Schema-less XML in Columns

Zuzana Částková[1] **Jaroslav Pokorný**[2]

[1] e-Fractal s.r.o.
Vinohradská 174, Praha 3, Czech Republic
castkova.zuzka@gmail.com

[2] Faculty of Mathematics and Physics
Charles University in Prague
Malostranské nám. 25, Praha 1, Czech Republic
pokorny@ksi.mff.cuni.cz

Abstract

C-store environment uses a relational database for storing table tuples on the disk by columns. Can it be effectively used as XML database? This paper considers XML data without a schema. A two-level model of C-store based on XML-enabled relational databases is proposed. A measure of the model suitability is the possibility of evaluating effectively XPath queries. The XPath fragment considered allows the node-test not referring to attribute values and text values. Child, descendant, parent, ancestor, siblings, and following (preceding) are just the XPath axes used here. Low level memory system enabling the estimation of the number of two abstract operations providing an interface to an external memory is characteristic for algorithms for each axis. We will show that our algorithms are mostly of logarithmic complexity in n, where n is the number of nodes of XML tree associated with a XML document.

Keywords: XML, XPath axes, relational database, column stores, C-store

1 Introduction

Column-oriented databases (column stores) are an attractive area both for practice and research in the past few years. The values for each single column (or attribute) are stored contiguously, compressed and densely packed in this architecture. The DBMS C-store introduced in (Stonebraker, et al 2007) is a particular approach to column stores.

Among original motivations for use of column stores belong read-intensive analytical processing workloads, such as those encountered in data warehouses and OLAP. As mentioned by Abadi, Madden, and Hachem (2008) even show that various attempts with row-store converge to performance that is significantly slower on a recently proposed data warehouse benchmark. A more advanced application of C-store is its variant SW-store for storing RDF data as describe authors of (Abadi, et al, 2009). Abadi in (Abadi 2007) explores other more general applications of C-store like so called tables with wide schemas and tables with sparse attributes, i.e. tables with many `NULL` values. Considering XML-enabled DBMSs, approaches that use rather universal table approach belong also to this category. For a detailed discussion of column-oriented database systems see, e.g., tutorial by Abadi, Boncz, and Harizopoulos (2009). In Částková (2009) the author examines a use of column stores for storing and processing XML data.

Our goal is not only to store XML data in column store but also to evaluate effectively an appropriate set of XML queries. Choosing the XPath language means to consider evaluation of XPath axes, i.e. relationship types in which a current node is associated to other nodes in the XML tree. We consider child, descendant, parent, ancestor, siblings, and following (preceding) axes in the paper. We will show that our algorithms are mostly of logarithmic complexity in n, where n is the number of nodes of XML tree associated with a XML document. This improves results published by Bojanczyk and Parys (2008) providing $O(2^{|\varphi|}*n)$ complexity, where $|\varphi|$ is the size of query φ. Their queries deal with attribute values and only checking attribute values for equality.

As most known column store architecture we have chosen the C-store for our research.

In the paper we will consider XML data without schema. In Section 2 we recall some basics of C-store. Section 3 describes a low-level memory system forming an abstract level to real disk memory. In Section 4 we propose a two-level model of C-store based on XML-enabled relational databases. We use a simple structure-centred mapping for storing XML data into relational database and a combination of two well-known numbering schemes: foreign key and depth first. Section 5 is devoted to design and analysis of algorithms for processing XPath axes. Section 6 concludes the paper.

2 C-store: overview

According to work (Stonebraker, et al 2007) we can imagine the C-store approach in two logical levels: user-oriented logical tables and so called projections. Projections are collections of columns each sorted on some attribute(s). We denote this fact in the relational schema by a delimiter and a list of sort attributes. For example, `R(A, B, C | B)` means that projection `R` is sorted on `B`.

One column can occur in more projections. Consequently, a redundancy occurs in C-store database which means that any compression seems to be appropriate for columns. At a logical level C-store supports standard tables equipped by the primary key and, possibly, a set of foreign keys. A C-store projection is *anchored* on a given logical table and retains all duplicate rows, i.e. it has the same number of rows as its anchor table.

In fact, C-store physically stores projections, i.e. C-store considers a logical table as a set of materialized views. The columns of projections are stored column-wise, using separate storage for each column. Because of potentially different ordering of projections it is necessary to ensure a reconstruction of rows of a logical table of C-store. For this purpose, C-store considers row numbers of projections and *mapping tables* (*join indices*). Row numbers are not stored in projections, they are calculated as needed. The join index `T1>>T2(row_number)` is one column table modelling one-to-one mapping between projections `T1` and `T2` anchored on the same logical table `C`. For example, suppose the 3rd row of `T1>>T2` contains the number 6. It means that the 6th row of `T2` and 3rd row of `T1` belong to the same row of `C`. It is well-known, that join indexes are expansive to store and maintain in the presence of updates. Fortunately, there is a lot of rather static XML data which narrows the use of C-store.

