# Cost-Sensitive Reordering of Navigational Primitives

Carl-Christian Kanne Matthias Brantner Guido Moerkotte
Universität Mannheim
Mannheim, Germany
cc|msb|moer@pi3.informatik.uni-mannheim.de

## ABSTRACT

We present a method to evaluate path queries based on the novel concept of partial path instances. Our method (1) maximizes performance by means of sequential scans or asynchronous I/O, (2) does not require a special storage format, (3) relies on simple navigational primitives on trees, and (4) can be complemented by existing logical and physical optimizations such as duplicate elimination, duplicate prevention and path rewriting.

We use a physical algebra which separates those navigation operations that require I/O from those that do not. All I/O operations necessary for the evaluation of a path are isolated in a single operator, which may employ efficient I/O scheduling strategies such as sequential scans or asynchronous I/O.

Performance results for queries from the XMark benchmark show that reordering the navigation operations can increase performance up to a factor of four.

## 1. INTRODUCTION

XPath[7] is a cornerstone of XML query processing, both as a stand-alone query language and as a building block of the standard languages XQuery [2] and XSLT [6]. There is no hope for the construction of efficient XML query evaluation engines if path expressions cannot be evaluated efficiently, so it is no surprise that research about the efficient evaluation of XPath queries proliferates (e.g. [3, 4, 5, 10, 11, 13, 14, 18, 20, 24]). It is surprising, however, that most approaches only consider the logical level. They minimize the number of logical steps that have to be processed to evaluate a path expression, but neglect the fact that the steps may have vastly differing physical costs, especially if the data volumes grow large. Hence, a lot of optimization potential is lost, as much of the actual processing time is spent in the physical system layer, performing I/O requests and data representation changes.

In this paper, we present an efficient method for XPath evaluation that reorders navigational primitives based on their physical cost.

**Example 1** As a simple motivating example, consider a flat document structure consisting of a root node `a` and child nodes `b-g` (Fig. 1). The nodes are stored on a hard disk with 4 pages, numbered 0–3. A straightforward evaluation of the simple XPath query


*SIGMOD 2005* June 14-16, 2005, Baltimore, Maryland, USA.


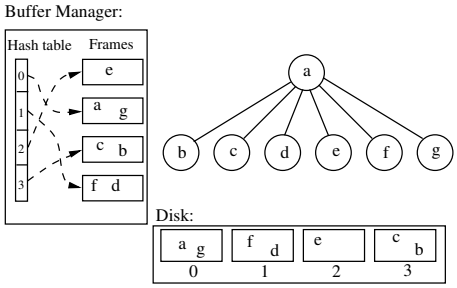


**Figure 1: Document tree access**

`descendant-or-self::X` just iterates over all nodes, selecting those with tag `X`. This looks like an efficient access plan until we consider the evaluation process in the DBMS in more detail. To access the nodes, the data has to be loaded into a main memory buffer from disk. This is supervised by the buffer manager in Fig. 1, which caches loaded pages for future reference.

In our example query, we start with node `a`, causing the buffer manager to load disk page 0. Then we proceed to `b`, loading page 3, having to seek all the way to the end of our disk. Traversing `a`'s children `b-g` like this, we end up accessing the pages in the order 0,3,1,2. Hence, we incur two disk seeks of distance 2 and one of distance 1 on a document spanning just 4 pages. On large documents, these random I/O patterns will dominate evaluation costs.

For another physical cost aspect, compare the navigation from node `b` to `c` with the navigation from `e` to `f`. The target disk page is already in the buffer in both cases, but `e` is on a different page than `f`, while `b` is on the same page as `c`. The buffer manager needs to access its hash table to find the main memory location of `f`'s page 1. This requires multi-thread synchronization in a typical DBMS. In comparison to that, the intra-page navigation for `b` and `c` is much cheaper. □

Note that a certain distribution of nodes on the disk pages may have a variety of reasons; a document import algorithm might regroup nodes to avoid wasting space, and incremental updates may fragment the physical layout. Even the concept of the "physical disk page" itself is relative, as we find more levels of indirection further down in the system. These include: mapping of DBMS page addresses onto operating system (OS) file pages, translation of OS file pages to logical (sic!) block addresses (LBAs), and resolution of LBAs by device drivers and disk controllers to physical locations on the disk platter, taking drive geometry and bad blocks into consideration. A precise determination of the resulting access latency times is difficult, or not even possible, for the DBMS.

Before introducing our method, we want to state the require-

ments for XPath evaluation in a real-world DBMS. We feel that this is necessary, because requirements which are considered a matter of course for traditional query evaluation engines are often overlooked in the context of XML. It is our conviction that a successful method for XPath evaluation in an industrial-strength DBMS must

1. be able to employ efficient physical access methods, such as sequential scans or asynchronous I/O.
2. be applicable on a wide range of efficient and updatable storage formats that support synchronization and recovery.
3. be implementable using proven, efficient design patterns such as *iterators* [12].
4. be interoperable with orthogonal logical optimization techniques.

In an assessment of related work with respect to these requirements (Sec. 2), we come to the conclusion that none of the existing approaches is adequate.

Our new XPath evaluation technique was built with the requirements mentioned above in mind. Our contributions can be summarized as follows:

1. We show how fast access methods like scans and asynchronous I/O can be utilized in an iterator-based physical path evaluation algebra, using existing, simple but flexible storage models.
2. We introduce the notion of *partial path instances* to represent incomplete computations that cannot be finished "yet" due to pending I/O.
3. We introduce physical operators that can process such partial path instances to evaluate paths.
4. We provide experimental results to demonstrate the performance gains achievable through our method.

Our method is based on a simple model which takes the physical costs of navigational primitives into account. The different cost factors, such as I/O and representation changes, are generalized by considering *clusters* of nodes which have cheap intra-cluster navigation costs. Our algorithms then minimize evaluation costs with several techniques:

We pool all expensive I/O requests for a path in a single operator. This I/O operator can efficiently process the requests by reordering their execution, and/or by forwarding many of them at once to the lower system layers, where even more beneficial decisions about their order can be made — this goes down to the level of the disk itself, where protocols like SCSI allow to manage several pending commands at once. An orthogonal feature of our method is the reduction of the number of times a cluster has to be visited, by performing as many navigation operations as possible during each visit of that cluster. This technique may be complemented by *speculative* evaluation of candidate path fragments to guarantee that each disk page is only visited exactly once.

The paper is structured as follows. In Sec. 2, we use the requirements stated above to assess related work. Sec. 3 reviews our system model, defining terms and stating the assumptions we make about storage and query execution. Sec. 4 introduces *partial path instances*, the first class citizens processed by our physical algebra. This algebra is covered in Sec. 5. Sec. 6 contains performance measurements of different paths and plan alternatives. Sec. 7 concludes the paper.

# 2. RELATED WORK

Most of the existing approaches to XPath evaluation deal with queries and documents only on a logical level (e.g. [3, 10, 11, 13, 14, 20]). and do not consider the physical placement of the data in secondary storage, or costs incurred by representation changes and main-memory access locality.

There is some related work that investigates ways to avoid costly random accesses during XPath evaluation [4, 5, 18, 24]. The common principle is to convert documents to a storage layout that allows for query execution plans that rely exclusively on physically sequential scans, or leaf scans in tree structures. The sequential scan is an access pattern that is typically detected by the lower system layers, which will make proper I/O scheduling decisions, avoiding many of the pitfalls related to physical layout mentioned in the introduction.

However, the proposed formats are not easily updated, as they use preorder numbers to identify nodes, or require the nodes to be stored in a particular order. Both of these properties are difficult to maintain during updates. Some of the approaches are also biased towards certain queries, e.g. [18] requires time linear in the size of the data even for very selective queries. It is also unclear how the special formats can be supported by the remaining components of an industrial-strength DBMS, such as synchronization and recovery.

Access support relation approaches [17, 25] precompute and materialize certain paths to allow sequential access patterns. Here, particular paths have to be selected a priori, and obviously only queries containing these paths are supported. Maintenance of the support relations is costly.

