# Overview of Query Optimization in XML Database Systems

Riham Abdel Kader Maurice van Keulen

November 12, 2007

# Contents

# Introduction

In the second half of the 20th century, the need for a standard for database management systems was growing. In 1970, Edgar Codd has laid down the foundation to databases, known as the relational model in which data represented as tuples is stored in tables called relations. One of the main aspects in this model is the use of algebraic operators to manipulate the data. Other types of database management systems, like Object-Oriented and Native XML, have later emerged aiming at supporting storage and querying facilities for other kinds of data. In general, the approach adopted by relational, Object-Oriented and XML database engines for answering a user query is depicted in Figure 1.

As shown in the figure, a user query is first parsed and mapped to its equivalent algebraic representation called logical plan or query plan. This plan is then optimized by applying several optimization techniques and strategies. The output of this phase is an execution plan also known as physical plan. The next phase consists of mapping the execution plan to a sequence of statements which will in turn be processed as a final step towards the generation of results.

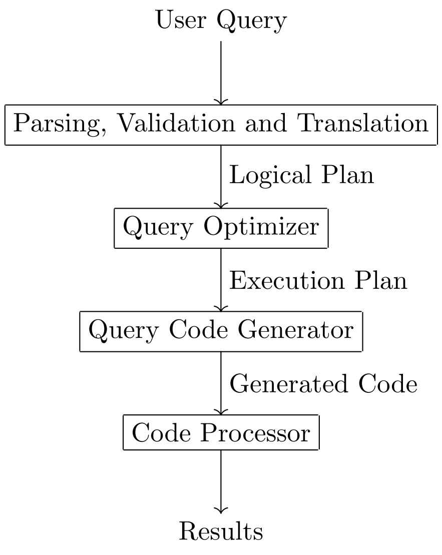

Figure 1: Query processing steps

The logical plan, which has either a tree or a graph structure, consists of a connected sequence of algebraic operators. The set of all operators defined by a database system forms what is called the database's logical algebra. A clearly and precisely defined logical algebra in any database system is the single mechanism to guarantee the soundness and completeness of any query evaluation. Moreover, optimization in database systems is easier in the presence of a logical algebra. A logical algebra is implemented in a physical algebra where each logical operator is implemented by one or more physical operators. Examples of logical operators are: *Select, Project, Join, Union, In-*

*tersection*, etc. Some possible physical implementations of, for instance, the *Join* operator are *Nested loop, Sort-merge,* and *Hash-based.* The difference between these physical operators is in the way they implement the intended functionality of the *Join*, resulting in a difference in the amount of resources (I/O cost, CPU resources, etc) consumed by each. One of the tasks of the optimizer is to map, based on some collected statistical data and cost estimation techniques, each logical operator in the plan to one of its corresponding physical implementations such that the execution time of the plan is minimized. Generally speaking, the optimization step is defined as the process by which the optimal or suboptimal plan is chosen for executing the user query.

The hardest step in the query execution process is the optimization phase. This report tries to give a closer overview of the state of the optimization process in the context of XML database systems. With this aim in mind, we first briefly summarize the traditional optimization techniques used in the relational model. We then review the different proposed systems for processing XML data. Finally, we present the research done in the field of optimizing the query execution in XML databases. We conclude this report by enumerating some of the problems still faced in the optimization process in the context of XML database systems.

# Chapter 1

# Optimization in Relational Database Systems

As mentioned in the Introduction, every user query submitted to a database system is converted to a logical plan which is then mapped to a physical plan. The aim is to choose the most favorable physical plan given some underlying statistics. It is the task of the optimizer component to make this choice, by applying several optimization steps which are depicted in Figure 1.1.

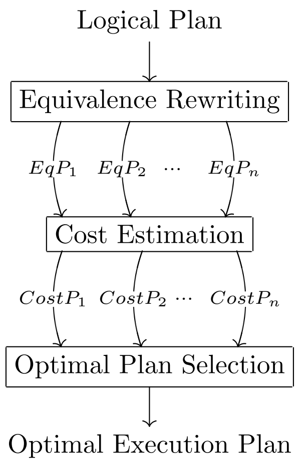

Figure 1.1: Optimization steps

The optimizer enumerates alternative plans by applying some rewriting rules to the original one, and picks the optimal plan according to statistical data distributions and cost estimation. One requirement of the optimization process, in the case of ad-hoc queries, is that it should be fast, *i.e.* it should represent only a small percentage of the time needed to execute a query.

This section describes each optimization step and gives a brief overview of the optimization techniques adopted in relational database systems. The ideas in Section 1.1 and Section 1.2 are based on information gathered from the two books [35, 39].

## 1.1 Search Space

The logical plan to which a user query is translated is in most cases not the optimal one. For this reason, the optimizer rewrites the original plan into equivalent query plans which will have a faster execution time. By equivalent query plans, we mean query plans that have a different ordering of their operators (and sometimes a different number of operators) and that still return the same result. The enumeration of plans is done in two different steps: algebraic rewriting and choice of physical operators.

### 1.1.1 Algebraic Rewriting

The algebraic rewriting consists of substituting a subtree $T$ of logical operators by an equivalent one $T'$ which might have an equal number of operators as $T$ with a different ordering, or a different number of operators. There are many rules for transforming an algebraic expression into an equivalent one, some of which are based on commutativity and/or associativity properties of the operators in the expression. A very small portion of equivalence transformations are listed below:

- Cascade of $\sigma$: $\sigma_{pred1 \wedge pred2}(R) \equiv \sigma_{pred1}(\sigma_{pred2}(R))$.
- Cascade of $\pi$: $\pi_{attr_1}(R) \equiv \pi_{attr_1}(\pi_{attr_2}(...(\pi_{attr_n}(R))...))$, if for all $i$, $attr_i \subseteq attr_{i+1}$ and $attr_i$ is a subset of the attributes of the relation $R$.
- Commuting $\sigma$ with $\pi$: $\sigma_{pred}(\pi_{attr}(R)) \equiv \pi_{attr}(\sigma_{pred}(R))$, if $attr$ includes all attributes used in $pred$.
- Commutativity of $\bowtie$ (or $\times$): $R \bowtie S \equiv S \bowtie R$.
- Associativity of $\bowtie$ (or $\times$): $R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$.
- Pushing $\sigma$ through $\times$: $\sigma_{pred}(R \times S) \equiv R \bowtie_{pred} S$, if $pred$ contains a condition comparing attributes from R to attributes from S.
- Commuting $\sigma$ with $\bowtie$: $\sigma_{pred}(R \bowtie_{pred'} S) \equiv \sigma_{pred}(R) \bowtie_{pred'} S$, if $pred$ contains conditions on only attributes in $R$.

The large number of existing equivalence rules results in the possibility of generating for a given logical plan a large number of equivalent plans of which some are faster. Enumerating a large number of plans and then performing a search operation to find the optimal one among them, slows down the optimizer, and hence violates the requirement stating that the time used by the optimization phase to optimize ad-hoc queries should be a small fraction of the total time to process the query. Moreover some of the algebraic rewriting rules are not optimizations if applied by themselves. For example, cascading a Select operator might not lead to a better plan. If, however, it is combined with the rule for pushing Select operators through Joins, it will result in faster plans. For these reasons, several heuristics based on general observations are proposed and are adopted by database management systems. These heuristics aim at restricting the number of rules to be chosen for rewriting a given plan and thus reduce the size of the generated search space. Some basic heuristics are listed below:

- Break a cascade of Select ($\sigma$) operators and push $\sigma$ as far down in the query tree as possible.
- Break a cascade of Project ($\pi$) operators and push $\pi$ as far down in the query tree as possible.
- If possible, always merge a Cartesian Product ($\times$) and a Select ($\sigma$) operator into a join operator.
- Plans are generally restricted to left deep plans.
- Reorder Joins such that the ones that generate the smallest intermediate result sizes are executed first.
- Group operators in the plan that can be executed by a single scan of the relation to reduce the number of relation scans.

One of the main heuristics used by optimizers is to reduce as early as possible the size of intermediate relations generated by the operators in the plan. This is accomplished by executing Select and Project operators as early as possible to reduce the number and size of the generated tuples and by executing operators with the lowest selectivity before other operators. This is, however, only a heuristic: in some cases pushing selection or projection down to a certain relation in the plan might eliminate the possibility of using an index in the join processing which might result in a suboptimal plan. Therefore another heuristic is to generate plans that preserve the possibility of using built indices and that construct as much as possible the smallest intermediate results.