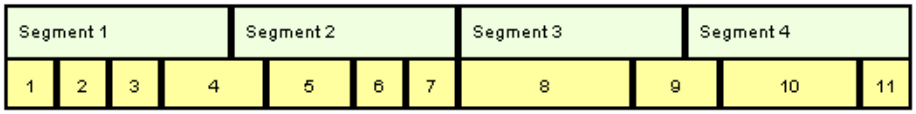

Figure 1: Segments and positions of items

3 Low-level Memory System

We will consider a linear tape consisting from particular positions as model of classical disk memory. A *position* is addressable and can contain a data item. Besides positions we will also consider *segments* of fixed size. Each segment can contain more data items. Each data item can share more segments (see Figure 1). We will present a mutual correspondence between a position on the disk and a segment on the tape. Let C be the number of disk cylinders, S be the number of sectors on one disk track and H be the number of disk heads. Further let $c \in \{0,\ldots, C\text{-}1\}$, $s \in \{0,\ldots, S\text{-}1\}$ and $h \in \{0,\ldots, H\text{-}1\}$. Then the triple $<c, s, h>$ uniquely determines a position on the disk. The associated segment is determined as follows:

$$segment_{<c,s,h>} = c*(H*S) + s*H + h.$$

We have introduced segments as intermediate stage between physical disk operations and operation over the tape with positions.

Imagine now that the read head is moving over the tape. It can read the content from its current position. Let i be arbitrary position on the tape and let j, k be positions different from i such that $|j\text{-}i| > 1$ and $|k\text{-}i| = 1$. If i is the current position on the tape, we call the reading the position j a *jump*, whereas reading the position k we consider as reading a neighbour. Let i be a current position on the tape. For comparison of time complexity of particular algorithms we will use the following operations:

`jump(j)` – reading the random position j from the tape,

`next()` – reading the next (or the previous) position ($i \pm 1$) on the tape.

`next()` operation has time demands dependent on a current position. In fact, a position is non-addressable on disk. Each position holds a concrete value of one attribute of one table row. Implementation of mapping positions on segments influences the exaction time of `next()` operation and, mainly, of `jump()`operation.

We will consider `next()` and `jump()` as basic operations for estimating time complexity of particular algorithms. Now we introduce a set of basic operations that we will be used in all these algorithms:

`SetProjection(P)` – setting up a projection (or a mapping table) `P` as current (up to the next setting all operations go on over data in this projection).

`SetCol(C)` – setting up the `C` column (in current projection) as current.

`Search(X)` – finding the first occurrence of `X` item in current projection and column. At the same time, the record, where the item is found, is set up as current. The function requires several executions of `jump()`or `next()` operation dependent on the current projection is primarily ordered current column or not.

`Read(I)` – reading Ith record in current projection and column. At the same time, Ith record is set up as current. This function corresponds exactly to one execution of `jump()` operation.

`Next()` – reading the next record in current column. At the same time the next record is set up as current. This operation agrees with execution of operation `next()`.

`Previous()` – reading the previous record in current attribute. At the same time, the previous record is set up as current. Complexity of this operation is the same as of execution of `next()` operation.

`ReadCol(C)` – reading value in `C` column (in the same projection) of the current record.

Two mapping algorithms used in the next sections will be evaluated based on how the associated data models are optimal for evaluation of an axis query. We will restrict only on retrieval of all nodes belonging to the chosen axis of a context node. Algorithms searching nodes of particular axes will be completed by a calculation of their time complexity on the level of abstract disk operations `next()` and `jump()`.

4 XML Data in C-store

First, we chose a method how to represent XML data in a relational database. It is clear that two aspects of the design have to be considered:

- a logical database schema enabling the reconstruction of the XML data structure
- a physical database schema used for storing XML data in C-store

We will use only one logical table. The physical tables are projections of the logical table and anchored on it.

There are a lot of methods concerning the first issue as it is shown by Mlýnkova and Pokorný (2005). For example, in the work (Boncz et al, 2006) we can find a schema based on coding preorder and postorder traversal in one logical table. The basic logical table contains attribute `Pre` containing node's preorder rank, `Size` containing the number of nodes in the subtree below the node, and `level` storing the distance from the tree's root. In combination with so called *staircase join* this approach can significantly support XPath processing, particularly in R-store environment. Here we use the approach described in work (Kuckelberg and Krieger 2003), where a mapping of XML document into a relational database by other structure-centred mapping is described. For this method, of the main importance is the knowledge of the list of child nodes for each node from XML tree. This is in accordance with an intuition that a support for path reconstruction should be at disposal. Particularly in C-store, we will ensure that representations of these nodes will be close to each other a column. There are several approaches, how to implement this list.

We use combination of two methods for storing data into C-store:

FK (*Foreign Key Method*) – This method uses a unique identifier for each node of XML tree and a foreign key reference to its parent (see Figure 2).

DF (*Depth First Method*) – This method uses traversing XML tree in a depth first manner. It stores to each node of XML tree a couple of values (*min*, *max*), that are assigned by traversing tree in a depth first manner. Suppose a counter which is increased each time another node is visited. After entering a node *U* in first time its *min* value is set to the current counter value, after last leaving the *U* node the current counter value sets the *max* value of *U*. Counter is increased each step by one (see Figure 3). If *U* is an arbitrary node, we denote its *min* and *max* values as *min*(*U*) and *max*(*U*), respectively.