In addition, if an execution plan based on these scan-based methods requires several scans in a single query execution plan, for example because several location path expressions occur, or if several queries run in parallel, there may be interference effects that again result in additional disk arm movement.

An inspiration for our work is the Assembly operator [15], which efficiently loads complex objects into main memory for further processing. Such an operator could be used in XML query evaluation, treating documents as complex objects. However, in the case of path evaluation, we are interested only in selected nodes reachable by a certain path, and do not want to load and keep the whole document in main memory. Further, I/O scheduling is done by the Assembly operator itself, whereas our approach may delegate scheduling decisions to the system layers that are better informed about physical details, such as the operating system, drivers and on-disk controllers. The authors of [15] explicitly warn that multiple concurrently active Assembly operators (in the same or different execution plans) would have detrimental effects on performance. Our method avoids these negative effects. The Assembly operator also treats the traversal of all object references in the same way, whereas we distinguish between expensive and inexpensive traversals for further performance gains. The Assembly operator may also visit disk pages several times, whereas our operators may speculatively process objects to avoid repeated I/O. Another advantage of our method is that we allow arbitrary physical algebra expressions to specify component iterators, opening our approach to orthogonal optimization techniques.

# 3. SYSTEM MODEL

In this section, we discuss our assumptions about how XML documents are stored and processed. We have two goals. First, we want to define the family of systems in which the techniques presented in this paper can be applied. Second, we explain which fea-

tures of a DBMS we can exploit to improve performance, if they are present.

Our assumptions are not demanding. In fact, most existing DBMSs conform to the model if a simple tree schema is used. More sophisticated XML storage schemes are also covered by our model.

## 3.1 Logical Tree Model

We describe XML documents using a labeled, ordered tree model, where nodes are labeled with tags taken from a tag alphabet $\Sigma$. For brevity reasons, we do not discuss XML node types such as text nodes, attributes, namespace nodes, and entities. They can be incorporated into our path evaluation method without difficulty.

## 3.2 Covered Tree Storage Models

We assume that every document is stored as a single labeled tree, but do not require a particular storage format on the individual nodes. Our method is applicable on formats that use single records for every node [21], formats that combine several nodes into tuples based on schema information [8, 23], formats that cluster subtrees as long as they fit on a single page [9], formats that store complete documents stored in large objects (LOBs), and any combination of these formats.

However, we require that individual nodes can be identified using a NodeID, which may be logical or physical, as long as there is a means to deduce the physical location of the node from the NodeID.

**Example 2** If each node is stored in a separate physical record, a typical form of a NodeID in an actual storage engine is the *record ID*, or RID, consisting of a page number and a slot number on the page. If several nodes share a record, then the NodeID may comprise a RID together with an intra-record index, e.g. as an attribute number or an offset. □

## 3.3 Clustering

Typically, the physical nodes are *clustered* in some way. A *cluster* is a set of nodes that are stored physically close to each other, such that navigation between nodes in the same cluster is much cheaper than navigation between nodes in different clusters.

There may be several levels of clustering, for example one level where connected nodes of a subtree are combined in a single physical record, and an additional level on which these physical records are stored on the same disk page.

One of the clustering levels is the unit of I/O. In most DBMSs, this level of clustering is the disk page, as whole pages are transferred between the page buffer and the disk.

We assume that the cluster(s) a node belongs to can be determined from its NodeID[1]. For example, a RID contains as one component the physical page number on which a record is stored.

Clustering may be optimized by special update algorithms, but even without intentional clustering, there is often a time-of-creation clustering which causes related nodes to be stored on the same disk pages.

Note that we allow, but do not require the clusters to be aligned with query access patterns, such as XPath axes. While specialized clusterings help performance, they can only match a limited set of access patterns, while penalizing others. Clusterings are also expensive to maintain in the presence of updates.

[1] Systems whose native NodeIDs do not provide cluster information can incorporate our method by implementing extended NodeIDs annotated with clusters.

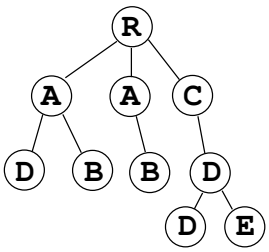


**Figure 2: Sample logical tree**

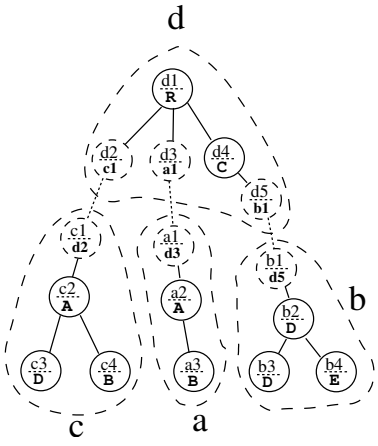


**Figure 3: Corresponding storage layout for Fig. 2**

## 3.4 Border nodes

To allow a simple presentation of our evaluation algorithms, we need to explicitly incorporate the navigation cost between nodes into our storage model. Therefore, we classify the edges of document trees, based on the navigation cost. An edge that links two nodes in the same cluster is called *intra-cluster* edge, while all other edges are *inter-cluster* edges.

In addition to the *core nodes* representing logical document nodes, we introduce *border nodes* to model inter-cluster edges. In our physical storage model, there is one border node at each end of an edge that crosses the border between two clusters. Each border node contains the NodeID of the border node on the opposite side of the border.

We call the set of core nodes $C$, and the set of border nodes $B$. For border nodes $x \in B$, we require an operation $target(x)$ returning its target NodeID.

**Example 3** Fig. 2 shows a logical tree, and Fig. 3 a corresponding clustered physical tree. The clusters `a`–`d` are represented by dashed lines. Each node contains its NodeID above the dashed line. The core nodes have their tag below the dashed line, while the border nodes at the ends of inter-cluster edges have their companion border node's NodeID below the dashed line. □

The border nodes can be used to represent an incomplete edge traversal during navigation of inter-cluster edges (see below).

The concept of explicit border nodes is mainly intended to allow for a succinct description of our physical algebra. Our method is applicable even if no explicit border nodes appear in the storage format. As explained above, we do not require a homogeneous representation even for core nodes.

**Example 4** Consider a simple object-relational schema for la-

beled trees, where each core node is stored in one object with a tag attribute, a reference attribute for the parent core node, and a list-valued attribute containing child references. To describe this schema in our model, we would use a small cluster for each tuple, containing a subtree with the parent reference as root border node, having the core node as child, which, in turn, has the child references as children border nodes. □

## 3.5 Navigation

We require the existence of *navigational primitives* which allow to iterate, based on a given context node, over all nodes along an XPath axis, one node at a time.

These navigational primitives must be able to efficiently return nodes *using intra-cluster navigation only*. This enables our cost-driven evaluation method to control when expensive inter-cluster navigation is performed. The navigation primitives only return intra-cluster results, possibly including border nodes that point to other clusters, where additional result nodes may be located. Access to the targets of these result border nodes can be scheduled based on the target cluster ID.

## 3.6 Swizzling

A DBMS can have one or more types of buffer pool to avoid secondary memory access and/or representation changes [16]. This offers further optimization potential, as follows.

Nodes can be represented in the query engine by using NodeIDs. However, the DBMS may also allow to use direct, or *swizzled* [19], pointers into buffer pools, and supply fast navigational primitives based on these pointers. In this case, there also have to be explicit conversion operations to translate NodeIDs into pointers and vice versa.

Converting a direct pointer to a NodeID, *unswizzling*, is cheap. *Swizzling*, i.e. translating a NodeID to a pointer, is more expensive, as it requires access to the buffer manager, which may require multithread synchronization, and translation table lookups.

Our method can explicitly issue conversion requests, minimizing the amount of expensive swizzling operations and using direct pointers as much as possible.

However, if only NodeIDs can be used as input and output of the navigation primitives, our method works as well, using the identity function for swizzling and unswizzling. While in this case the performance benefit of direct pointers is lost, our method still optimizes inter-cluster navigation costs.