### 1.1.2 Choice of Physical Operators

The logical plans produced from the algebraic rewriting phase do not indicate how the operators in each of these plans are to be executed. Each logical operator in the database is implemented into several different physical operators. The physical operator determines the way the data on disk is accessed and the way it is manipulated and processed.

After the logical plans are generated, the optimizer will substitute each logical operator by one of its corresponding physical implementations such that the resulting physical plan is the most efficient. Although not as big as the number of equivalent plans possibly generated by the algebraic transformations, the number of choices for translating a logical plan into a corresponding physical one is still big. First, there are a number of methods in which stored data can be accessed from disk, including a simple scan of the relation, a scan over an index, an index-based predicate lookup. Second, as mentioned in the Introduction, the functionality of each logical operator can be implemented in several alternative ways.

The choice of the physical implementation of a logical operator depends on several factors: the existence of an index or a hash table, sortedness of the input relations, etc. The Join operator, for example, will be replaced by a sort-merge implementation if the input relations are known to be sorted by the joined attributes, and will be substituted by a hash-based algorithm if the joined attributes are hashed using the same hash function.

It is also possible to replace a sequence of logical operators by one physical operator. For example, a Select and a Project operator can be executed in a single algorithm instead of two separate ones and a Join can be combined with a Select and a Project into one physical operator. This reduces the number of generated intermediate tables and the number of times the data is accessed, which results in decreasing the processing time of the plan.

Generated physical plans can be classified in two categories: pipelined and materialized plans, depending on whether they contain a blocking operator. In a non-pipelined plan, all tuples are held by at least one blocking operator in the plan, before being processed and output to the next operator. The Sort operator is an example of such a blocking operator as it needs to compare all its input tuples before outputting them in the required order. Once sorted, these tuples are then routed to the next operator. In some cases, if the intermediate result input to or generated by the blocking operator is large, then it needs to be materialized. In a pipelined plan, tuples are not held by any operator and every tuple is directly routed to the next operator as soon as the current operator has finished processing of the tuple. Typically, pipelined plans are a better choice if faster response time is needed. In general, they are preferred since they do not require any materialization of results.

The generation of alternative plans is, as discussed above, based on heuristics thus the optimizer can not decide which one of the plans is the best. Therefore the optimizer will need to examine every plan assigning a cost to each and then picking the one that satisfies the required optimization criteria which is in most cases fastest execution time. To accomplish this, a cost model is defined with which each operator and subsequently each plan is associated with a cost.

## 1.2 Cost Model

One important property of a physical operator is its execution cost accounting for the resources used by the operator to produce its result. The cost components are:

- CPU cost: which represents the cost of performing operations (*i.e.* computation, searching, sorting) on the data buffers in memory.
- I/O cost: is the cost for accessing secondary storage. It includes cost for searching, reading, and writing disk blocks, and storing intermediate

results when these are too big to fit in memory. The access cost for a block on disk depends on the data layout (files stored contiguously or scattered on disk) and the access method (such as indexes, hashing).

- Communication cost: which consists of transferring the query and its results in a distributed environment.

To estimate the cost of the operator, the cost components are combined using a predefined cost function, where a weight might be assigned to each resource to reflect the value of its contribution towards the final cost. The costs of all operators in the plan are then summed up to compute the cost of the whole plan. The optimizer will then compare the cost of all generated plans and will choose the one having the minimum cost. In reality it is not an easy task to combine all cost components in a single function and to give a suitable weight for each. Therefore in most databases the cost model is simplified and cost functions only take into account the I/O cost (since, in non distributed systems, it contributes the highest to the cost of the plan), *i.e.* the optimizer searches for the plan with the minimal I/O cost.

To fairly compare all plans, the optimizer should assign cost estimates that are as accurate as possible without spending much time in computing these costs. We present in the next section the type of information that needs to be collected and saved in order for the optimizer to accurately estimate the cost of plans.

### 1.2.1 Statistics

The cost of operators highly depends on the size of their input relation(s). Therefore the optimizer needs a way to estimate the cardinality of operator's input, in other words to estimate the selectivity of operators. The operator's selectivity is the ratio of the number of tuples output by the operator to the number of input tuples. To accurately estimate the selectivity of operators, some statistical information needs to be collected by the optimizer and saved in the DBMS catalog for later use. Different kinds of statistics are gathered for each relation $R$, we enumerate few of them here:

- *Tuples(R)*: number of tuples in the relation $R$,
- *Blocks(R)*: number of blocks needed to store relation $R$,
- *TupleSize(R)*: size of tuple in the relation $R$.

Some statistics are also gathered for some attributes $A$ in the relation $R$. The most used ones are listed below:

- *DistinctValues(R, A)*: number of distinct values of attribute $A$ in relation $R$,
- *Max(R, A)*: maximum value of attribute $A$ in relation $R$,

- *Min(R, A)*: minimum value of attribute $A$ in relation $R$.

Moreover the catalog might also save for each relation information about:

- its primary key if it has one.
- the way its tuples are stored: unordered, ordered, contiguously, sparsely,
- the names of indexed or hashed attributes,
- built hash tables and indices and their type (primary, secondary, or clustering index).

Statistics are most of the time estimates and not accurate reflection of the database state. In fact tuples might be deleted, inserted and modified frequently and updating the catalog after every change to the database state can be expensive.

The usage of the statistics to estimate the selectivity of operators is based on some assumptions made by the optimizer. Although these assumptions, in most cases, do not hold in reality, they are adopted to make the selectivity estimation process faster and easier. Some of the used assumptions are:

- Uniform distribution of attribute values,
- Independence of attribute values,
- Constant number of tuples per page,
- Random placement of tuples among pages.

The value distribution of attributes directly affects the selectivity of operators in the query plan and assuming it is uniform might lead to inaccurate selectivity estimates. One more accurate method to capture the value distribution of a certain attribute and hence to better estimate the selectivity of operators is to build a histogram on that specific attribute. A histogram on attribute $x$ divides the values of $x$ into $k$ buckets where each of the bucket corresponds to a range of values such that the height of the bucket determines the number of attributes included in the range. The larger $k$ is, the higher the histogram accuracy. The values within a bucket are, however, assumed to be uniformly distributed. Correlations among attributes can also be captured by building a two dimensional histogram. This is, however, very large, therefore in many systems a two dimensional histogram is substituted by some statistical information such as the number of distinct pairs of values or a combination of an histogram on the first column and the associated distinct number of values in the second column [10].

### 1.2.2 Cardinality and Cost Estimation

Using the statistics collected on base data, the optimizer will assign for each operator in the generated plans a selectivity value. The cardinality estimation of the input relation(s) of all operators can thereafter be computed: by, starting from the base tables, incrementally multiplying the selectivity of each operator to the cardinality of its input table(s).

As mentioned earlier, the computation of the cost of each operator is, in most database systems, simplified such that it takes into account the I/O cost only. In such a setting the operator's cost function consists of the cardinality of its input relation(s) since the number of I/O operations that the operator will perform is dependant on the cardinality estimation of its input table(s). The cardinality estimation is one essential parameter for estimating the number of disk accesses but not the only one. Other important factors that affect the number of I/O accesses carried by an operator are the physical layout of data and indices on disk and the amount of memory available for buffering disk pages. Therefore although the operator's cost is reduced to its I/O cost, the multiplicity of parameters involved in the computation of the I/O cost renders this operation a difficult and complex one.

Once the cost of every operator is derived, the cost of the plan is then computed by summing up the operators' cost. The different plans can then be compared and the one with the minimum estimated cost is chosen for execution. It should be mentioned that generating accurate plan cost estimations remains one of the hardest operations and open issues in the optimization process.

## 1.3 Plan Enumeration

A plan enumeration algorithm is designed to efficiently explore the search space of equivalent plans to pick an optimal one based on cost estimations. Early database systems restricted their search space by using certain heuristics, one of which is to consider only left-deep plans. With the increase of query complexity and the power of computers, the search space was expanded to bushy trees and plans containing cross products and other kinds of operators, resulting in better and more complex plans. More sophisticated plan enumeration techniques were then proposed and integrated in optimizers. One main focus of plan enumeration algorithms is to find a good solution for the join ordering problem. The proposed enumeration algorithms tackling this problem can be categorized into three classes [50]:

- Deterministic algorithms,
- Randomized algorithms,
- Genetic algorithms.