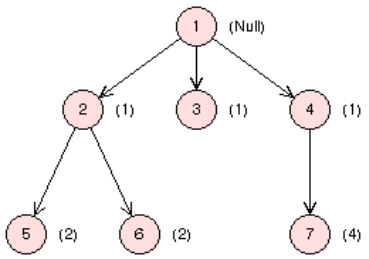

Figure 2: Node identifiers and foreign keys

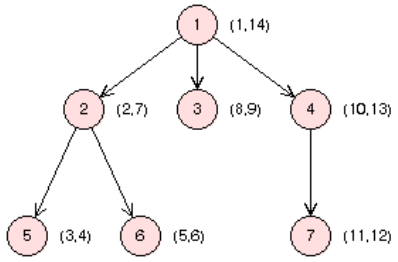

Figure 3: Pre/post-order of tree nodes

We store the nodes of XML tree into the logical table

```
Node(ID, Type, Name, Value, Parent,
     Min, Max, Min_of_Parent),
```

where `ID` (node identifier) is the primary key, `Type` determines the node type (element, attribute, text), `Name` its name (only for elements and attributes, for text data the value is set to `NULL`), `Value` determines value node (for nodes of type attribute or text, for elements the value is set to `NULL`). Attribute `Parent` is a foreign key determining the parent node, `Min` and `Max` contain *min* and *max* values, respectively, of DF method, `Min_of_Parent` is the *min* value assigned to the parent node.

Clearly, the logical table is redundant to some measure. It implements the list of children in a tree in two ways.

Now we introduce a physical data model, over which we will work with C-store. An important part of the design of the physical tables (projections) is specification of ordering of their rows. It will used for algorithms computing particular axes. XML tree is mapped into the following projections:

```
FK1(ID, Type, Name, Value, Parent,
    Min| Parent, Min)
DF(ID, Type, Name, Value, Min, Max,
   Min_of_Parent| Min)
FK2(ID, Parent| ID)
```

Observe that `FK1` is sorted by `Parent` and secondary by `Min`. Further, we use mapping tables `FK2>>DF` and `FK2>>FK1`.

5 XML Axes Querying

In this section we describe algorithms for computing axes for an XML tree mapped into relations `FK1`, `DF`, and `FK2`. For each algorithm we provide an analysis of its time complexity. Let *n* be the number all nodes of XML tree and *id* be the identifier of the current (context) node *N*. *m* we will denote the cardinality of the result set, i.e. the cardinality of axis of *N*. We will also work with value sequences. So, we introduce one important notion for situations when some values lie in the sequence consecutively. Let $A = a_1,...,a_n$ be a sequence. Let *M* be a subsequence of *A*. Then *M* is a *contiguous subsequence* of *A*, iff there is $i \leq n$ such that $\{a_i,...a_{i+|M|-1}\} = M$.

5.1 Child

To retrieve all children of *N* we use sequential reading of one attribute that is trivial for C-store. We want to retrieve all nodes, whose parent is *N*. We use projection `FK1`, which is sorted on attribute `Parent`. Due to ordering we easily find the first occurrence (row number) of searched child and all other occurrences are in contiguous subsequence behind it. By reading the corresponding sequence of `ID` values we obtain the query result.

Algorithm 5.1 (Child axis)
Input: *id* of the input node *N*
Output: the set *D* containing all children of *N*

```
1  D = {}
2  SetProjection(FK1); //Finding
                  corresponding records
3  SetCol(Parent);
4  r = id; start = Search(id);
5  while r == id:
6  (r,stop) = Next();
7  stop = stop - 1;
8  SetCol(ID); //Relational projection
                  on ID attribute
9  D.add(Read(start));
10 while start < stop:
11 (d,start) = Next();
12 D.add(d);
```

Algorithm complexity. Finding the first child requires one `Search()` in column Parent with complexity *O(log n)*. Finding the rest of children takes *O(m)* `Next()` operations.

5.2 Descendant

For finding the set of all descendants of *N*, FK approach would lead to recursive calling of relational operation join. Therefore we use DF approach and its mathematical properties. We are searching for the set of all nodes, which have been visited during navigation XML tree in depth first manner later than the current node *N*, but they were left earlier than from the *N*. Thus, we have the conditions

$$min(N) < min(P) \quad (1)$$

$$max(N) > max(P), \quad (2)$$

where *P* is any descendant of *N*.

We use projection `FK2` along with mapping table `FK2>>DF` for quick finding the position of *N* in projection `DF`. Since this projection is sorted on attribute `Min`, all descendants of *N* appear behind the position. Finally, by the attribute `Max`, we have to distinguish real descendants of *N* from nodes, which fulfil only condition (1) (nodes in the following axis).

Statement 1: Let all nodes of XML tree be stored in an array *A* ordered by value `Min` of DF method. Denote by *pos*(*U*) the position of the node *U* in *A*. Consider the node *N*. Then the following holds for all nodes *P* such that *pos*(*P*) > *pos*(*N*):

If *max*(*P*)>*max*(*N*), then there is no node *P1* such that *pos*(*P1*) > *pos*(*P*) and *P1* is a descendant of *N*.