## 3.7 Asynchronous I/O

One way of reducing the I/O cost of inter-cluster edges in our method is asynchronous I/O, which is supported by most current operating systems.

If the DBMS allows to asynchronously process requests to load clusters into buffers, our technique can take advantage of this. The interface we expect allows to issue requests for buffer accesses without waiting for them to complete. A separate call must be available to retrieve completed requests.

Such an interface allows the lower system layers to reorder the I/O requests, minimizing access latency times. At least part of the reordering may be done by the on-disk controller, which has precise knowledge about drive geometry and head positions, further improving performance.

# 4. PATH INSTANCES

This section introduces the concept of *partial path instance*s. This data type can represent incomplete path evaluations, facilitating the separation of cheap intra-cluster navigation from expensive inter-cluster navigation. The physical algebra we present in the next section creates, selects and combines partial path instances to fully evaluate XPath expressions.

## 4.1 Location Paths

Let $\pi$ be a location path expression in XPath. We denote with $|\pi|$ the number of location steps in $\pi$. $\pi_i$ is the $i$th location step ($1 \leq i \leq |\pi|$), $axis_{\pi_i}$ is the step's axis, and $nt_{\pi_i}$ is its node test. In our model, node tests are specified as a subset of the tag alphabet $\Sigma$, describing the node tags that are allowed at this step. This limits our method to a subset of XPath predicate types, but as we will see later, our physical algebra expressions can be incorporated into a more expressive algebra that provides full XPath support.

## 4.2 Full Path Instances

A *full path instance* is a map $p$ from the steps $0, 1, \ldots, |\pi|$ of a location path $\pi$ to the set $C$ of core nodes, mapping each step $i$ to a core node $p_i \in C$. The 0th location step $\pi_0$ represents the context nodes on which to evaluate $\pi$, so that $p_0$ is the context node at which the path instance originates.

Of course, we require for each $0 < i \leq |\pi|$ that $p_i$ is reachable from $p_{i-1}$ using step $\pi_i$.

Let $\pi$ be a location path, $c$ a context node, and $r$ a result node reached by $\pi$ starting from $c$. Then a full path instance $p$ with $p_0 = c$ and $p_{|\pi|} = r$ can be considered a certificate for the fact that $r$ is indeed a result node of $p$, describing how $r$ can be reached from $c$.

## 4.3 Partial Path Instances

A *partial path instance* is a map $p$ representing a fragment of a path instance.

Given the set of border nodes $B$ (see definition in Sec. 3.4), and $\epsilon$ as a null value, $p$ is a map $p : \{0, 1, \ldots, |\pi|\} \longrightarrow C \cup B \cup \{\epsilon\}$ which maps each step number $i$ to $p_i$, and which satisfies the condition

$$\exists l, r : 0 \leq l \leq r \leq |\pi| \text{ such that}$$

$$\forall i : \text{all of} \left\{ \begin{array}{rcl} i < l & \Rightarrow & p_i = \epsilon \\ i = l & \Rightarrow & p_i \in C \cup B \\ l < i < r & \Rightarrow & p_i \in C \\ i = r & \Rightarrow & p_i \in C \cup B \\ i > r & \Rightarrow & p_i = \epsilon \end{array} \right\} \text{hold}$$

Note that $l$ and $r$ are unique for each path instance. Informally, a partial path instance maps only a consecutive subsequence of location steps to document nodes, where the two ends of the subsequence may be incomplete navigations represented as border nodes.

$p$ is said to be *left-incomplete* iff $p_l \in B$, and *left-complete* otherwise. *Right-complete* and *right-incomplete* are the equivalents for the right path end. A path instance that is left- and right-complete is called *complete*. Full path instances are a special case of partial path instances: A path instance $p$ is *full* iff it is complete, and $l = 0 \wedge r = |\pi|$.

**Example 5** Tab. 1 shows some path instances based on the sample tree (Fig. 3), and the location path `/A//B` with context node d1. Note that a path instance can be non-full, but complete.

The path instances represent knowledge about the document tree with respect to the query. This information is used in our operators. For example, path instance 3 can be interpreted as "If d1 is a context node, we can reach a2 after step 1". Path instance 7 says "if we have d1 as a context node, we can reach d3 while processing step 1". Path instance 9 implies "if we can reach a1 while processing step 1, we have a3 as a result node". □

| No | $\pi_0$ Context | $\pi_1$ /A | $\pi_2$ //B | $l$ | $r$ | F | L | R | C |
|---|---|---|---|---|---|---|---|---|---|
| 1 | d1 | $\epsilon$ | $\epsilon$ | 0 | 0 | - | + | + | + |
| 2 | d1 | a2 | $\epsilon$ | 0 | 1 | - | + | + | + |
| 3 | d1 | c2 | $\epsilon$ | 0 | 1 | - | + | + | + |
| 4 | d1 | c2 | c4 | 0 | 2 | + | + | + | + |
| 5 | d1 | a2 | a3 | 0 | 2 | + | + | + | + |
| 6 | d1 | d2 | $\epsilon$ | 0 | 1 | - | + | - | - |
| 7 | d1 | d3 | $\epsilon$ | 0 | 1 | - | + | - | - |
| 8 | c1 | c2 | c4 | 0 | 2 | - | - | + | - |
| 9 | a1 | a2 | a3 | 0 | 2 | - | - | + | - |

F=full, L=left-complete, R=right-complete, C=complete

**Table 1: Path instances for `/A//B` in the tree from Fig. 3**

## 4.4 Path Instance Representation

The definition of a partial path instance as a map is convenient for purposes of understanding and reasoning, but, in fact, it contains more information than actually needed. As we will see, when processing partial path instances in our algebra, the operator algorithms only need the values $l$, $p_l$, $r$, and $p_r$ of each path instance $p$. Hence, we can represent partial path instances as tuples with 4 attributes. We will use a tuple for each partial path instance $p$ with the attributes $S_L(p)$, $N_L(p)$, $S_R(p)$, and $N_R(p)$, where

$$N_L(p) := p_l \quad S_L(p) := l \quad N_R(p) := p_r$$

$$S_R(p) := \begin{cases} r-1 & \text{if } p_r \in B \\ r & \text{otherwise.} \end{cases}$$

The offset of 1 in the step number of right-incomplete path instances simplifies the specification of our operators' algorithms, because this way, we can treat right-incomplete paths as paths where the final step has not been fully evaluated.

For brevity reasons, we also define $end_L(p) := (S_L(p), N_L(p))$ and $end_R(p) := (S_R(p), N_R(p))$.

# 5. PHYSICAL OPERATORS

After explaining our storage model and defining the concept of partial path instances, we now turn to our physical algebra. The operators of our algebra work on sequences of partial path instances and allow for very efficient location path evaluation.

We limit our exposition to the 4 physical operators for location paths. In a typical system, they are part of a more expressive algebra capable of representing access plans for larger subsets of XPath (such as [1, 3]), and more powerful languages, such as XQuery.

Before elaborating on our novel technique, we first review a comparatively simple approach to path evaluation, which in our terminology exclusively processes *complete* path instances. In Sec. 5.3, we introduce operators that can process *right-incomplete* (but left-complete) path instances, using asynchronous I/O to improve performance. Finally, in Sec. 5.4, we show how our algebra can be extended to incorporate also *left-incomplete* path instances. This makes path evaluation susceptible to access plans using a single scan, again reaching a higher level of performance. In Sec. 5.5, we describe how to reestablish document order in the result, since our operators process and deliver nodes in an order that lowers physical evaluation costs, which may be different from document order.

## 5.1 Simple Method

The straightforward approach to evaluate location paths uses nested loops, one for each location step. In addition, final duplicate elimination and sorting may be required.

This is easily expressed as a chain of Unnest-Map [3] operators (one for each location step), where each operator reads a context node from its producer and then enumerates all result nodes for a step. To respect the node-set and order semantics from XPath, final duplication elimination and/or sorting operators [14] are added[2].