### 1.3.1 Deterministic Algorithms

Algorithms in this class build a good plan step by step in a deterministic manner, either by employing some heuristics or by performing an exhaustive or pruned search of the space.

**Dynamic programming [10]:** One of the algorithms proposed in this class is based on dynamic programming. It searches for a left-deep plan in a bottom-up fashion starting with single relations and adding increasingly join operations and relations. In the first round, the algorithm enumerates alternative plans with all possible scans for every joined relation. In the subsequent loops, it constructs in the $k$th iteration a set of $k$-joins plans by considering the set of $(k-1)$-joins plans and extending each of these plans with a join operation. The set of alternative plans is pruned at each iteration by removing every plan that has a cheaper equivalent one. Two plans are considered equivalent if they join the same set of relations and the sort order of their result is the same. A plan is interesting even if its cost is not minimal as long as its output is sorted or has an index since this can reduce the cost of subsequent operations such as sort-merge join, duplicate elimination, and grouping. Finally, the algorithm will return a set of partial solutions consisting of at least one optimal solution. A disadvantage of this dynamic programming approach is the high memory consumption making the optimization of queries consisting of 10 joins and more very expensive.

**Minimum selectivity algorithm [50]:** Another algorithm in this class is based on the minimum selectivity heuristic which considers plans characterized by low cardinality intermediate results as good solutions. This algorithm builds a left-deep plan by recursively selecting the join with the smallest selectivity factor such that the size of the input to the next join is reduced as much as possible. The algorithm starts by picking the relation with the smallest cardinality and adds it to the plan. The relation whose join with the chosen relation has the smallest selectivity factor is then selected and appended to the plan. The process repeats by finding the relation that joins, with a lowest selectivity factor, with any of the previously selected relations. The algorithm terminates when all relations are added to the plan.

### 1.3.2 Randomized Algorithms

Randomized algorithms view plans as points in the search space where a cost is associated with each point. Two points are connected if and only if they can be transformed into one another by exactly one move. The algorithms perform a random walk through the space via a series of moves according to certain rules and terminates when no more moves are applicable or the time limit is attained. Iterative Improvement and Simulated annealing are two examples of randomized algorithms.

**Iterative improvement [52]:** This algorithm starts from a random point in the space and selects one random neighbor. If the cost associated with the selected neighbor is smaller than the current state's cost and the selected point is considered a local minimum then the move is completed, otherwise another random neighbor is picked. This process is repeated starting from the new state until a number of moves is performed or a time limit is exceeded. The point with the lowest cost encountered so far is the selected plan.
Iterative improvement does not attempt to determine the neighbor with the lowest cost before performing its next move. In fact, the number of neighboring points is high and it will be very impractical to compare all the neighbors of a state to identify the local minimum. For this reason only a subset of all the neighboring points are compared and the one with the lowest cost and whose cost is lower than the current state's cost is declared local minimum.
One disadvantage of this algorithm is that it can be caught in a high-cost local minimum. Since it only accepts moves that lead to a lower cost, the chosen plan might be unacceptable if the search space contains a large number of high-cost local minima.

**Simulated annealing [52]:** Simulated annealing overcomes the problem presented by Iterative Improvement by accepting moves that increase the cost and thus avoids being trapped in a high-cost local minimum. This algorithm is based on the annealing process of crystals from liquid solutions which explains the reason why some physics terminology is often associated with it. Simulated annealing proceeds the same way as Iterative Improvement but it also accepts a non-improving move with a certain probability. This probability depends on the increase between the current and the new state's cost and another predetermined parameter called *temperature*. The lower the increase in the cost and the higher the temperature, the higher the probability to accept the move. The value of the parameter *temperature* is repetitively decreased during the plan enumeration process.

### 1.3.3 Genetic Algorithms

The ideas presented in this section originate from the work presented in [50].

Genetic algorithms are based on Darwin's principles of survival of the fittest. It simulates the natural biological evolution process: the fittest members of a generation are the one to survive and their offsprings will inherit their features.

One of the main characteristics of a genetic algorithm is that it works with a set of solutions (the population) instead of considering a single one. Another characteristic is that solutions should be coded as strings called chromosomes using a suitable encoding. For example, the following left-deep tree $(((R_2 \bowtie R_4) \bowtie R_3) \bowtie R_1)$ can be represented as "2431". Each solution is associated with a fitness measure that reflects its cost.

The algorithm starts with a random population of solutions from which offsprings are generated by random crossover and mutation. The fittest members of the newly generated offsprings will be selected to become parents and will be propagated to the next generation. The subsequent population will be derived from them, and the process is repeated. The algorithm terminates after a predefined number of generations or when no further improvement can be made. The chosen plan is the fittest solution in the last generation.

A basic genetic algorithm usually defines 3 operators: selection, crossover and mutation. The performance of a genetic algorithm depends on how these operators are defined. The selection operator chooses the fittest members of a population and propagates them to the next generation. The most common type of selection is the roulette wheel where every individual is given a probability proportionate to its fitness. Solutions are then selected based on these probabilities. The crossover operator generates offsprings by combining a number of the fittest members of the population. It is equivalent to the biological process of a parent passing some of its genetic characteristics onto its offsprings. The mutation operator applies a random alteration to a random set of members of the population. The objective from using this operator is to introduce new features not present in any member of the population and to ensure diversity among the generation. The mutation operator prevents from falling into a local optimum.

# Chapter 2

# Existing XML Database Systems

The eXtensible Markup Language (XML) [8] is defined and standardized by the W3C as a simple and flexible text format to exchange data in distributed systems and over the Web. Elements in an XML document can be nested within each other, resulting in a tree-like structured representation. This tree representation is what is known as the memory-resident Document Object Model (DOM) where a node represents an element, an attribute, or text data. Beside the tree-based representation, an XML document can also be linearly viewed as a stream of tokens. Simple API for XML (SAX) [21] parses an XML document as tokens and is invoked by callbacks to handle the parsed tokens.

Several languages have been defined for selecting and transforming XML data, of which XPath [4], XQuery [5], and XSLT [33] are standardized by W3C. XPath expressions are defined against a node-labeled tree view of an XML document where an XPath represents a traversal over the nodes in this tree along its axes. XPath expressions consist generally of a series of steps where a step is an axis identifier and a node test (example: "*//site/regions//item*"). This XPath searches for all *site* elements descendant of the root of the document, then for all *regions* nodes children of the *site* elements, and finally returns all *item* nodes descendant of the retrieved *regions* elements. The subexpression "*//site*" is a step where "//" is an axis of type descendant and *site* is a node test. The evaluation of an XPath expression returns either a sequence of atomic nodes or a sequence of nodes with their subtree. One important requirement is that the nodes in the returned sequence should be ordered in document-order unless the user explicitly disregards order.

While XPath can only perform selections on an XML document, XQuery supports richer operations (*e.g.,* selections, joins, projections and aggregations). In fact, XQuery is defined as a superset of XPath and path expressions in XQuery are based on the XPath syntax. The basic building block of XQuery is the *FLWOR* expression (an acronym for *for* ... *let* ... *where* ... *order by* ...

*return* ...). XQuery also allows the generation of new XML documents. This facility is provided by the *Element Constructor* which permits the construction of new elements and attributes with their data and structural relationships.

There have been several proposals for systems to evaluate an XQuery against stored or streaming XML documents. These processing systems differ in several aspects, two of which are the storage layout adopted and the type of data-model and algebra chosen. In the following, we review some of the most known and used systems and briefly describe the specifications of each.

## 2.1 Evaluating XQueries over Stored Data

Current systems for processing XQueries over stored XML documents can be classified into three categories: native, memory-resident, and relational approaches. Systems belonging to the same category differ in the data model and algebra they define. We present a more detailed description of each of the categories.

### 2.1.1 Native XML Database Systems

A native XML database system is designed and structured according to the semi-structured data model and its specialized querying languages. The defined model should at least include elements, attributes, PCDATA and take into account document order. Examples of native XML research prototypes are *Timber* [30] and *Natix* [18]. Each one of these systems provides its own XML-specific storage manager that is capable of storing and indexing XML documents. The evaluation of an XQuery starts by generating its logical query execution plan, then optimizing the plan which is finally sent to the query processor that executes it through accesses to the storage manager. Beside the storage manager and the query processor, *Natix* supports also all familiar features of a DBMS transactional manager, such as recovery and concurrency control. On the other hand, Timber allows the modification of the XML document (i.e. node/content deletion, insertion, and update). The two systems adopt a different type of algebra and a different approach for processing XPath expressions.