Proof: Consider the order of nodes in array *A*. Traversing XML tree in a depth first manner values min and max of particular nodes have been assigned in this order:

> *min* value of *N*, successively, all *min* and *max* values of all its descendants, *min* value of *N*, *min* and *max* values of nodes in the following axis for *N*.

Nodes in array *A* are arranged by the min value of DF method, i.e. in the order:

> *N*, all *N* descendants, nodes in the following axis for *N*.

The first node *P* fulfilling the conditions of Statement 1 is obviously the node in the following axis of *N* (it fulfills (1) and does not fulfil (2)). Hence each node *P1* appearing in array *A* behind node *P* is also a node in the following axis and, consequently, can not be its descendant.

A consequence of Statement 1 is, that although the tuples in projection DF are not ordered by the `Max` attribute, we need not to test the `Max` values of all nodes with minimum greater than `Min` of *N*. Namely, once we retrieve the first node in the following axis, we can stop the searching, since we most certainly will find no node on the child axis. Finding the first node, which is not a descendant of *N*, depends on the moment, when (2) stops to hold (even in by maximum non-ordered array).

Algorithm 5.2 (Descendant axis)
Input: *id* of the input node *N*
Output: the set *P* containing all descendants of *N*

```
0  P = {}
1  SetProjection(FK2); //Finding N node
2  SetCol(ID);
3  start = Search(id);
4  SetProjection(FK2>>DF); //Finding
   its position in DF through mapping
   table
5  SetCol(position);
6  start = Read(start);
7  SetProjection(DF); //Finding all
   descentants as sequence start...stop
8  SetCol(Max);
9  Max = Read(start); m = Max;
10 while m <= Max:
11 (m,stop) = Next();
12 stop = stop -1;
13 SetCol(ID); //Relational projection
   on ID attribute
14 Read(start); //No storing – start is
   position of N
15 while start < stop:
16  (p,start) = Next();
17 P.add(p);
```

Algorithm complexity. Finding the *N* node requires one `Search()` execution with complexity *O(log n)*, since it is performed on `ID` in `FK2` projection, hence on an ordered sequence. Thus, the time complexity of this function is *O(log n)*. After that *N* was found in `FK2` projection and consecutively through a mapping table in DF projection, retrieval of all its descendants requires *O(m)* `Next()`operations.

5.3 Parent

Algorithm for finding the parent of the node *N* is trivial. It uses the only `FK2` projection. There, with logarithmical complexity, the position of the context node is found and the value of `Parent` attribute on the associated position. However, this approach uses no property of column-wise storing; conversely a use of row store would entail smaller number of disk operations.

Algorithm 5.3 (Parent axis)
Input: *id* of the input node *N*
Output: *id_r* of the parent of *N*

```
0  SetProjection(FK2); //Finding N node
1  SetCol(ID);
2  Search(id);
3  id_r=ReadCol(Parent); //Reading
                         value in Parent
```

Algorithm complexity. Time complexity of function `Search()` requires *O(log n)* `jump()`operations. The use `ReadCol()`operation instead of `Read()` enables a query optimization natively more suitable for row-stores.

5.4 Ancestor

Storing tables column-wise is appropriate for certain types of queries. The optimal way, how to approach data stored in such way is to read one column and based on its values to decide about query result. As was apparent from the previous algorithms, beneficial is when the result members appear in a contiguous sequence. In this case the time complexity of the algorithm is the same as the time complexity finding the start of this sequence and passing just the set of resulted nodes. During reading the sequence no additional steps are necessary.

Statement 2. There is no algorithm mapping XML tree nodes on such sequence *P*, that for arbitrary node the set of all its ancestors is a contiguous subsequence of *P* and, moreover, each node of XML tree appears just once in *P*.

Proof: Proof is trivial. Assume that there exists such algorithm. Then it should map the tree in Figure 2 on a node sequence *P* in such way, that {1,2}, {1,3}, {1,4} are contiguous subsequences of *P* and in the same time node 1 appears in P just once. Such sequence does not exist.

A consequence of Statement 2 is that during searching all ancestors of *N* in XML tree it is not possible to exploit advantage of sequential reading (not even using any other method implementing the list of following nodes than DF or FK).

Now, we describe two algorithms finding all ancestors the context node. The first one is based on the FK method, the second one on the DF method (Sections 5.4.1-2). Then we will discuss advantages and disadvantages of both algorithms (Section 5.4.3). Finally, we describe an algorithm, which combines both approaches and reaches better results (Section 5.4.4).

5.4.1 Algorithm FK

Remind that both FK and DF methods describe a way, how to implement for a tree structure the lists of childern. FK method is based on knowledge of the parent of each node. In the previous sections, every time when we use FK method, we used attributes `ID` (as the primary key) and `Parent` (as the foreign key). But FK method can be also viewed in other way. We use attribute `Min` as the primary key and `Min_Parent` as a foreign key. Then we use projection `DF`, instead of projection `FK1` or `FK2`. Obviously, all principles of FK method will be preserved:

- `Min` attribute is really the unique node identifier,
- `Min_Parent` attribute contains values of `Min` attribute or `NULL` (if the node has no parent), so it is really a foreign key.