In our terminology, such an approach passes non-full, but complete path instances from operator to operator. The original context nodes $n$ are enumerated by a leaf operator as path instances $p$ with $N_L(p) = N_R(p) = n$ and $S_L(p) = S_R(p) = 0$.

Each Unnest-Map operator is then responsible for extending the path instance by one step, increasing $S_R(p)$ by one, and enumerating the step's result nodes in $N_R(p)$. The final Unnest-Map operator has full path instances as a result, to which possibly a duplicate elimination operator is applied.

All the involved path instances are complete because the navigational primitives used in the Unnest-Map operators only consider the logical tree structure, not its physical layout. Hence, many random physical accesses may occur when enumerating the result nodes for each step, as explained in Example 1.

## 5.2 Operator Specifications

All individual operators specified below are iterators [12]. An iterator is defined by its input, parameters, execution state, and algorithms for the open, next and close method used to enumerate a result.

We describe all of our operators according to a common structure. After a brief introduction explaining the role of the operator, we give an overview of its input, parameters and execution state, and an informal description of the output.

Due to space constraints, we cannot give full pseudocode or another complete formal specification of the operators. Instead, we verbally describe the algorithm for each operator's next method in addition to our informal output specification. We omit the open and close methods for brevity reasons, as they only contain some initialization and cleanup of the execution state.

## 5.3 Left-complete path instances

We now show how to translate location paths into physical algebra expressions which yield efficient physical access patterns. We begin with expressions that process only left-complete path instances.

The core principle in our method is to avoid random physical accesses caused by immediate traversals of inter-cluster edges. Only cheap intra-cluster navigation is performed immediately. Inter-cluster accesses are deferred to a later point in time when I/O for the required cluster is less expensive, i.e. because the I/O has been performed asynchronously. We start by giving a brief overview of the new operators (Sec. 5.3.1) and their cooperation, and then go into detail in a separate subsection for each operator.

### *5.3.1 Overview*

Our method is a descendant of the simple nested-loop evaluation explained above. However, we restrict navigation in the Unnest-Map operators to intra-cluster accesses that do not require I/O. The modified Unnest-Map operators are called XStep operators (Sec. 5.3.2).

[2]To avoid exponential complexity[11], additional duplicate elimination may be advised after intermediate results[14]. Still, the straightforward method explained here produces correct results.

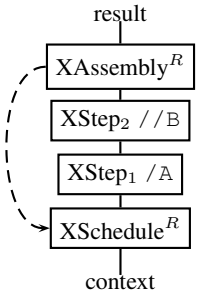


**Figure 4: Plan for /A//B**

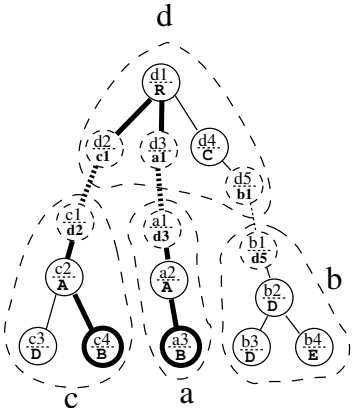


**Figure 5: A clustered tree**

Since the XStep operators do not perform I/O, they sometimes produce right-incomplete path instances. These are consumed by our new XAssembly$^R$operator (Sec. 5.3.3), which identifies new clusters that have to be visited, and forwards them to the XSchedule$^R$ operator.

All necessary I/O for one path is isolated in a single XSchedule$^R$ operator that produces the input of the XStep operator chain, and schedules and manages access to the physical clusters (Sec. 5.3.4).

**Example 6** In Fig. 4, a sample plan for the simple query `/A//B` is shown. The subplan "context" produces the context nodes, and XSchedule$^R$schedules access to the corresponding clusters. The XStep operators then extend the partial path instances by nodes for the two steps. The XAssembly$^R$operator returns full path instances, and forwards any clusters that have to be visited back to XSchedule$^R$.

Fig. 5 shows the clustered document tree from Fig. 3, with the bold nodes **c4** and **a3** representing the query result. The bold lines are the paths used to reach them.

Fig. 6 shows how the plan from Fig. 4 processes the sample tree using left-complete paths. Node d1 is the input context node. The gray area denotes those parts of the document that have been processed after each cluster has been processed.

(I) `d` is accessed by XSchedule$^R$. Then, the XStep operator for `/A` determines that nodes a1 and c1 are potential continuations of the path, but they require inter-cluster edges to be traversed. Hence, XAssembly requests cluster accesses to `a` and `c` from XSchedule$^R$. Node d4 is labeled `C`, failing the node test `A`, so access to cluster `b` is not required. Now, XSchedule$^R$ asynchronously accesses cluster `a` and `c`, to retrieve nodes a1 and c1. Operating system, device drivers and intelligent disks now may decide which cluster can be delivered

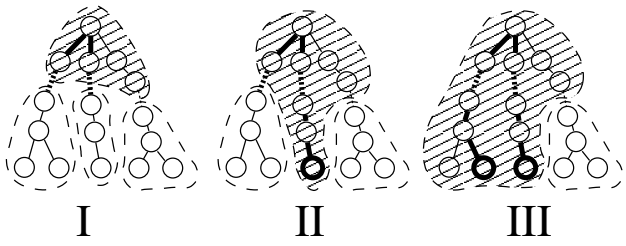


**Figure 6: Path processing with XSchedule$^R$**

fastest. In our example, `a` is accessed first, and is processed while `c` is loaded in the background. (II) Now, XSchedule$^R$ returns node a1, and the XStep operators can, using only intra-cluster edges, return a full path to node a3 to XAssembly$^R$, which delivers it as a result. (III) In the meantime, cluster `c` has been loaded. The second result c4 is returned, analogously to node a3 above. Note that cluster `b` is never visited because d5 was not produced as an XStep result in step (I). □

### 5.3.2 XStep

XStep performs all of the cheap intra-cluster navigation in our plans. An XStep operator tries to extend partial path instances by one step to the right. In the execution plan for a location path $\pi$, there is one XStep operator XStep$_i$ for each location step $\pi_i$.

#### 5.3.2.1 Overview.

**Input** Partial path instances from XStep or XSchedule

**Parameters**

| | |
|---|---|
| $i$ | step number |
| $\pi_i$ | location step to evaluate |

**Output** All input path instances with $S_R(p) \neq i-1$, *and* the result of applying step $\pi_i$, using only intra-cluster navigation, to all input instances with $S_R(p) = i-1$.

**Execution State** Iterator position in step result for current input instance

#### 5.3.2.2 XStep next method.

For each input path instance $p$, the operator XStep$_i$ decides whether to process or to forward $p$, depending on whether $S_R(p) = i-1$.

If $S_R(p) = i-1$, XStep$_i$ is applicable to $p$, and XStep$_i$ tries to extend $p$ by enumerating nodes which can be reached using $\pi_i$.

XStep$_i$ works similar to a regular Unnest-Map in the simple method above. It applies step $\pi_i$ to the path, but only as far as possible without leaving the current cluster. This is done as follows:

XStep$_i$ enumerates all nodes $s$ which can be reached from $p_{i-1}$ using step $\pi_i$. This enumeration is performed by navigating through the document tree according to the axis and the node test of $\pi_i$. During this navigation, the operator traverses edges in the document tree. It distinguishes two cases, one for intra-cluster edges and one for inter-cluster edges.

1. For each $s$ which can be reached by using only intra-cluster edges and which satisifies $nt_{\pi_i}(s)$, an output partial path instance $p'$ is returned which is identical to $p$ except that $p'_i := s$. In this case, the path instance is extended by one step, hence $S_R(p') = S_R(p) + 1 (= i)$.

2. If enumeration of the result nodes of $\pi_i$ comes across a border node $b$, then the associated inter-cluster edge is not traversed, i.e. the target node is not accessed immediately. Instead, an output partial path instance $p'$ is returned, which is identical to $p$, but has as $N_R(p')$ the border node $b$. The paths $S_R(p)$ remains equal to $S_R(p')$ $(= i - 1)$, because the step has not been fully evaluated yet.