The algebra used by Timber for processing XML documents is called Tree Algebra for XML (TAX) [31]. As its name implies, it is a tree-based algebra that manipulates sets of ordered labeled trees rather than sets of tuples. Each operator in TAX takes as input one or more collections of data trees, and generates a collection of data trees as output. The core of the TAX operators is a pattern tree which consists of a node-labeled and edge-labeled tree and is used as a mean to identify nodes and attributes of interest while evaluating a query. Figure 2.1 shows an example of a pattern tree representing an XPath expression. The edges of the pattern tree can be of type child or descendant. The double frame indicates the node to be returned from the evaluation of the pattern tree. The functionality of the TAX operators is to match their core pattern tree against the input of collection of trees and return a set of subtrees, called witness trees, that have a structure identical to that of the pattern tree. The authors in [31] admit that the biggest challenge in defining a tree algebra for XML is the flexibility and heterogeneity properties of this language. That is data trees in an XML document might have a complex and variable structure due to missing and/or repeated sub-elements, whereas an algebra requires the manipulation of objects having a homogeneous structure. This and other kinds of difficulties were overcome by introducing new types of pattern trees: the *Generalized Pattern Trees* [11], and the *Tree Logical Classes (TLC)* [42].

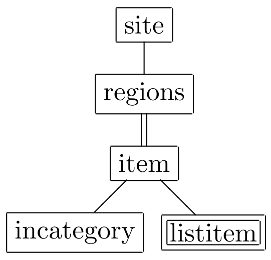

Figure 2.1: Pattern Tree of the path "*//site/regions//item[/incategory]/listitem*"

The approach used by Timber for processing an XPath expression is join-based; *i.e.* the pattern tree equivalent to the XPath is mapped to a sequence of join operators and executed as such. For example the path "*//regions//item*" is translated to a join between the retrieved sequences of "*regions*" and "*item*" elements. This join operator, named Structural join [2], is not a normal relational join although it is very similar to it. In fact this join is made tree-aware, that is, it can use the properties of the XML structure to optimize the execution of the XPath expression. Another method for executing a path is by processing its equivalent pattern tree using a Holistic Twig join operator [9]. The technique uses a chain of linked stacks to represent partial solutions to every node in the pattern tree. These solutions are then composed to obtain the result of the path.

Query processing in the *Natix* native XML database system is based on a tuple-based algebra [18]. The data model consists of a set of ordered sequences of tuples where each tuple consists of a sequence of attribute values which can be a number, a string, an XML node, a text node, an XML attribute node, or a sequence of tuples. This last type, sequence of tuples, introduces the possibility of having nested tuples. The work in [7] provides a complete algebra for the XPath language as a first step towards an XQuery algebra which in turn is described in [38]. It presents translation rules to translate any XPath expression into an algebraic sequence of operators by decomposing the path into location steps, where each step is translated into an *Unnest-Map* operator, and then connecting these operators by a *d-join* operator. Unlike Timber, the processing in Natix is navigation-based; *i.e.* solutions are computed by starting from a context node and navigating some part of the XML document (e.g the node's subtree), and analyzing encountered nodes each one at a time.

Examples of other native XML database systems are: eXist [15], Oracle Berkeley DB XML [41], and X-Hive [58].

### 2.1.2 Memory-resident XML Database System

A memory-resident database system relies on main memory for data storage. It eliminates access to disk by loading the data, storing and manipulating it in memory. The *Galax* system [16], an open source implementation of XQuery, does not provide a storage manager, instead it operates on memory-resident XML documents. The main goal of *Galax* developers is completeness: they handle XPath expressions, FLWR expressions, user defined functions, and overloaded built-in functions. Researchers can use *Galax* as a platform for experimenting with a complete XQuery implementation. *Galax* builds the in-memory DOM tree representation of the queried XML document before evaluating an XQuery against that document. The fact that this system performs the query evaluation against memory-resident XML documents renders this system unable to handle the processing over large documents.
The algebra for the *Galax* system is presented in [47] where a complete algebra for XQuery that integrates both a tuple-based and a tree-based algebra is proposed. The advantage of this algebra is that it combines XML values and tuples, enabling the use of traditional optimization techniques developed for the relational paradigm. An XML value is defined in this system as an ordered sequence of items where an item is an atomic value or a node (*i.e.,* element, attribute, comment). A tuple is a record with fields containing XML values. A large set of operators is defined and classified into three categories: XML Operators; Tuple Operators; and XML-Tuple Operators. The first set manipulates XML values only, the second set only handles tuples, and the third set consists of boundary operators capable of supporting both kinds of data. The work in [47] also describes rules on how to compile path expressions, type expressions, and FLWR expressions into an algebraic representation.

Saxon [49] and Qizx [46] are also memory based XML systems which have a java-based implementation.

### 2.1.3 Relational XML Database Systems

Some other research prototypes [6, 13, 53] prefer reusing mature relational technology instead of designing and building a native XML system from scratch. Relational technology is reused by mapping the XML data model into the relational model and storing XML documents in a relational database. In order to process XML queries in this context, two approaches can be used; the query is either directly mapped into a logical plan of relational operators, or translated first into an SQL expression and then compiled to a relational algebra, before being sent to the database engine for execution. However, several challenges such as efficiently evaluating XPath and XQuery expressions and reconstructing the document-order of the returned XML elements can arise from this mapping. To overcome these shortcomings, the relational algebra is usually extended with some XML-specific operators.

*MonetDB/XQuery* [6], a system built over a relational main-memory engine, supports the processing of XQueries by compiling an XQuery into a DAG of relational operators. It defines a number encoding for XML data: the *XPath accelator* structure [24, 28]. Based on this structure, an XML document is mapped into a two-dimensional plane representation by assigning to each node in the document a pre-order and a post-order rank. With this numbering scheme, every node in the XML document divides the plane into four disjoint regions, each corresponding to one of the XPath axes: preceding, descendant, ancestor, and following. Thus the process of selecting nodes from the document consists of simple selection predicates on the pre/post-order ranks, easily indexed and optimized by an RDBMS. Updates to the database are also supported in the MonetDB/XQuery system.
As one can already predict, MonetDB/XQuery adopts the relational algebra as its processing algebra. However it extends it by two new operators [23]: the *node construction* operator and the *staircase join* operator. The former is used to support the element construction facility provided by XQuery, while the latter is adopted to accelerate the evaluation of XPath location steps. The staircase join operator [26, 27] represents a single XPath step and is built upon the XPath accelerator structure. It uses the *pre/post* numbering to exploit additional tree properties and thus is tree-aware. This extra tree knowledge allows the staircase join to reduce its search space without missing any matching node. The staircase join is also extended by a loop-lifted version [6] to efficiently evaluate XPath expressions nested in *for* loops.

As mentioned above, other proposals first translate XPath and XQuery expressions into SQL queries and then map the SQL statements into an algebraic relational representation. The work in [53] proposes three order-encoding methods to represent XML order in relational databases, and defines rules to translate ordered XPath expressions into SQL queries by generating an SQL fragment for each location step in the XPath. Finally, these fragments are concatenated into a single SQL expression. The authors in [13] introduced a compositional translation from a subset of the XQuery language into a set of SQL view definitions. The SQL query is then compiled into a relational algebra and executed.

## 2.2 Evaluating XQueries over Streaming XML Data

Several systems were also proposed to process XQueries against streaming XML documents. We give, in this section, a brief overview of some of them.

*Tukwila* [29], an XML query engine, targets mostly the data integration of several streaming XML documents. *Raindrop* [51], an XQuery subscription system, is another framework that supports query evaluation over streaming XML documents. Both systems support a pipelined execution of an XQuery. The main contribution of these approaches is the integration of both the automata-based and the algebra-based approaches for query evaluation. The automaton approach is a state-transition model well suited for matching regular expressions over a certain alphabet. Since basic XPath expressions can be viewed as regular expressions and the tokens generated by the streaming documents as the alphabet, the automaton model was adopted to structurally match XPath expressions against the input XML data. This automaton model, however, can not handle queries that are more complex than XPath and does not provide a good support for sophisticated query optimization techniques. In contrast, the algebraic model can handle the expressive power of complex queries and offers a good foundation for query optimization. The algebraic paradigm, however, expects a well-defined data model on which it can operate, and tokens derived from an XML stream do not meet this requirement since a token looses its semantics and structure when presented alone without the context provided by other tokens in the stream. For these reasons, it was thought to use these two paradigms together when processing XQueries over streaming XML data since they do complement each other. The data model used by such an approach consists of XML tokens as the source data and tuples as the intermediate data. The first level of operators in the logical plans are automata-based operators. These operators act as wrappers over the nested structure of the XML documents, that is they hide this nesting from the other operators in the plan. They manipulate the tokens generated by the streaming documents, match XPath expressions, construct tuples for the selected XML elements and subtrees, and then forward these tuples to the algebraic operators in the plan.