Moreover, `DF` projection is ordered by the `Min` attribute. Consequently, for finding all ancestors of *N* the use of FK method over projection `FK2` is equivalent to the use of FK method over projection `DF`. One iteration of searching the parent of *N*, its grandparent, etc., means in both cases searching in sorted column (`ID` or `Min`, respectively) and reading the value, which searched in the next iteration (`Parent` or `Min_Parent`, respectively). The algorithm uses the projection `FK2` and the mapping table `FK2>>DF` for finding *N* in projection `DF`. Then it uses above described FK method over projection `DF`. The preference of projection `DF` over projection `FK2` serves as the basis to for easy integration of the algorithm with Algorithm 5.5 into a combined algorithm (see Section 4.4.4).

Algorithm 5.4 (Ancestor axis – FK method)
Input: *id* of the input node *N*
Output: the set *P* containing all ancestors of *N*

```
0  P = {}
1  SetProjection(FK2); //Finding N node
2  SetCol(ID);
3  pos = Search(id);
4  SetProjection(FK2>>DF); //Jump to
                projection DF
5  SetCol(position);
6  pos = Read(pos);
7  SetProjection(DF);
8  SetCol(Min_Parent);
9  min_r = Read(pos);
10 while min_r is not NULL: //Finding
                parent, grantpater,…
11 SetCol(Min);
12 pos = Search(min_r);
13 min_r = ReadCol(Min_Parent);
14 if min_r is not NULL:
15 P.add(ReadCol(ID));
```

Algorithm complexity. First, the algorithm retrieves the node *N* in projection `FK2` by function `Search()`. First two jumps are performed (for approaching `Min_Parent` column of `DF` projection). The algorithm is then recursive. In each recursion level, we first find out the given node by `Search()`operation in sorted `Min` column of `DF` projection. Then we read in `Min_Parent` column of the same projection the minimum of parent of this node. This minimum becomes the „current“ in the next level of recursion. Finding the nearest ancestor takes *O(log n)* executions of operation `jump()`. This process repeats *m*-times.

5.4.2 Algorithm DF

This algorithm uses properties of sequential reading. Sequential reading of current column can be done by two

ways – with the `Next()` or the `Previous()`. For our purposes we consider both approaches as identical. In practice, `Previous()`operation can be easily transformed on `Next()`operation by creating the projection ordered in the reverse order. To be consistent with Algorithm 5.6 we use the `Previous()`operation in this algorithm. But the algorithm uses projection `FK2` as well as the mapping table `FK2 >> DF` for finding *N* node and projection `DF` retrieval of its all ancestors.

Algorithm 5.5 (Ancestor axis – DF method)
Input: *id* of the input node *N*
Output: the set *P* containing all ancestors of *N*

```
0  P = {}
1  SetProjection(FK2); //Finding N node
2  SetCol(ID);
3  stop = Search(id);
4  SetProjection(FK2>>DF); //Finding
          its position in DF through
          mapping table
5  SetCol(position);
6  stop = Read(stop);
7  SetProjection(DF);  // Reading
                maximum of N node
8  SetCol(Max);
9  Max = Read(stop);
10 i = stop;     // Testing all nodes
          with Min value less than Min
          of N node on Max condition
11 while i > 1:
12 (m,i) = Previous();
13 if m > Max:
14 P_pos.add(i);
15 SetCol(ID);  //now P_pos contains a
      list of positions, we need list
      of IDs
16 i = stop; id = Read(i);
17 while i > 1:
18 (id,i) = Previous();
19 if i in P_pos:
20 P.add(id);
```

Algorithm complexity. Finding *N*, i.e. executing `Search()` function in attribute `ID` of projection `FK2`, has complexity *O(log n)* jumps on disk. After finding *N* we have first to execute *O*(1) jumps on disk to reach the `Max` column of `DF` projection and to read `Max` value of *N*. Finally, the algorithm reads at most *n* members in a contiguous sequence, which means *O(n)* executions of operation `next()`.

5.4.3 Comparison of FK and DF algorithms

We can see that both algorithms for finding node *N* have the same time complexity. Thus, we will compare complexity of finding the set of ancestors of *N*.

Algorithm 5.4 consists of *O(m*log n)* jumps on disk. An advantage is that the algorithm reads no superfluous data. A disadvantage is that items read are not in a contiguous sequence, i.e. jumps are needed. On the other hand, Algorithm 5.5 reads more items (including that ones not belonging to the result set), but its advantage is, that these items are stored in a contiguous sequence. The time complexity of such reading is *O(n)*.

Suitability of particular algorithms can be judged by two factors – „density“ of sequence read by DF method and technical parameters of disk. *Sequence density* determines how many data in sequence is relevant for the query result. The higher the sequence density is, the more usable DF method is. Decreasing the density means that DF method is less and less effective up to certain moment (dependent on technical parameters of disk) when *m* jumps will take less time than processing *n* `next()`operations.