   Note that the enumeration of the result nodes $p_i$ for the given $p_{i-1}$ continues after such a right-incomplete path instance is returned. XStep remembers in its execution state which node was delivered last. This value is used by a subsequent next call to continue the navigation through the local cluster. This either generates right-complete partial path instances (case 1) or right-incomplete path instances (case 2).

If the right end of $p$ was not generated by $\pi_{i-1}$, XStep$_i$ is not applicable to this path instance, and $p$ is directly handed over to the operator's consumer. Once an XStep operator sets $N_R$ to a border node, the remaining XStep operators do not process that path instance. Because its $S_R()$ value has not been increased to $i$, it is passed through to the XAssembly$^R$ operator (see below).

#### 5.3.2.3 Swizzling.

If the DBMS in which our method is implemented supports swizzling (Sec. 3.6), we can exploit this for better performance.

This is done by passing only swizzled node representations between the XStep operators. Navigation is more efficient on main-memory pointers, and the XStep operators process path instances only with navigation operations. Hence, when passing path instances from one XStep operator to the next, we can exclusively use the swizzled representation for nodes. There is no need to swizzle and unswizzle repeatedly. Only the nodes in our remaining operators' main-memory structures (see below) need to be represented in an unswizzled form, to avoid a large number of live references that prevent objects from being swapped out of the buffer.

### 5.3.3 XAssembly$^R$

XAssembly$^R$ is the topmost operator in an evaluation plan for a path $\pi$.

The XStep operators sometimes produce incomplete path instances, since they do not perform I/O. Hence, in contrast to the simple method using Unnest-Map operators, the output of the XStep operator chain is a superset of the final result for the path query, as it includes not only the full path instances, but also right-incomplete path instances that need to be extended across inter-cluster edges.

XAssembly$^R$ performs two main tasks: First, it filters the full paths that have been produced by the XStep chain, and returns them to the consumer. Second, when right-incomplete paths have been produced, it notifies the XSchedule$^R$ operator that new clusters have to be visited.

#### 5.3.3.1 Overview.

**Input** Partial path instances from the XStep operator chain

**Parameters**

| | |
|---|---|
| $X$ | reference to the associated XSchedule operator |
| $l$ | path length |

**Output** All full path instances for $\pi$

**Execution State**

| | |
|---|---|
| $R$ | set of right path ends known to be reachable |

#### 5.3.3.2 XAssembly$^R$ next method.

On each next method call, XAssembly$^R$ requests a new path instance $p$ from its producer. $p$ is either a full path instance, if the final XStep operator could generate a core result node for the final step, or a right-incomplete path instance if evaluation had to stop at some step due to an inter-cluster border.

Full path instances are directly returned to the consumer. For right-incomplete path instances, XAssembly$^R$ checks whether $target(end_R(p))$ is already contained in $R$. If not, it is added to $R$, and XAssembly$^R$notifies the XSchedule operator of this new intra-cluster edge to visit. This is done by adding the partial path $(N_L(p), S_L(p), target(N_R(p)), S_R(p))$ to XSchedule$^R$'s queue $Q$. XSchedule$^R$ will schedule access to the target node, and continue processing of the partial path instance when the corresponding cluster has been loaded.

To allow XSchedule$^R$'s $Q$ to be accessed, XAssembly$^R$ is parameterized by a reference to the corresponding XSchedule$^R$ operator.

Note that XAssembly$^R$ is not a pipeline-breaker, i.e. it does not need to materialize intermediate results, but can return results (full path instances) directly as they are produced by the XStep chain.

#### 5.3.3.3 Duplicate elimination.

Several combinations of consecutive location steps may produce duplicates [14]. These duplicates sometimes can be prevented by query rewrites, or dynamic programming/memoization [10, 11], or they can be eliminated after having been created.

The XAssembly$^R$ approach automatically eliminates duplicates on the final result, since the final result nodes are also contained in $R$ and never returned twice.

Partly, duplicates of intermediate steps are also eliminated. Whenever an inter-cluster edge is encountered by the XStep operators, the path instance is forwarded to the XAssembly$^R$ operator, which performs duplicate elimination by means of $R$, so no inter-cluster edge is traversed twice for the same step.

There are additional techniques to avoid duplicates which are orthogonal to our approach and can be incorporated. In the simple Unnest-Map method, it is possible to push duplicate elimination down into the chain of Unnest-Map operators. This works with our XStep operator chain as well. We can improve on the approach, by allowing these duplicate eliminations to access $R$ in the XAssembly$^R$ operator directly. This way, all duplicate eliminations share the same data structure, factorizing some of their overhead.

### 5.3.4 XSchedule$^R$

The XSchedule$^R$ operator is responsible for performing the physical accesses to the queried data.

#### 5.3.4.1 Overview.

**Input** From producer: context nodes $x$ in form of non-full, complete path instances $p$ with $S_L(p) = S_R(p) = 0$ and $N_L(p) = S_L(p) = x$.

From XAssembly: Right-incomplete path instances whose target cluster must be visited

**Parameters**

| | |
|---|---|
| $k$ | desired minimum size of $Q$ |

**Output** The input and all right-incomplete partial path instances forwarded by XAssembly, in a scheduled order that minimizes I/O costs

**Execution State**

- $c$ current cluster
- $Q$ queue of unprocessed partial path instances

#### 5.3.4.2 XSchedule$^R$ next method.

XSchedule$^R$'s next method proceeds in three steps. First, the queue of still unprocessed path instances is maintained, possibly replenishing it from the producer. Second, any cluster accesses in the queue that have not been scheduled yet are submitted to the asynchronous I/O subsystem. Third, the operator returns a result path from a cluster for which I/O has already been completed.

**Queue Processing** The items in $Q$ are either non-full, right-complete path instances from the producer, or right-incomplete path instances added by XAssembly$^R$. The path instances by XAssembly$^R$ contain as right ends the target nodes of inter-cluster edge traversals.

$Q$ is sorted lexicographically by the target cluster ID of $N_R()$ of the contained paths first, and $S_R()$ second.

Whenever a path instance is requested by the consumer, XSchedule$^R$ verifies that the queue $Q$ contains at least $k$ items. If it does not, XSchedule$^R$reads instances from its producer. These input instances are added to the queue. The producer is read until exhausted, or until at least $k$ right ends are in the queue. The reason to set a desired minimum size for $k$ is to have enough scheduling alternatives for the asynchronous I/O subsystems to choose from. Since location paths are typically evaluated on a single context node, the choice of $k$ does not matter much. The default value of $k$ in our system is 100.

**Scheduling Cluster Accesses** Whenever a new right incomplete path $p$ is added to the queue, either by the replenishing step above, or by the XAssembly$^R$ operator, it is checked whether the access to the associated cluster has already been submitted to the asynchronous I/O subsystem. If not, a new asynchronous I/O request is added to the subsystem's queue, requesting access to the cluster of $N_R(p)$.

**Returning a result path** The XSchedule$^R$ operator also maintains a *current cluster* $c$ in its state, which is initially an invalid reference.

If there is no incomplete path in $Q$ that has a right end in $c$, the XSchedule$^R$ operator asks the asynchronous I/O subsystem for a completed I/O request $r$, possibly blocking until one is completed. It sets the current cluster $c$ to the one accessed by $r$.

As the last processing step, XSchedule$^R$ checks whether there exists a path $p$ in $Q$ that has as $N_R(p)$ a node in $c$. If yes, $p$ is removed from $Q$ and returned to the consumer.

Processing ends when $Q$ is empty.

## 5.4 Left-incomplete path instances

The previous section explained how incomplete path instances can represent incomplete computations about a path result. All the path instances were left-complete, i.e. it was known that the right end of the path instance could be reached from some context node. This allowed us to be more flexible about the order in which clusters were visited.

To further improve performance, we have two goals. (1) We want to prevent visiting any cluster more than once. (2) We want to evaluate paths using a single sequential scan.

