical (OttoQL) level.

8.1 Row-Oriented Optimization

STUDENT.dia: M(<u>STID</u>, NAME, FAC, LOC, SCHOLARSHIP,
M(COURSE, MARK), L(HOBBY))
I2.idx.dia: M(M(<u>STID</u>, STUDENT_TID))

Program 1: Find the student with number 1234!

```
        aus doc("STUDENT.dia")
        mit STID="12345678"
has to be transferred to
        aus ("I2.idx.dia" value_subtid << STID::"12345678">>)
        gib M(STUDENT_TID)        # gib: similar to restruct from [1]
        replace STUDENT_TID by ("STUDENT.dia"  value_tid STUDENT_TID)
```

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

```
        aus doc("STUDENT.dia")
        mit STID in L["1234"; "5678"; "9101"]
has to be transferred to
        aus doc("I2.idx.dia")
        mit STID in L["1234"; "5678"; "9101"]                # sequential scan
        gib M(STUDENT_TID)
```

```
    replace STUDENT_TID by (“STUDENT.dia” value_tid STUDENT_TID)
or
    aus (“I2.idx.dia” value_subtid << STID::”1234”>>)
    gib M(STUDENT_TID)
    insert (“I2.idx.dia” value_subtid << STID::”5678”>>)
    insert (“I2.idx.dia” value_subtid << STID::”9101”>>)
    replace STUDENT_TID by (“STUDENT.dia”  value_tid STUDENT_TID)
```

Program 3: Give all computer science students from Magdeburg!

```
    aus     doc(“STUDENT.dia“)
    mit     LOC=”Magdeburg”
    mit     FAC=”computer science”
```

Firstly, we assume that no FAC-index exists, but a two-level LOC-index:
LOC1.idx.dia: M(M(<u>LOC</u>, LOC2_TID))
LOC2.idx.dia: M(<u>LOC</u>, M(STUDENT_TID))

```
    aus     (“LOC1.idx.dia” value_subtid <<LOC::”Magdeburg”>>
    gib     M(LOC2_TID)
    ext     (“LOC2.idx.dia” value_tid LOC2_TID) at LOC2_TID
    gib     M(STUDENT_TID)
    replace STUDENT_TID by (“STUDENT.dia” value_tid STUDENT_TID)
    mit     FAC=”computer science”
```

If we additionally have 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.

8.2 Column-Oriented Optimization

We refer to the column database of section 5.2.

Program 4: Find the sum of scholarships for each register of faculty maths.

```
aus doc(“STUDENT.dia”)
mit FAC=”maths”
gib M(REGISTER, SU) SU:=sum(SCHOLARSCHIP)
```

Internal program:

```
1a)
    aus doc(“STUDENT_FAC.col.dia”)
    mit FAC=“maths“
    gib L(STUDENT_POS)
    =: $p1
1b)
    aus doc(“STUDENT_REGISTER.col.dia“)
```

```
        mit REGISTER: STUDENT_POS in $p1
        ext { aus doc(”STUDENTSCHOLARSHIP.col.dia“)
              mit STUDENT_POS=STUDENT_POS‘
              gib SCHOLARSHIP} at REGISTER
        gib M(REGISTER, SU) SU:=sum(SCHOLARSHIP)
```

or
1b2)

```
        aus doc(“STUDENT_REGISTER.col.dia”)
        , doc(“STUDENT_SCHOLARSHIP.col.dia“)
        mit STUDENT_POS in $p1 # selects in both collections
        gib M(STUDENT_POS, REGISTER?, SCHOLARSHIP?)
        gib M(REGISTER, SU) SU:= sum(SCHOLARSHIP)
```

Program 5: Find the average of marks of each register of faculty maths.

```
        aus doc(“STUDENT.dia”)
        mit FAC=”maths”
        gib M(REGISTER, AVER) AVER:=avg(MARK)
```

Internal program:

```
        aus doc(“STUDENT_FAC.col.dia”)
        mit FAC=“maths“
        gib L(STUDENT_POS)
        =:$p1

        aus doc(“STUDENT_REGISTER.col.dia“), doc(”STUDENT_PAGENO.col.dia“
        , doc(“STUDENT_SLOTNO.col.dia“)
        mit STUDENT_POS in $p1 # selects in all three files
        gib M(STUDENT_POS, PAGENO?, SLOTNO?, REGISTER?)
        ext (“STUDENT_i.dia” value_tid (PAGENO, SLOTNO) at REGISTER?
        gib M(REGISTER, AVER) AVER:=avg(MARK)
```

9 Related Work

The storage structures of AIM/P and DASDBMS [6], [7], [10] are very similar to each other. A collection is stored with the help of a pointer array. Contrary to our approach mini-directories are also used in the case of a small complex record. Therefore, especially small subtuples need additional storage area for pointers. A pointer to a segment in general consists 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 generally not deeply structured and much smaller than the records the developers from AIM/P and DASDBMS had in mind. Our records are forced to have a key contrary to AIM/P and DASDBMS.

In [8] four types of indexes are presented. Value and text indexes can be directly represented by DIA-file structures as described above. We do not need link indexes, because the access to a subtuple always goes over the parent (head) row. We shall consider path indexes in future work.

In [9] a general method for storing and updating XML data is described. In this paper 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 already inserted in design phase of the database. That means we split a deeply structured document into several "tables", such that each record will be sufficiently small. If we introduce corresponding keys, then also the order of the documents can be maintained.

10 Summary

1. The described storage structure for DIA-file files contains with a minimal set of concepts components of sequential (XML files), relational, hierarchical and network (object-oriented) and column-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 (TIDs) 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, while they still might be useful for computer scientists. Each DIA-file 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.
3. Subtuples are stored sorted. Therefore, we have direct access to subtuples with a help of a mini-directory. 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 two types of addresses (TIDs for row-oriented, position numbers for column-oriented manipulation, and subtuplekeys for direct access to subtuples). Because of the ordering of subtuples we only need 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. The column-oriented storage together with complex tags allows to save large amounts of tag data.

5. (External) indexes (full text indexes included) can also be implemented by DIA-file files or by repeating groups in other files. Therefore, all functions, which will be implemented for primary DIA-file files, can be applied to indexes.
6. Metadata do not require a separate file concept. They are stored as a structured tuple of fixed type with address “<P>1</P><S>1</S>”. In other implementations all metadata of a database could be stored in one or more specialized DIA-file files.

Acknowledgement
I thank Andreas Lübcke 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, A.Hauptmann, D. Schamschurko, M. Schnabel, Internet server for *OttoQL*: http://otto.cs.uni-magdeburg.de/otto/web/index.html

[5] 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

[6] 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

[7] 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

[8] 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

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

[10] 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

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

[12] 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

[13] 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

[14] 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

[15] E. C. Sivaji, “State of the art in Column Oriented Database Management System”, Master Thesis University Magdeburg June 2010, Supervisor G. Saake, A. Lübcke

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