The *XML Stream Machine* (*XSM*) [36] is a transducer-based system for processing XQueries over XML streams. Transducers are similar to finite state machines in that they perform actions based on their current state and a given input. This system evaluates an XQuery by first translating it into a network of XSMs linked via internal buffers and then reducing this network into one single XSM by recursively applying the XSM composition operation. The aim of this reduction is to diminish the number and the size of the intermediate buffers and minimize the computation performed for each token generated by the stream. Execution code is then automatically generated for the optimized XSM that can then be efficiently processed.

Other systems for processing streaming XML data are TurboXPath [54], BEA/XQRL [20], and FluXQuery [34].

Although some optimization techniques for processing XML queries over stored documents can also be used in the streaming context, optimization of queries over streaming data requires a different set of techniques like: load shedding, adapting to changes in a stream's rate, collecting different types of statistics .... Our presentation of the XML optimization techniques in the following sections will not cover those techniques specific to streaming data as this is outside the scope of this report.

# Chapter 3

# Optimization in XML database Systems

After presenting the most common optimization techniques used in relational databases, and describing some of the existing XML database systems, we are now ready to give an overview of optimization in the XML context.

## 3.1 Equivalence Rewriting Based Optimization

We have described in section 1.1.1 some rules that rewrite a relational algebraic expression to an equivalent one. In XML, equivalence rewriting can be accomplished at two different levels: XQuery core level which is equivalent to syntactical rewriting, and algebraic level.

### 3.1.1 Syntax Rewriting

Figure 1 states that queries submitted to a database system are mapped to logical plans. For XPath and XQuery queries, this is not always the case. There exists an extra intermediate step where a user query is first converted into a core language before being translated into a logical plan. The XQuery core language defined in [14] is a proper subset of the XQuery language and consists of a set of simple expressions to which the more complex XQuery expressions are rewritten. The process by which this mapping is accomplished is called the normalization phase. The need for normalizing a certain query lies in the fact that static type analysis and dynamic evaluation rules defined by W3C in [14] are described in terms of the core language, and thus are better applied on the core expression representation of the query. Optimizing an XPath or XQuery query by syntactically rewriting it, is applied either on the query itself or on its equivalent core expression. The papers described in this section present optimization techniques based on syntactic rewriting rules.

**Backward Axes Removal** Several XML processing systems can not handle XPath expressions with reverse axes, such as parent, ancestor, preceding, etc. Moreover, some types of systems (*e.g.*, streaming systems) are not efficient in evaluating this group of axes. An example of an XQuery core level equivalence rewriting is the technique described in [40]. It tries to overcome the first problem and to optimize the processing in the second situation by defining two sets of equivalence rules to rewrite an XPath expression involving reverse axes to an equivalent reverse-axis-free one. One set of rules rewrites the XPath to an expression including a join operator while the other set rewrites the XPath to one containing more location steps.

**Order and Duplicate Elimination** One important requirement with optimization potential is that nodes returned from XPath or XQuery evaluation should be in document-order unless the user explicitly disregards order. The XQuery standard specifies that an XPath is translated to an XQuery core expression such that every step in the XPath is followed by a distinct-docorder operator. This operator maintains not only the document order but also the uniqueness of the nodes throughout the path evaluation. Although having these operators after each step keeps the intermediate results duplicate-free, it might slow down the system's performance. An alternative is to sort and eliminate duplicates only at the end of the path evaluation but this will result in performing unnecessary and redundant work which may also degrade the system's performance. The technique proposed in [17] decides after which steps in the plan to keep these expensive operators and after which they can be safely removed, such that the evaluation time of the plan is minimized. Properties like ordered, and duplicate free are assigned to a path expression if its evaluation returns a set of nodes that conforms to these properties. The described technique is an automaton-based algorithm called DDO which, given a starting set of nodes having a certain property, can infer the property of the set resulting from applying a certain XPath axis. By inferring the properties of the result, it can be decided if the presence of a sort and/or duplicate elimination operator is necessary after this step. Although this proposal can handle all XPath axes, it is limited by one constraint: the input to the start state in the automaton can only be a singleton node.

Other systems tackle the order and duplicate-free problem at a lower level. The duplicate elimination and order problem in MonetDB/XQuery, for instance, is partially solved inside the staircase join itself [27]. This operator is implemented such that it generates results sorted in document order and free of duplicates. Order is preserved by using the pre/post numbering of nodes, while distinctness is guaranteed by pruning out the elements from the input context nodes for which the application of the axis step in question will return identical results. Structural joins [2] for example, used in Timber, are also implemented such that they return the result sorted either in reverse or in document order depending on the used variant. If the result is output in document order then no sort operator is needed. Holistic Twig joins [9], also implemented in Timber for evaluating XPath expressions, output a result sorted in document order. These two classes of operators, however, do not guarantee a duplicate-free result.

### 3.1.2 Algebraic Rewriting

Since only a subset of the heuristics applied in relational database systems (section 1.1.1), like predicate push down, can be imported to the XML context, investigation in new strategies is definitely needed. In this section, we give an overview of the rewriting rules at the algebraic level in XML systems.

**Join Reordering** Join reordering is a very important and widely used optimization technique in databases that use an algebra with joins, thus it is very natural to study its application in the XML context. In relational databases, join reordering is a highly studied problem and as we have seen in section 1.1.1 and in section 1.3 several rewriting rules and enumeration algorithms were proposed for reordering join expressions. One fundamental difference for ordering joins between relational and XML systems lies in the fact that XQuery returns its generated result in document order and thus manipulates order-preserving joins. These types of joins are associative but not commutative, which reduces the number of alternative plans that can be generated by the optimizer. Moreover, adopting the general heuristic used by optimizers in relational database systems that only considers left-deep trees will limit the search space even more. These two constraints enforced in the process of generating alternative plans increase the chance of missing the optimal or suboptimal plan. The work presented in [37, 57] tackles the problem of join reordering in the Natix and Timber systems. The proposed solution is to extend the search space by considering plans that are not left-deep and using joins that are not order-preserving since order can be recovered at a relatively small cost by adding a sort operator at the appropriate position in the plan. The conclusion made by the authors is that for optimizing the evaluation of XQueries contrary to relational database systems, exploring a larger search space increases the optimizer's chance to find a better plan.

**Plan Simplification** The work on algebraic equivalences in the context of MonetDB/XQuery is presented in [25]. The paper argues that plans generated by Pathfinder are large in size which makes the application of classical rewriting techniques (see section 1.1.1) rather difficult and limited. The used operators, however, are simple and restricted variants of relational operators which allows the use of inference rules to infer properties for intermediate results and operators, like key, cardinality, denseness, and functional and multivalued dependencies. Given the inferred properties of a single operator in the plan, predefined equivalence rules simplify it by pruning for example some obsolete input columns, and/or replace it with a simpler less expensive operator (*e.g.*, replace a join with a projection). The multivalued dependency property is used to detect the presence of an invariable item sequence $e$ inside a loop, which does not depend on the loop variable, and hence can be removed from the loop.

**Pattern Tree Minimization** In several XML database systems, pattern trees represent the basic structure to express and match path queries against the stored data. Therefore the efficiency of the query processing depends on the efficiency of matching the pattern tree which in turn depends among others on the size of the tree. The work in [3] optimizes XQueries by minimizing the size of its equivalent pattern trees. It proposes three polynomial time algorithms CIM, CDM and ACIM to eliminate redundant nodes in the pattern tree. CDM and ACIM are used in the presence of integrity constraints ($IC$) on the underlying data while CIM is employed when no early knowledge of the data is available. CIM is based on containment mappings and generates the unique equivalent minimal tree of a given pattern tree. Its polynomiality originates from two properties: a node is redundant only if its children are and the order of elimination of redundant nodes is immaterial. The application of ACIM and CDM is restricted to a certain class of ICs limited to required child, required descendant and required co-occurrence and under which the equivalent minimal query is unique. ACIM consists of first augmenting the pattern tree with redundant nodes and edges satisfying the given ICs and then applying CIM. To make ACIM faster, CDM is first applied to generate a locally minimal equivalent pattern tree by annotating each node with information content and propagating this information up the pattern tree. Their approach, however, does not consider value-based conditions and other type of constraints like required parent or ancestors.