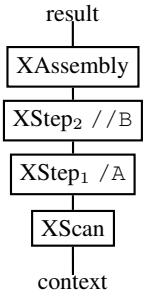


**Figure 7: XScan-based Plan for /A//B**

A consequence of our goals is that during a single visit, we need to extract all information from a cluster that might be relevant for the evaluation of the path. However, we do not want to constrain the order in which the clusters are stored physically. Hence, we might have the situation where it is unknown whether some nodes in the currently processed cluster qualify for the result. This is because the nodes that would have to be examined to decide this are stored behind the current cluster and have not been visited yet.

In this section, we take the notion of incomplete computations one level further and also allow *speculative* evaluation of paths. The result of speculative evaluation is information of the kind "if node $x$ can be reached in step $i$, then we can also reach node $y$ in step $j$". We can store such information in main memory, using left-incomplete path instances (Sec. 4.3). When we later visit node $x$'s cluster, we can then decide whether to return $y$ as a result or not.

Below, we give an overview of what plans that process left-incomplete instances look like, and describe how the operators XStep, XAssembly$^R$ and XSchedule$^R$ need to be modified to accept left-incomplete paths. The general operators are called XAssembly and XSchedule, and they behave in the same way as XAssembly$^R$ and XSchedule$^R$ if no left-incomplete paths are involved.

We also introduce an alternative for XSchedule as I/O-performing operator. The new XScan operator does not use asynchronous I/O, but performs a single sequential scan to access the clusters. The choice of the I/O-performing operator is up to the optimizer, and depends on the selectivity of the query (as we will see in Sec. 6).

Especially when processing left-incomplete paths, our method may sometimes require large main memory structures. In Sec. 5.4.6, we discuss how the evaluation plan can revert to a *fallback mode* in low memory situations. Resources are released, and the remaining results are computed using the less memory-demanding simple method from Sec. 5.1.

### 5.4.1 Overview

Plans for path processing with left-incomplete instances are similar to the ones for right-incomplete path processing. The only difference is that the XSchedule operator may be replaced by an XScan. XScan visits all clusters of a document collection exactly once. In addition to path instances beginning with context nodes, XScan speculatively creates non-full, left-incomplete path instances. The XStep operators extend both kinds of instances by the nodes reachable by intra-cluster edges, and the XAssembly operator finally merges incomplete path instances, creating longer, and, eventually, full path instances.

**Example 7** Fig. 7 shows a plan in which XScan has been chosen as the I/O-performing operator. Note that in contrast to the XSchedule-based plan (Fig. 4), no connection between the I/O-

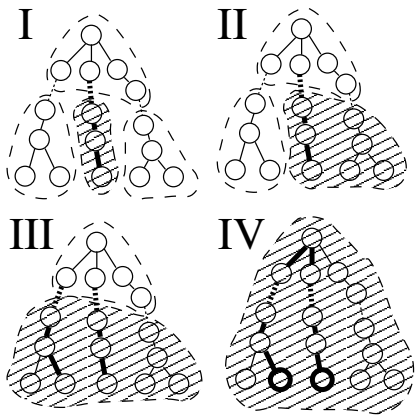


**Figure 8: Path processing with XScan**

performing operator and XAssembly exists, because XScan unconditionally scans all clusters of the document (or document collection).

When executing the query on the same document (Fig. 5) as in our example for the XSchedule plan, the cluster access proceeds as shown in Fig. 8. The clusters' physical order on disk is `a, b, c, d`.

(I) First, cluster `a` is examined. a3 is identified as a candidate result node for the path, but only if the parent cluster `d` contains a proper input context node. Hence, a left-incomplete path instance is created and stored (indicated by bold lines), which will be processed in step IV (see below). (II) Cluster `b` is visited during the scan. However, the XStep operators find no nodes in `b` matching the node tests of our path, so no action is taken. (III) Cluster `c` is visited, and a left-incomplete path instance is generated for node c4, as above. (IV) Finally, the scan arrives at cluster `d`. The XStep operators produces right-incomplete paths leading to the border nodes d2 and d3, pointing to `a` and `c`. XAssembly detects that these right-incomplete paths can be merged with the two left-incomplete paths created earlier, so the result nodes a3 and c4 can be returned. □

### 5.4.2 XStep

The XStep operator remains exactly as for right-incomplete processing above (Sec. 5.3.2). It only extends path instances to the right by one step, and does not care whether the path instance is left-complete or not.

### 5.4.3 XScan

The XScan operator is the second of our two I/O performing operators, complementing the XSchedule operator. Instead of performing dynamic scheduling of cluster accesses, it scans all clusters of a document or document collection.

#### 5.4.3.1 Overview.

**Input** context nodes $x$ in form of non-full, complete path instances $p$ with $S_L(p) = S_R(p) = 0$ and $N_L(p) = S_L(p) = x$.

The input is required to be sorted by the cluster ID of $N_R(p)$.

**Parameters**

| | |
|---|---|
| $D$ | document or collection of documents to scan |
| $l$ | path length |

**Output**

Returns the input, *and*
all partial path instances $\{(b, i, b, i) | b \in B \wedge 0 \leq i < l\}$, obtained by a sequential scan of $D$

**Execution State**

| | |
|---|---|
| $c$ | current cluster |
| $b$ | current border node |
| $i$ | current step number |

#### 5.4.3.2 XScan next method.

The operator sequentially scans the specified clusters, loading them into the buffer.

For each cluster $c$, XScan first returns all path instances from the producer which have right ends in $c$. Since the input context nodes are sorted by cluster ID, all context nodes for $c$ are located consecutively at the producer's current iterator position.

Once all context nodes for a cluster have been returned, XScan produces partial path instances for that cluster. These additional path instances are speculatively produced in order to avoid having to visit the cluster again, as explained in the introduction to Sec. 5.4.

For all border nodes $b$ in the current cluster, XScan generates one left-incomplete path instance $l_{bi}$ for each step $i$ in the path. The $l_{bi}$ are non-full partial path instances with $S_L(l_{bi}) = S_R(l_{bi}) = i$, and $N_R(l_{bi}) = N_L(l_{bi}) = b$.

These left-incomplete paths are processed by the XStep operators, which try to extend them into longer partial paths. If that fails, i.e. for a path $l_{bi}$ the XStep operator for step $i$ cannot enumerate any nodes starting with $b$, then $l_{bi}$ is filtered and not forwarded to XAssembly (see definition of XStep).

### 5.4.4 XSchedule

XSchedule is based on XSchedule$^R$ (Sec. 5.3.4). In the following, we only explain the differences.

The general XSchedule operator may also speculatively evaluate partial paths. Generation of left-incomplete path instances may be required even for paths without a scan, because it avoids multiple visits of the same cluster. Such multiple visits can occur, for example when a full path instance of length 3 is computed which contains a node from one cluster $c_1$ in the first step, then a node from another cluster $c_2$ in the second step, and finally another node from $c_1$ in the third step. In this case, the intra-cluster edge between $c_1$ and $c_2$ is traversed twice, requiring two accesses to $c_1$.

To prevent multiple visits, the specification for the general XSchedule operator includes a flag called `speculative` indicating that left-incomplete paths have to be generated.

When visiting a cluster $c$, XSchedule first returns results for that cluster, in the same way as XSchedule$^R$, but using $Q$ instead of the producer.

It first returns all paths $p$ in $Q$ which have a $N_R(p)$ in $c$, remembering them in $N$. If `speculative` is set, then additional left-incomplete path instances are generated, in the same way as in the XScan operator (Sec. 5.4.3).

### 5.4.5 XAssembly

In the presence of left-incomplete path instancess, XAssembly has additional responsibilities compared with XAssembly$^R$ (see Sec. 5.3.3). We must make sure that we process our speculatively generated path instances when a cluster is visited whose contents may complete some of our left-incomplete path instances.