Also of importance is that density of sequence read by DF method can be different in different parts of the sequence. Knowledge of the densities where nodes have minimum less than *N* leads to idea to combine approaches FK and DF for finding all ancestors of *N*. In sequence parts with low density Algorithm 5.4 is used, for parts with high density we use Algorithm 5.5. Of course, the knowledge of densities has to be obtainable without additional reading the sequence. In such case, the algorithm effectiveness could be worse than that one of Algorithm 5.5 itself. In the next section we describe the combined algorithm in detail including implementation of decision, which of two approaches should be used.

5.4.4 Combined algorithm – „Jump and go“

The name of algorithm reflects the movement that the algorithm reminds – jumping alternated with walking step by step.

Algorithm is based on recursive findings the parent of given item, grandparent, etc. It decides dynamically in each iteration, whether it will search out the parent of current item by binary search (`Search()` operation) over sorted `Min` column (jumping), or via reading sequence of several items appearing in projection `DF` immediately before the current node (walking).

Now we introduce a metric for measuring the density of a given sequence part. We can observe that the sequence part on which we decide contains nodes appearing in projection `DF` between the parent of current node (including) and the current node (out of it). Thus, this sequence contains several (≥0) nodes not relevant for the query and the only one node (parent current node) relevant for the query. The number of non-relevant nodes is essential for the sequence part desnsity. It corresponds exactly to the number of nodes appearing in projection DF between the current node and its parent.

Statement 3. Let all nodes of XML tree be stored in an array *A* sorted by the *min* value of DF method. Let *U* be a node and *R* its parent. Denote by *p* the number of nodes appearing in array *A* between nodes *R* and *U*. Then

$$p = \frac{\min(U) - \min(R) - 1}{2} \qquad (3)$$

Proof: During traversing XML tree in a depth first manner *min* and *max* values of particular nodes were assigned in this order:

> *min* value of *R*, successively all *min* and *max* values of all nodes appearing between nodes *R* and *U*, *min* value of *U*.

Simultaneously, the counter of DF method was increased by 2×*p* between assigning *min*(*R*) and *min*(*U*). Thus,

$$min(R) + 2 \times p + 1 = min(U) \qquad (4)$$

Then (4) immediately implies the equality (3).

A consequence of Statement 3 is, that for setting the density of the nearest actual sequence we need to know only `Min` value the current node and `Min` value of its parent. Both these values are stored in projection `DF`, so we need not to read the sequence and know its density in unit time.

To be able to chose either jumping or walking for a given sequence part, we have to compare their time demands. Jumping works similarly as Algorithm 5.4. Assuming, that we just read the values of `Min` and `Min_Parent` attributes for current node, we use only its corresponding part – finding position, on which its parent appears. Then a new iteration follows, i.e. the decision process, which approach will be applied.

Walking is based on Algorithm 5.5. First, we read the value of `Max` attribute of the current node and then values of `Max` attribute for all nodes, appearing between the current node and its parent (these nodes have the `Max` value less than the current node). The parent is recognized, when the value of `Max` attribute is greater than `Max` value of the current node. In this moment we stop the searching. Again a new iteration the algorithm follows.

Assume the current node U, its minimum $min(U)$ and minimum for its parent $min\_r(U)$. Finding the parent node U by jumping requires

$$\log(n) \times t_s \tag{5}$$

time, where t_s is average time needed by a jump on disk. In practice, we can the jumping method even improve in such way, that we search out only in those part of DF projection before the current item (see Section 5.4.2). The number n would be lower in each step. Finding the parent node U method need (see Statement 3)

$$\frac{min(U) - min(R) - 1}{2} + 1 \times t_n \tag{6}$$

where t_n is average time operation `next()`. The numbers t_s and t_n are constant for the disk used, n is constant pro each XML file. Comparing values (5) and (6) we can determine which approach is for sequence actual more appropriate:

- if value (5) is less, then we apply the jumping method,
- if value (6) is less, then we apply the walking method.

Finally, we introduce a formal description of the complete algorithm „Jump and go“.

Algorithm 5.6 (Jump and go)
Input: *id* of the input node *N*
Output: the set *P* containing all ancestors of *N*

```
0  P = {}
1  SetProjection(FK2); //Finding N node
2  SetCol(ID);
3  pos = Search(id);
4  SetProjection(FK2>>DF); //Jump to
                  projection DF
5  SetCol(position);
6  pos = Read(pos);
7  SetProjection(DF);
8  SetCol(Min_Parent);
9  min_r = Read(pos);
10 while min_r is not NULL:   //Finding
             parent, grantpater,…
             +Decision what to chose
11 min = ReadCol(Min);
12 if log(konst.n)*konst.ts <=((min-
   min_r-1)/2 +1)*konst.tn:
13 SetCol(Min);               //Jumping
14 pos = Search(min_r);
15 P.add(ReadCol(ID));
16 else:
17 m = max = ReadCol(Max);    //Walking
18 SetCol(Max);
19 while pos > 1 and m <= max:
20 (m,pos) = Previous();
21 if m > max:
22 P.add(ReadCol(ID));
23 SetCol(Min_Parent);
24 min_r = Read(pos);
```

5.5 Sibling

For finding all siblings of the node N we again use sequential reading of one attribute. We use projection `FK2`, in which we approach the value of `Parent` attribute for N. Siblings are nodes, which have this value the same. Using mapping table `FK2>>FK1` we find the occurrence of N in `FK1`. Due to the secondary ordering the records by attribute `Min`, we know that the sequence of items with the same value of `Parent` attribute appearing before the given node in projection `FK1` corresponds to its younger siblings and the sequence behind the item corresponds to its older siblings.