## 3.2 Cardinality estimation and Cost Model

An accurate cost model is needed for an optimizer to choose the best plan. Cardinality estimation greatly affects the accuracy of the cost model. Therefore a lot of research was devoted to the problem of cardinality estimation of XPath expressions in XML databases.

### 3.2.1 Cardinality Estimation

The main difference in cardinality estimation between relational (section 1.2) and XML databases lies in the fact that path queries specify structure constraints in addition to value constraints. Thus the optimizer should collect statistics about both value distribution and structural relationships between elements. This section describes techniques developed to summarize XML documents into synopses. Synopses are evaluated based on the following criteria: Is the synopsis' construction algorithm expensive? What XQuery/XPath subset does the synopsis cover? Does the synopsis produce accurate estimations? Does the synopsis support updates to the database? Does the synopsis support value constraints? Is the synopsis recursion-aware? Does it also estimate the distribution of the data?

An early work on selectivity estimation of simple path expressions found in [12] estimates the number of twig matches in a node-labeled tree by using a summary data structure referred to as Correlated Subpath Tree (CST). The problem of cardinality estimation on twigs is reduced to the problem of estimating substring selectivity. The proposed technique stores count statistics about frequently occurring subpaths and maintains the correlation among the subpaths sharing the same root. The disadvantages of this approach are that the whole CST must be built before being pruned, and it does not handle wildcards.

The work in [1] proposes two different synopses, path trees and Markov tables, to summarize XML documents. A path tree represents the structure of an XML document where path tree vertices are associated with the cardinality of these nodes in the document. Markov tables store cardinality statistics for subpaths of lengths up to a certain value. The selectivity of longer paths is estimated by combining the cardinality statistics of several subpaths. Path trees and Markov tables may grow large and hence can be summarized by compressing some non frequent vertices and subpaths and replacing them by wildcards nodes and *-paths correlated with information about the deleted statistics. This technique supports only simple linear path queries.

StatiX [22] is a framework providing selectivity estimation for XQueries in the presence of an XML schema by transforming the schema such that statistics are collected at different levels of granularity. The approach uses a histogram to maintain information on both the structure and values in the XML document. The application of this technique, however, is restricted to a small subset of the XQuery grammar.

The technique proposed in [56] builds a two-dimensional position histogram based on the start and end labels assigned to nodes using a certain numbering scheme. The histograms are used to estimate the result sizes of path expressions that use descendant and/or ancestor steps only and can not be adopted for more general path expressions.

The Bloom histogram synopsis proposed in [55] provides efficient and accurate cardinality estimation for XPath expressions and supports updates on the underlying XML database by reflecting any change in the document through updates to the dynamic summary component. This approach, however, can handle only simple path expressions.

The XSketch synopsis proposed in [43] estimates the selectivity of complex XPath expressions over graph-structured XML data by capturing the key structural information (i.e., label path and branching) in the graphs. The construction algorithm of the XSketch synopsis employs a heuristic based on greedy forward selection. It generates a label-split graph from the XML tree by merging all XML nodes with the same label into one vertex, and then successively refines the graph by exploiting localized backward and forward stability properties in the graph. The synopsis is built using a sample of path queries which makes it dependent on the generated set. The work in [44] augments each node in the structural XSketch synopsis with distribution information on the element values in the XML graph to estimate the cardinality of path queries that also contain value-based constraints.

TreeSketch [45], another synopsis developed by the same authors, also employs a structural clustering technique. Unlike XSketch, it is based on count-stability which is a refinement of forward stability. The construction of the synopses starts by building a count-stable summary graph of an XML data tree and then incrementally merges element clusters that are closely similar in their subtree structure until the memory budget is met. XSketch and TreeSketch present two shortcomings: the construction time of the synopsis is high and updating it is also expensive. TreeSketch, however, has one advantage over XSketch: it is orders of magnitude more accurate in estimating results cardinality and needs less time to construct.

The work in [19] describes another synopsis for XML documents to estimate the selectivity of path expressions. Their approach supports branching XPath queries including all axis types. The construction of the synopsis lies in translating the XML document to an SLT (Straight Line Tree) grammar by using a tree compression algorithm. An XPath query is then converted into tree automata which in combination with the SLT grammar can be used to estimate the results of the query. The authors claim their approach has several advantages over the existing approaches: the construction of the synopsis is less expensive, it can handle updates, produces better estimates and handles all XPath axis.

Except for [19, 55], all existing approaches can not handle updates to the underlying database. Only two of the proposals described above [22, 43] support XPath queries with value constraints. None of the existing approaches perform well on recursive data sets, that is if they are recursion-aware, and none can provide an estimation on the result's distribution.

### 3.2.2 Cost Models

Developing cost models for operators must take into consideration, among other parameters, the physical layout of data, the way this data is retrieved from disk, the amount of memory available, and the algorithm implementing the operators' functionality. The fact that XML operators are much more complex due to the nature of the XML data renders the prediction and modeling of the data access a hard process. Therefore constructing an accurate cost model for XML query processing is far more difficult than developing cost models for relational databases as described in section 1.2. Most research on cost-based XML optimization has only focused, as we have seen in the previous section, on the problem of cardinality estimation. Comet, described in [59], is one of the few approaches known for defining a cost model for XML.

Comet is a statistical learning technique that can be used to model the CPU cost of complex XML operators. First a set of queries and data features critical in determining the cost of the operator need to be identified, then using statistics and analytical formulas feature values are estimated. Finally Comet employs the transform regression method to learn the functional relationship between the feature values and the operator's cost. The resulting cost function is then used to estimate the cost of the operator and can be updated through a process of query feedback to adapt to changes in query workload and system environment. However this approach can only estimate the CPU cost of operators and can not be used for determining I/O cost.

## 3.3 Plan Enumeration and Selection

Similar to the relational plan enumeration algorithms presented in section 1.3 whose main focus in on enumerating different join orders, execution plan enumeration in Timber consists of enumerating the join orders in the plan. The work in [57] proposes five cost-based enumeration techniques which explore the space of execution plans by reordering join operators and guarantee the choice of the optimal or suboptimal plan. The difference between the proposed algorithms is the number of plans generated for choosing the optimal one, (*i.e.*, the time spent on the enumeration of plans), and the certainty of picking the optimal plan at the end. The first two algorithms, *Dynamic Programming* and *Dynamic Programming with Pruning*, enumerate respectively all or a subset of equivalent plans and make sure the optimal one is chosen. The next two algorithms apply some pruning techniques to decrease the search space but might fail in finding the optimal plan. The fifth algorithm reduces the search space by only considering fully-pipelined plans. The latter three algorithms trade off the optimality of the chosen plan for the time spent on enumeration. The authors conclude that the choice of the enumeration algorithm should depend on the query being executed: if the query is not too expensive, then the fully-pipelined algorithm can find a good plan in a short time. If the query's execution is slow then it is best to use the *Dynamic Programming with Pruning* algorithm to produce the optimal plan.

The work in [32] proposes three different plans for evaluating XPath expressions each using one type of navigational primitive. The first technique translates every location step in the path to an Unnested-Map operator while the second and third group operations requiring expensive I/O access to the disk into a single operator, either a XScan or a XSchedule operator. The two operators are responsible for scheduling and managing inter-cluster operations such that the time spent accessing the physical layer is minimized, i.e., reducing the number of times a page is loaded from hard disk, and optimizing the order in which pages are accessed. XSchedule employs an asynchronous I/O while XScan involves a sequential scan to the data. The notion of partial path instances which represent the incomplete evaluation of XPath expressions due to pending I/O is introduced. It facilitates the separation between cheap and expensive I/O operations, and helps in avoiding random physical accesses: only nodes belonging to the same cluster are looked up, while access to other clusters is deferred and grouped into the XScan or XSchedule operator. The results show that XSchedule and XScan almost always outperform the simple method, while XSchedule outperforms XScan if the query is highly selective.