#### 5.4.5.1 Overview.

**Input** Partial path instances from XStep chain

**Parameters**

| | |
|---|---|
| $X$ | reference of associated XSchedule operator (may be 0 if XScan is used). |
| $l$ | path length |

**Output** Full path instances for $\pi$

**Execution State**

| | |
|---|---|
| $R$ | set of reachable right ends |
| $S$ | set of all discovered left-incomplete path instances which are candidates for full paths |

#### 5.4.5.2 XAssembly next method.

The algorithm for XAssembly's next method call generates full paths, by processing the following two cases in a loop until a full path has been found, or the producer is exhausted and there are no more reachable paths in $S$.

1. If there is a path instance $x \in S$ with $end_L(x) \in R$, this means that we can now also reach $end_R(x)$, because $x$ can be interpreted as "if we can reach $end_L(x)$, we can also reach $end_R(x)$". If $end_R(x) \notin R$, $end_R(x)$ is added to $R$. If $x$ is right-complete and $S_R(x) = |\pi|$, then $x$ is a full path, and can be returned as a result. Finally, $x$ is removed from $S$.

2. If no such $x$ can be found, and if there are no reachable path instances in $S$, a new path instance $y$ is requested from XAssembly's producer (the XStep chain). If $y$ is a full path instance, it is returned. If it is left-incomplete, it is added to $S$. If it is left-complete, but right-incomplete, $end_R(y)$ is added to $R$.

Note that the second case above is very similar to the next method of XAssembly$^R$. The only difference is that left incomplete paths are added to $S$. In the absence of left-incomplete paths, XAssembly behaves like XAssembly$^R$.

#### 5.4.5.3 Notification of XSchedule.

A zero reference for XSchedule may be used in the general XAssembly operator. XAssembly only tries to add items to $Q$ if a nonzero reference is used as a parameter. This allows XAssembly to be used with both XSchedule and XScan operators, as the latter does not have a queue to maintain.

#### 5.4.5.4 Paths beginning with //.

A special optimization is possible if the location path begins with `/descendant-or-self::node()`, or its abbreviation "//". In this case, we know that *all* nodes of a document can be reached after the first step. More formally, we have $\forall x : (x, 1) \in R$.

In this case, we can avoid storing right ends for the first step. Instead, we add code to the containment check for $R$, which always returns true if the step number of the right end is 1.

This reduces memory usage and improves XAssembly performance. However, it only works when we are guaranteed to visit all clusters containing nodes of the document. Hence, we only use this technique when an XScan is used as input for the XStep operator chain.

### 5.4.6 Fallback

The data structure for $S$ may become large, if the selecivity of the queries is low. However, in our method we can limit main memory consumption. If the DBMS's main memory limit for a single query is reached, we switch to a fallback mode. Fallback mode has the advantage that it does not need the main memory for speculatively evaluated path instances (i.e. $S$). Only the main-memory structures for duplicate elimination are required. The disadvantage is that in fallback mode, the physical evaluation costs may become much higher.

In fallback mode, the XStep operators do not stop navigation at cluster borders. Instead, they behave as simple Unnest-Map operators that evaluate the entire result for each context node, no matter whether different clusters are accessed or not. Hence, only complete path instances are generated by the XSteps, and only full path instances leave the XStep chain.

XAssembly discards its $S$ data structure, and behaves merely as a duplicate elimination operator on the result nodes, i.e. it hands all incoming tuples over to the consumer if they are not in $R$ yet.

If XSchedule is used, the only difference to regular operation is that the `speculative` flag is set to `false`, to avoid producing left-incomplete path instances. $Q$ is processed as before, but is only replenished by the XSchedule's producer, i.e. the context nodes on which to evaluate the location path.

If XScan is used, its fallback behaviour is that it restarts its producer and, after that, behaves as the identity operator. As a result, the whole location path is completely reevaluated. However, due to the partially processed result, the reevaluation benefits from the state of the duplicate elimination structures, preventing reevaluation of already produced results.

## 5.5 Document Order

If the result of the location path requires the nodes to be in document order, sometimes a final sort operator is required in the simple Unnest-Map method [14]. When reordering navigational primitives as in our method, this sort is always required, because the clusters and nodes are not processed in document order.

We assume here that the nodes carry some information that allows to reestablish document order, such as ORDPATHs [21]. However, if this is not the case, or is too expensive, it is possible to perform this sort as part of XAssembly$^R$. During processing, XAssembly can remember the relationships of the border nodes, and use that information to return results in document order. Due to space constraints, we omit the details of this process and assume that ordering information is present in each node.

Sometimes sorting is unnecessary because document order of the result is not relevant, e.g. when an `unordered` keyword is used in XQuery, or aggregates such as `count` are computed on the result.

# 6. EVALUATION

Our goal is to provide a fast and scalable approach to evaluate XPath queries. To evaluate our method, we performed experiments comparing different execution plans for queries from the XMark benchmark on various document sizes.

## 6.1 Environment

The experiments were performed using our native XML-DBMS Natix [9], which features an algebraic XPath compiler capable of generating plans for the Simple method [**?**]. To evaluate the concepts of this paper, we implemented support for XAssembly-based plans in the compiler, and incorporated implementations of XAssembly, XStep, XSchedule, and XScan into Natix' extendible runtime system.

| No. | XPath queries |
|---|---|
| Q6' | `count(/site/regions//item)` |
| Q7 | `count(/site//description)+count(/site//annotation)+count(/site//email)` |
| Q15 | `/site/closed_auctions/closed_auction/annotation/description/parlist/listitem/parlist/listitem/text/emph/keyword/text()` |

**Table 2: Selected XMark queries**

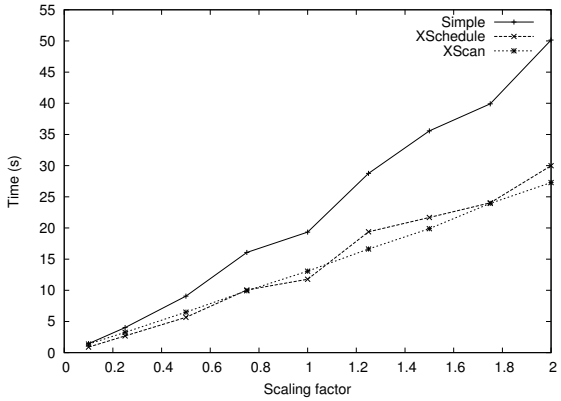


**Figure 9: Results for Q6'**

Natix was configured to use a page buffer with a capacity of 1000 pages. We were careful to prevent any undesired cache effects from spoiling our results: (1) To bypass the operating system cache and any additional levels of indirection in the operating system used, we used Linux' O_DIRECT flag. (2) To avoid a hot on-disk cache, we used exclusively documents larger than 10MB, with a different document size in each run.

## 6.2 Benchmark

In order to produce comparable results for our method, we decided to choose the XMark benchmark [22] as basis for our evaluation. We generated XML documents with the XMark document generator (xmlgen) for scaling factors 0.1, 0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75 and 2.

We want to focus on the effects of our new technique. Hence, we selected those queries which can be fully evaluated using exclusively our new operators. We do not evaluate queries for which our operators would be part of a larger execution plan that also relies on other advanced operators.

Out of the 20 queries in the XMark benchmark, seven can be formulated using a single XPath location path, and two of these (Q7 and Q15) can be expressed with only XAssembly, XStep, XScan and XSchedule. A variant of Q6 with an additional aggregation over the regions is also covered by our operators, and is called Q6' below. The evaluated queries are listed in Tab. 2.

We used our prototype XPath compiler to create execution plans for the selected queries. For each query, three plans were generated, one based on the Simple method, one based on XSchedule with `speculative` set to false, and one based on XScan.

## 6.3 Results

Fig. 9, Fig. 10 and Fig. 11 give results for Q6', Q7 and Q15, respectively. The graphs show a plot of the scaling factor against total execution time. For an XMark scaling factor of 1, Tab. 3 shows the total execution time, and CPU usage both as an absolute value and as a fraction of the total execution time.