Algorithm 5.7 (Sibling axis)
Input: *id* of the input node *N*
Output: the sets *M* and *S* containing all younger siblings and older siblings of *N*, respectively

```
0  M = {}, S = {}
1  SetProjection(FK2); //Finding N node
2  SetCol(ID);
3  start_stop = Search(id);
4  SetProjection(FK2>>FK1); //trough
                  join-index
5  SetCol(position);
6  start_stop = Read(start_stop);
7  SetProjection(FK1); // FK1
8  SetCol(Parent);
9  parent = Read(start_stop);
10 r = parent; //Finding set M =
                  [start, ...start_stop]
11 while r == parent:
12 (r,start) = Previous();
13 start = start + 1;
14 r = parent; Read(start_stop);
   //Finding set S = [start_stop,
   …,stop]
15 while r == parent:
16 (r,stop) = Next();
17 stop = stop - 1;
18 SetCol(ID);  //Relational projection
                  on ID attribute
19 i = Read(start);
20 while start < stop:
21 if start < start_stop:
22 M.add(i);
23 if start > start_stop:
24 S.add(i);
25 (i,start) = Next();
```

Algorithm complexity. Operation `Search()` has complexity *O(log n)*. After finding N in projection `FK1`

the time complexity of selection of all siblings (operations `Next()/Previous()`) is *O(m)*.

Remark: Algorithm 5.7 has found all siblings of *N* as a union of all younger and older siblings. A trivial modification of the algorithm enables to restrict its result set only to younger or older siblings without changing the time complexity of the algorithm.

5.6 Following (Preceding)

Let *U* be a node of XML tree. While for arbitrary node *V*

$$min(V) < min(U), \quad (7)$$

holds, then *V* is either ancestor or preceding node of *U*. The difference between ancestor and preceding node is in value *max*. Whereas the preceding nodes have *max* value less than the node *U*, the ancestors have *max* value greater than node *U*. To be preceding the next condition (8) has to hold together with the condition (7) for the node V:

$$max(V) < max(U). \quad (8)$$

Otherwise, if for arbitrary node *V*

$$min(V) > min(U) \quad (9)$$

and

$$max(V) > \max(U), \quad (10)$$

then node *V* follows node *U*.

Notice that finding the result set *M* leads to calculation of a set difference. Let `Min_inv` be a set of nodes that fulfil the minimum invariant, i.e. the condition (7) for preceding axis and the condition (9) for the following axis. Further, let *Max_not* be the set of nodes that do not fulfil the maximum invariant (condition (8) and (10), respectively). Thus, *Max_not* is the set of ancestors (as concerns searching preceding) or of descendant (as concerns searching following). The query result is then the set

$$M = Min\_inv \setminus Max\_not.$$

Now we introduce a universal algorithm for searching both preceding and following nodes. The algorithm uses `DF` projection for finding the result set. It also uses projection `FK2` and mapping table `FK2 >>DF` for finding the node *N*. The algorithm finds all preceding nodes or all following nodes, depending on which couple of operations `Init` and `MaxInvariant` is used.

Preceding	Following
`Init (position) return(1, position)`	`Init (position) return (position+1, N+1)`
`MaxInvariant (max) return max < MAX`	`MaxInvariant (max) return max > MAX`

Operation `Init()` implements minimum invariant, i.e. it returns couple (*start*, *stop*) in projection `FK1`, *MAX* denotes the `Max` value of *N*.

Algorithm 5.8 (Following, preceding axes)
Input: *id* of the input node *N*
Output: the set *M* containing all preceding nodes (respectively following nodes) of *N*

```
0  SetProjection(FK2); //Finding N node
1  SetCol(ID);
2  position = Search(id);
3  SetProjection(FK2>>DF); //trough
                join-index
4  SetCol(position);
5  position = Read(position);
6  SetProjection(DF);  //Finding set of
                items not belonging to
                the result
7  SetCol(Max);
8  MAX = Read(position);
9  (start, stop) = Init(position);
   Max_not = {}; i = start;
10 m = Read(i);
11 while i < stop:
12 if not MaxInvariant(m):
13 Max_not.add(i);
14  (m,i)Next();
15 SetCol(ID); //Finding the result set
16 i = start; id = Read(i);
17 while i < stop:
18 if i not in Max_not:
19 M.add(id);
20 (id,i) = Next();
```

Algorithm complexity. Let *p* be the number of nodes in the *Max_not* set. The retrieval of *N* node needs *O(log n)* jumps on disk. For obtaining the *M* set itself, the algorithm takes yet 2*(*m*+*p*+1) executions of `Next()` function.

6 Conclusions and future work

We can conclude with the following statement summarizing complexities of algorithms described in Section 5.