# Chapter 4

# Open Problems

Although a considerable amount of work has been done in the optimization of XML queries, several issues are still unresolved. We enumerate, in this section, some of the existing open problems.

## 4.1 Equivalence Rewriting

### 4.1.1 XPath Rewriting

As XPath is a central expression in XQuery, its evaluation performance has a big impact on the overall performance of the XQuery itself. Therefore one possibility to optimize the evaluation of XQueries is by rewriting XPath expressions into equivalent more efficient ones. Some work for syntactically rewriting XPath expressions, *e.g.* [40], has already been done; however, rewriting at this level presents three disadvantages. First it can only find rewritings that are expressible in XPath, hence missing some possible more efficient execution plans. Moreover, it is harder for the optimizer to use and benefit from the available statistics. Finally, even if statistical knowledge is used, applying the defined heuristic at this level, then translating the XQuery core plan to an algebra, and afterwards applying algebraic optimization rules might lead to unpredictable plans. The work in [48] showed that in some situations the decisions taken by query optimizers can be very unpredictable and the assumptions it makes do not always hold. Consequently we think a better choice that presents several advantages is to exploit the XPath symmetries at the algebra level where the application of rewriting rules can be synchronized with other optimization rules.

### 4.1.2 Element Construction Optimization

It is common in XQuery to query XML fragments that were constructed by the query itself. The fact that element construction is an expensive operation that often requires the copying of a complete XML subtree, gives rise to the XQuery optimization question: how can we remove or reduce, if possible, intermediate

XML fragment construction or push computation through the construction process. We illustrate this problem with the following simple XQuery example:

```
declare function local:foo($e as element()) as element()
{
<a>{$e}</a>
};
local:foo(<b>Hello World!</b>)//b
```

In this example, the construction of the XML element <a> is unnecessary, consequently the call to the function *foo* and the navigation to the *b* elements can be removed. A query rewriter module would like to find such unnecessary constructions and navigation and eliminate them. But if the navigation in the previous example attempts to access in the results returned by the *foo* function the element *a*, then this optimization is not possible and will lead to wrong results. It is also an open question whether it is better to perform this rewriting at the XQuery core or at the algebra level.

### 4.1.3 User Defined Recursive Function Optimization

XQuery is employed not only as a query language but also as a programming language, therefore it is not surprising to see that in different types of applications recursive functions are being defined by users to query XML data. Several research proposals have tackled the problem of optimizing user-defined functions in XQuery; however, little work focused on recursive functions. One of the techniques used to minimize the overhead of user-defined functions is inlining (*i.e.*, replacing the function call by the function code). This approach can be adopted for a specific class of recursive functions only in the presence of schema information, and can not be applied in all other cases since it is undecidable a priori when to break the recursion as this, in general, depends on the value of the passed data which can not be checked by the rewriter. A question arises here: can we perform this optimization at the algebra level and if so which recursive XQuery function definitions can be represented algebraically and what are the possible techniques to optimize these generated plans.

## 4.2 Cardinality Estimation and Cost Model

To make sure the rewriting phase will result in generating faster plans, the decisions made during the rewriting process should be based on collected statistics and a cost model. Since the accuracy of cardinality estimation greatly impacts the accuracy of the plan's cost estimation, a lot of research has been done to develop synopses to estimate the cardinality for XPath expressions. Each technique described in Section 3.2.1 has its own shortcomings, and thus there is a need for a better, compact synopsis that meets the following criteria: returns accurate estimations, covers a large subset of XQuery and not only XPath expressions, handles efficiently updates to the document and recursive data, is constructed at a low price, and supports queries with value constraints. Furthermore it is beneficial to complement the proposed approach for cardinality estimation with a technique that estimates the distribution of the returned data, which will result in more accurate predictions of the result size of subsequent operations. In fact this information is highly valuable for operators whose cardinality estimation greatly depends on the distribution of their input data like for example aggregation, and selection.

An accurate cost model is essential for selecting a good query plan. Little research has been done for modeling the cost of operators in XML database systems. The only proposed approach known to us that addresses this issue is Comet [59]; however, it only estimates CPU costs. Hence, a suitable cost model that takes into account both the CPU and I/O costs of operators in an XML database is needed.

## 4.3 Plan Selection

At compile time, several parameters such as size of input, selectivity and available resources are estimated by the optimizer in order to make its choice of the plan to be executed. The accuracy of these estimations is not always guaranteed and hence the static plan chosen by the optimizer is not always optimal or close to optimal. A technique called dynamic plan selection, which has been proposed in the context of relational database systems, postpones the selection of the best plan to runtime. According to our knowledge, no research has been done on this topic in the context of XML databases. Since predicting input cardinality and operator selectivity is even harder for XML than relational data, we think one interesting approach is to see to which extent dynamic plan selection can be applied for XML. With this technique, the plan generated by the optimizer will contain several subplans connected by a special operator (*e.g.*, a choice operator), where a choice between the equivalent subplans is made at runtime. In fact, MonetDB/XQuery presents a good platform to experiment with this approach since it supports materialization of intermediate results. The choice to shift to a better and semantically equivalent subplan can be made by considering, among other things, the size of intermediate results. Several questions arise when adopting this technique, three of which are: how is the dynamic plan created? Which optimization decisions are postponed to runtime? How effective is this technique in practice?

# Conclusion

We gave, in this report, an overview of query optimization techniques in XML database systems. To achieve this objective, we started by familiarizing the reader with the traditional optimization techniques employed in the context of relational database systems, and by enumerating and describing some of the existing XML database systems. A lot of work has been done for optimizing plans in relational systems and the techniques presented in this document are only a small subset. Despite many years of effort, there still exist significant open problems to be tackled. We have also reviewed some existing optimization techniques in XML databases such as syntactic and algebraic rewriting, cost modeling, and plan selection. Studying the techniques proposed for optimizing queries in XML database systems indicates that a lot of work has to be done in this field to reach the maturity of its relational counterpart. We have identified some problems in the XML context that present a potential for optimization, of which we mention XPath rewriting, element construction and user defined recursive function optimization. The need for an accurate cost model and an efficient plan enumeration algorithm is also identified. It is worth to mention that the big variety in existing XML engines regarding their storage model, data model and algebra increases the difficulty in defining XML optimization techniques that can be adopted by all systems.

# Bibliography

[1] Ashraf Aboulnaga, Alaa R. Alameldeen, and Jeffrey F. Naughton. Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. In *27th International Conference on Very Large Data Bases (VLDB'01)*, pages 591–600, 2001.

[2] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. *18th International Conference on Data Engineering (ICDE'02)*, 2002.

[3] Sihem Amer-Yahia, SungRan Cho, Laks V. S. Lakshmanan, and Divesh Srivastava. Minimization of Tree Pattern Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 497–508, 2001.

[4] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernandez, Michael Kay, Jonathan Robie, and Jerome Simeon. XML Path Language (XPath) 2.0. *http://www.w3.org/TR/xpath20/*, 2007.

[5] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, and Jerome Simeon. XQuery 1.0: An XML Query Language. *http://www.w3.org/TR/xquery/*, 2007.

[6] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2006.

[7] Matthias Brantner, Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. Full-Fledged Algebraic XPath Processing in Natix. *21st International Conference on Data Engineering (ICDE'05)*, 2005.

[8] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and Franois Yergeau. Extensible Markup Language (XML) 1.0. *http://www.w3.org/TR/2006/REC-xml-20060816/*, 2006.

[9] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 310–321, 2002.

[10] S. Chaudhuri. An Overview of Query Optimization in Relational Systems. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 34–43, 1998.

[11] Zhimin Chen, H. V. Jagadish, Laks V. S. Lakshmanan, and Stelios Paparizos. From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery. *29th International Conference on Very Large Data Bases (VLDB'03)*, 2003.

[12] Zhiyuan Chen, H. V. Jagadish, Flip Korn, Nick Koudas, S. Muthukrishnan, Raymond T. Ng, and Divesh Srivastava. Counting Twig Matches in a Tree. In *17th International Conference on Data Engineering (ICDE'01)*, pages 595–604, 2001.

[13] David DeHaan, David Toman, Mariano P. Consens, and M. Tamer Ozsu. A Comprehensive XQuery to SQL Translation Using Dynamic Interval Coding. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2003.