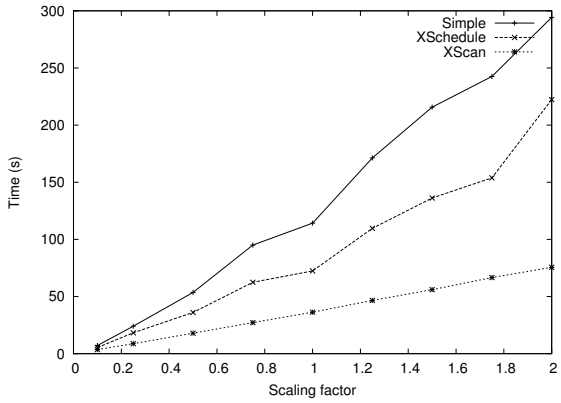


**Figure 10: Results for Q7**

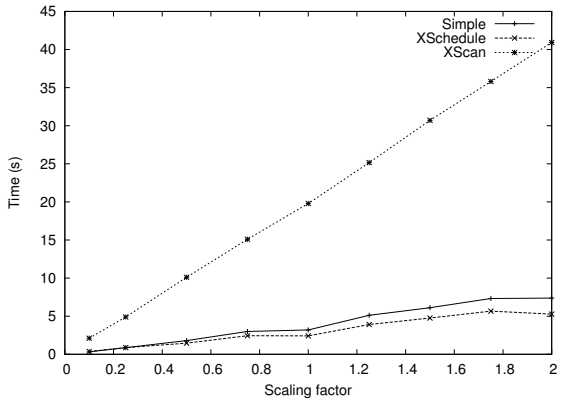


**Figure 11: Results for Q15**

In our experiments, the XSchedule plan was always faster than the Simple plan, with a typical performance gain of more than 50%. Also worth noting are the very similar CPU times for XSchedule and the Simple approach in all queries. The overhead for maintaining the main-memory data structures in XAssembly appears to be minimal, and more than compensated for by fewer I/O operations and higher access locality in the buffer.

Obviously, the performance of the XScan-based plan is linear in the size of the document. As expected, XScan's success depends on the selectivity of the query. For Q7, which has to examine a large part of the document, the low cost of the physical scan pays off, and it is faster by a factor of up to four than the Simple approach and faster by a factor of up to three than XSchedule. For the more selective Q15, the XScan plan is much slower than the other approaches. Here, the scan of the whole document is not a good idea, not only because more pages are loaded than necessary, but also because this leads to higher maintenance costs for the speculative

| Query | | Simple | | XSchedule | | XScan | |
|---|---|---|---|---|---|---|---|
| Q6' | total[s] | 19.33 | | 11.77 | | 13.07 | |
| | CPU [s] | 4.36 | 23% | 3.84 | 33% | 8.39 | 64% |
| Q7 | total[s] | 114.20 | | 72.41 | | 36.25 | |
| | CPU [s] | 23.30 | 20% | 20.70 | 29% | 22.54 | 62% |
| Q15 | total[s] | 3.19 | | 2.42 | | 19.79 | |
| | CPU [s] | 0.26 | 8% | 0.30 | 12% | 15.15 | 77% |

**Table 3: CPU usage for XMark scaling factor 1**

path instances in XAssembly's $S$.

# 7. CONCLUSION AND OUTLOOK

We have introduced an XPath evaluation technique that significantly reduces physical evaluation costs. We do not rely on a special storage layout, instead allowing a great range of physical storage models including formats that can be easily updated and require no preprocessing of the data.

Our approach differentiates between cheap and expensive navigation operations based on the physical location of the traversed nodes. We have introduced a physical algebra that performs all expensive I/O operations for one location path in a single operator. This renders possible optimized I/O scheduling decisions, improving performance. Two alternative I/O-performing operators were presented, the XScan operator based on sequential scans, and the XSchedule operator based on asynchronous I/O. To allow separation of cheap and expensive navigations, the first class citizens of our algebra are so-called partial path instances, representing computations that may not be complete yet, due to pending I/O.

A performance analysis with queries from the XMark benchmark has shown a performance gain up to a factor of four when using the XScan operator. Plans with XSchedule always perform better than plans that use a simple nested-loop method.

This foundation now allows us to explore many different areas in the context of large-scale XML query processing. One direction is to extend our approach to more complex queries. The path expressions considered in this paper have at most two incomplete ends, while in general, the presence of nested paths in predicates may result in path instances with more than two incomplete ends. We also want to investigate how our method can be used to speed up document export, where our "path instance" becomes the textual representation of a whole document (or subtree). Further research is needed to create a cost model to support the choice of the I/O-performing operator. Another area is the evaluation of multiple (sub)queries. Our method can be easily extended to evaluate multiple location paths with a single I/O-performing operator. We also expect concurrent queries to strongly benefit from asynchronous I/O, as scheduling decisions can be made based on more pending requests. Finally, the concept of partial path instances may make sense for other languages and data models than XPath and XML.

# 8. REFERENCES


[1] C. Beeri and Y. Tzaban. SAL: An algebra for semistructured data and xml. In *WebDB (Informal Proceedings)*, pages 37–42, 1999.

[2] S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. XQuery 1.0: An XML query language. Technical report, World Wide Web Consortium, 2002. W3C Working Draft 30 April 2002.

[3] M. Brantner, S. Helmer, C.-C. Kanne, and G. Moerkotte. Full-fledged algebraic xpath processing in natix. In *ICDE*, 2005. to appear.

[4] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *ACM SIGMOD*, pages 310–321, 2002.

[5] P. Buneman, B. Choi, W. Fan, R. Hutchison, R. Mann, and S. Viglas. Vectorizing and querying large xml repositories. In *ICDE*, 2005. to appear.

[6] J. Clark. XSL transformations (XSLT) version 1.0. Technical report, World Wide Web Consortium (W3C), November 1999.

[7] J. Clark and S. DeRose. XML path language (XPath) version 1.0. Technical report, World Wide Web Consortium (W3C) Recommendation, 1999.

[8] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *SIGMOD Conf.*, pages 431–442, Philadelphia, Pennsylvania, USA, June 1999.

[9] T. Fiebig, S. Helmer, C.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann. Anatomy of a Native XML base management system. *VLDB Journal*, 11(4):292–314, 2003.

[10] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *VLDB Conf.*, pages 95–106, 2002.

[11] G. Gottlob, C. Koch, and R. Pichler. Xpath query evaluation: Improving time and space efficiency. In *ICDE*, pages 379–390, 2003.

[12] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[13] T. Grust. Accelerating XPath location steps. In *SIGMOD Conference*, pages 109–120, 2002.

[14] J. Hidders and P. Michiels. Avoiding unnecessary ordering operations in xpath. In *DBLP*, pages 54–74, 2003.

[15] T. Keller, G. Graefe, and D. Maier. Efficient assembly of complex objects. In J. Clifford and R. King, editors, *SIGMOD Conf.*, pages 148–157. ACM Press, 1991.

[16] A. Kemper and D. Kossmann. Dual-buffering strategies in object bases. In *VLDB Conf.*, pages 427–438, March 1994.

[17] A. Kemper and G. Moerkotte. Advanced query processing in object bases using access support relations. In *VLDB Conf.*, pages 290–301, 1990.

[18] C. Koch. Efficient processing of expressive node-selecting queries on xml data in secondary storage: A tree automata-based approach. In *VLDB Conf.*, pages 249–260, 2003.

[19] J. E. B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE Transactions on Software Engineering*, 18(8):657–673, August 1992.

[20] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. In *EDBT Workshop on XML Data Management*, 2002.

[21] P. E. O'Neil, E. J. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. Ordpaths: Insert-friendly xml node labels. In *ACM SIGMOD*, pages 903–908, 2004.

[22] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse. Xmark: A benchmark for xml data management. In *VLDB Conf.*, pages 974–985, 2002.

[23] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *VLDB Conf*, pages 302–314, 1999.

[24] S. A.-K. Shurug, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural joins: A primitive for efficient xml query pattern matching. In *ICDE*, pages 141–152, 2003.

[25] Z. Xie and J. Han. Join index hierarchies for supporting efficient navigations in object-oriented databases. In *VLDB Conf.*, pages 522–533, 1994.