Statement 3. Let *n* be the number of nodes of XML tree, *m* be the number of nodes in an axis, and let *p* be the number of nodes not fulfilling the maximum invariant specified for following (preceding) axis. Axes queries over C-store modelled by relations specified in Section 4 have the complexities given in Table 1.

Analyzing these *O* – expressions from proofs described in Section 4, we can observe that the functions behind the formal complexities have mostly the constants *a* and *b* in leading terms *a*m* or *b**log *n*, respectively, equal to 1.

Of course, as it is mentioned by Částková (2009), the algorithms can be further optimized, particularly those ones for following (preceding) axes. Also it is possible to propose a different logical as well as physical model for relations implementing C-store. Another direction of future research is a use of indexes supporting queries over C-store. For example, authors of (Sidirourgos et al, 2008) show that with proper clustered indices the triple-store for RDF data performs better than the vertically-partitioned approach.

Analyzing these *O* – expressions from proofs described in Section 4, we can observe that the functions behind the formal complexities have mostly the constants *a* and *b* in leading terms *a*m* or *b**log *n*, respectively, equal to 1.

An important part missing in our paper concerns experiments with real XML data. Theoretical time complexities should be confirmed by real complexities gained with a help a column-oriented DBMS, e.g. C-store. Clearly, it should be done in accordance with our low-level memory system which seems to be sufficiently general and, consequently, usable for describing memory

Axis	#occurrences of `next()`	#occurrences of `jump()`
child	*O(m)*	*O(log n)*
descendant	*O(m)*	*O(log n)*
parent	0	*O(log n)*
ancestor - FK method	0	*O(m*log n)*
ancestor - DF method	*O(n)*	*O(log n)*
sibling	*O(m)*	*O(log n)*
following	*O(m+p)*	*O(log n)*
preceding	*O(m+p)*	*O(log n)*

Table 1: Complexity of algorithms for XPath axes

system in C-store. On the other, real column-oriented DBMSs can have a different physical data model than the one used in our research. For example, MonetDB provides the model based on the technique described in (Boncz et al, 2006) and mentioned in Section 4.

Another direction comes from the situation when we have an XML schema for XML data. First observations discussed by Částková (2009) show that knowledge of such schema gives no significant contribution to the design of physical schema design for C-store.

Aknowledgement This research has been partially supported by the grant of GACR No. P202/10/0573.

References

Abadi, D.J. (2007): Column Stores for Wide and Sparse Data. Proc. of 3rd Biennial Conference on Innovative Data Systems Research (CIDR), Asilomar, California, USA, 292-297, www.crdrdb.org

Abadi, D.J., Markus, M., Madden, S., and Hollenbach, K. (2009): SW-Store: a vertically partitioned DBMS for Semantic Web data management. *VLDB J.* **18**(2):385-406

Abadi, D.J., Madden, S., and Hachem, N. (2008): Column-stores vs. row-stores: how different are they really? *Proc. of ACM SIGMOD International Conference on Management of Data*, Vancouver, Canada, 967-980, ACM Press.

Abadi, D.J., Boncz, P.A., and Harizopoulos, S. (2009): Column-oriented Database Systems. *Proc. of VLDB Conference*, Volume 2 of the Journal "Proceedings of the VLDB Endowment", Lyon, France, 1664–1665.

Bojanczyk, M. and Parys, P. (2008): XPath evaluation in linear time. In: *Proc. of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (PODS), Vancouver, BC, Canada. 241–250, ACM Press.

Boncz, P., Grust, T., van Keulen, M., Manegold, S., Rittinger, J. and Teubner, J. (2006): MonetDB/XQuery: a fast XQuery processor powered by a relational engine. *Proc. of the 2006 ACM SIGMOD International Conference on Management of Data*, Chicago, IL, USA, 479-490.

C-store (2010): A Column-Oriented DBMS. http://db.csail.mit.edu/projects/cstore/. Accessed 1 Nov 2010.

Částková, Z. (2009): XML data representation with the help of C-store. Diploma thesis. Faculty of Mathematics and Physics, Charles University in Prague, Czech Republic. In Czech.

Kuckelberg, A. and Krieger, R. (2003): Efficient Structure Oriented Storage of XML Documents using ORDBMS. *Lecture Notes in Computer Science, Volume 2590/2008*, 131–143, Springer-Verlag Heidelberg.

Mlýnková, I. and Pokorný, J. (2005): XML in the World of (Object-)Relational Database Systems. In *Information Systems Development Advances in Theory, Practice and Education*. Vasilecas, O., Caplinskas, A., Wojtowski, G., Wojtowski, W. and Zupancic, S (eds.), Springer Science+Business Media, Inc.

MonetDB (2010): http://monetdb.cwi.nl/ Accessed 1 Nov 2010.

Sidirourgos, L., Goncalves, R., Kersten, M., Nes, N., Manegold, M. (2008): Column-store support for RDF data management: not all swans are white. *Proc. of VLDB Endow.*, Vol. 1, No. 2, 1553-1563, ACM.

Stonebraker, M., Abadi, D.J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E.J., O'Neil, P.E., Rasin, A., Tran, N., and Zdonik, S.B. (2007): C-Store: a column-oriented DBMS. *Proc. of VLDB Conference*, Vienna, Austria, 553–564, ACM.