[14] D. Draper, P. Frankhauser, M. Fernandez, A. Malhotra, K. Rose, M. Rys, J. Simeon, and P. Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. *W3C working draft, http://www.w3.org/TR/xquery-semantics/*, 2004.

[15] eXist Open Source Native XML Database. http://exist.sourceforge.net/.

[16] Mary Fernandez, Jerome Simeon, Byron Choi, Amelie Marian, and Gargi Sur. Implementing XQuery 1.0: The Galax Experience. *29th International Conference on Very Large Data Bases (VLDB'03)*, 2003.

[17] Mary F. Fernández, Jan Hidders, Philippe Michiels, Jérôme Siméon, and Roel Vercammen. Optimizing Sorting and Duplicate Elimination in XQuery Path Expressions. In *16th International Workshop on Database and Expert Systems Applications (DEXA'05)*, pages 554–563, 2005.

[18] T. Fiebig, S. Helmet, K.-C. Kanne, G. Moerkotte, J. Neumann, and R. Schiele. Anatomy of a Native XML Base Management System. *28th International Conference on Very Large Data Bases (VLDB'02)*, 2002.

[19] Damien Fisher and Sebastian Maneth. Structural Selectivity Estimation for XML Documents. In *23rd International Conference on Data Engineering (ICDE'07)*, 2007.

[20] Daniela Florescu, Chris Hillery, Donald Kossmann, Paul Lucas, Fabio Riccardi, Till Westmann, Michael J. Carey, and Arvind Sundararajan. The BEA streaming XQuery processor. *30th International Conference on Very Large Data Bases (VLDB'04)*, 13(3):294–315, 2004.

[21] Simple API for XML. http://www.saxproject.org/.

[22] Juliana Freire, Jayant R. Haritsa, Maya Ramanath, Prasan Roy, and Jérôme Siméon. StatiX: Making XML Count. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 181–191, 2002.

[23] T. Grust and J. Teubner. Relational Algebra: Mother Tongue - XQuery. In *Twente Data Management Workshop on XML Databases and Information Retrieval (TDM 2004)*, 2004.

[24] Torsten Grust. Accelerating XPath location steps. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 109–120, 2002.

[25] Torsten Grust. Purely Relational FLWORs. In *2nd International Workshop on XQuery Implementation, Experience and Perspectives (XIME-P'05)*, 2005.

[26] Torsten Grust and Maurice van Keulen. Tree Awareness for Relational DBMS Kernels: Staircase Join. In *Intelligent Search on XML Data*, pages 231–245, 2003.

[27] Torsten Grust, Maurice van Keulen, and Jens Teubner. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In *29th International Conference on Very Large Data Bases (VLDB'03)*, 2003.

[28] Torsten Grust, Maurice van Keulen, and Jens Teubner. Accelerating XPath evaluation in any RDBMS. *ACM Transactions on Database Systems (TODS'04)*, 29:91–131, 2004.

[29] Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. An XML Query Engine for Network-Bound Data. *28th International Conference on Very Large Data Bases (VLDB'02)*, 2002.

[30] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER: A Native XML Database. *28th International Conference on Very Large Data Bases (VLDB'02)*, 2002.

[31] H. V. Jagadish, Laks V. S. Lakshmanan, Divesh Srivastava Divesh, and Keith Thompson. TAX: A Tree Algebra for XML. *8th Workshop on Data Bases and Programming Languages (DBPL'01)*, 2001.

[32] Carl-Christian Kanne, Matthias Brantner, and Guido Moerkotte. Cost-Sensitive Reordering of Navigational Primitives. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 742–753, 2005.

[33] Michael Kay. XSL Transformations (XSLT) Version 2.0. *http://www.w3.org/TR/2007/REC-xslt20-20070123/*, 2007.

[34] Christoph Koch, Stefanie Scherzinger, Nicole Schweikardt, and Bernhard Stegmaier. FluXQuery: An Optimizing XQuery Processor for Streaming XML Data. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB'04)*, pages 1309–1312, 2004.

[35] Michael Kifer; Arthur Bernstein; Philip M. Lewis. *Database Systems: An Application Oriented Approach.* Addison-Wesley, 2005.

[36] B. Ludascher, P. Mukhopadhyay, and Y. Papakonstantinou. A Transducer-Based XML Query Processor. *28th International Conference on Very Large Data Bases (VLDB'02)*, 2002.

[37] Norman May, Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. XQuery Processing in Natix with an Emphasis on Join Ordering. In *Proceedings of the 1st International Workshop on XQuery Implementation, Experience and Perspectives (XIME-P'04)*, pages 49–54, 2004.

[38] Norman May, Sven Helmer, and Guido Moerkotte. Nested Queries and Quantifiers in an Ordered Context. *Proceedings of the 20th International Conference on Data Engineering (ICDE'04)*, pages 239–250, 2004.

[39] Ramez Elmasri; Shamkant B. Navathe. *Fundamentals of Database Systems.* Addison-Wesley, 2004.

[40] Dan Olteanu, Holger Meuss, Tim Furche, and François Bry. XPath: Looking Forward. In *Proceeding of the EDBT Workshop on XML Data Management (XMLDM)*, pages 109–127, 2002.

[41] Oracle Berkeley DB XML. http://www.oracle.com/database/berkeley-db/xml/index.html.

[42] Stelios Paparizos, Yuqing Wu, Laks V. S. Lakshmanan, and H. V. Jagadish. Tree Logical Classes for Efficient Evaluation of XQuery. *ACM SIGMOD International Conference on Management of Data*, 2004.

[43] Neoklis Polyzotis and Minos N. Garofalakis. Statistical Synopses for Graph-Structured XML Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 358–369, 2002.

[44] Neoklis Polyzotis and Minos N. Garofalakis. Structure and Value Synopses for XML Data Graphs. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB'02)*, pages 466–477, 2002.

[45] Neoklis Polyzotis, Minos N. Garofalakis, and Yannis E. Ioannidis. Approximate XML Query Answers. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 263–274, 2004.

[46] Qizx. http://www.axyana.com/qizxopen/.

[47] Christopher Re, Jerome Simeon, and Mary Fernandez. A Complete and Efficient Algebraic Compiler for XQuery. *22nd International Conference on Data Engineering (ICDE'06)*, 2006.

[48] Naveen Reddy and Jayant R. Haritsa. Analyzing plan diagrams of database query optimizers. In *31th International Conference on Very Large Data Bases (VLDB'05)*, pages 1228–1240, 2005.

[49] Saxonica: XSLT and XQuery Processing. http://www.saxonica.com/.

[50] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Heuristic and Randomized Optimization for the Join Ordering Problem. *The International Journal on Very Large Data Bases*, 6(3):191–208, 1997.

[51] Hong Su, Jinhui Jian, and Elke A. Rundensteiner. RAINDROP: A Uniform and Layered Algebraic Framework for XQueries on XMLStreams. *International Conference on Information and Knowledge Management (CIKM)*, 2003.

[52] Arun N. Swami and Anoop Gupta. Optimization of Large Join Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 8–17, 1988.

[53] I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. Shekita, and C. Zhang. Storing and Querying Ordered XML Using a Relational Database System. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 204–215, 2002.

[54] Attila Barta Vanja Josifovski, Marcus Fontoura. Querying XML streams. *The International Journal on Very Large Data Bases*, pages 197–210, 2005.

[55] Wei Wang, Haifeng Jiang, Hongjun Lu, and Jeffrey Xu Yu. Bloom Histogram: Path Selectivity Estimation for XML Data with Updates. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB'04)*, pages 240–251, 2004.

[56] Yuqing Wu, Jignesh M. Patel, and H. V. Jagadish. Estimating Answer Sizes for XML Queries. In *Proceedings of the 8th International Conference on Extending Database Technology (EDBT'02)*, pages 590–608, 2002.

[57] Yuqing Wu, Jignesh M. Patel, and H. V. Jagadish. Structural Join Order Selection for XML Query Optimization. In *Proceedings of the 19th International Conference on Data Engineering (ICDE'03)*, pages 443–454, 2003.

[58] X-Hive. http://www.x-hive.com/products/db/index.html.

[59] Ning Zhang, Peter J. Haas, Vanja Josifovski, Guy M. Lohman, and Chun Zhang. Statistical Learning Techniques for Costing XML Queries. In *Proceedings of the 31th International Conference on Very Large Data Bases (VLDB'05)*, pages 289–300, 2005.