Holistic Top-\(k\) Query Processing for XQuery Full-Text

Masterarbeit im Fach Informatik
Master’s Thesis in Computer Science
von / by
Fisnik Kastrati

angefertigt unter der Leitung von / supervised by
Prof. Dr. Matthias Jarke
Prof. Dr. Gerhard Weikum

betreut von / advised by
Dr. Martin Theobald
Dr. Christoph Quix

Saarbrücken, Aachen
February 2010
Non-plagiarism Statement

Hereby I confirm that this thesis is my own work and that I have documented all sources used.

Aachen, 11. February 2010

(Fisnik Kastrati)

Declaration of Consent

Herewith I agree that my thesis will be made available through the library of the Computer Science Department.

Aachen, 11. February 2010

(Fisnik Kastrati)
God may not play dice with the universe, but something strange is going on with the prime numbers.

—Pál Erdős
Acknowledgments

This thesis project could not have been completed without Dr. Theobald who not only served as my advisor, but also encouraged and challenged me throughout my master’s thesis project. Dr. Theobald guided me through this master’s thesis project and the writing of this thesis, never accepting less than my best efforts. His guidance and support from the initial to the final level enabled me to develop an understanding of the subject. I thank him very much.

I would like to show my gratitude towards Dr. Quix for advising me throughout the course of this project.

I am deeply indebted to my supervisors Prof. Jarke and Prof. Weikum for making it possible to do the necessary research work for this thesis project at the renowned Max-Planck-Institute for Computer Science in Saarbrücken.

I’m very thankful to my parents for their emotional support throughout the course of my master studies, their support has been decisive towards completion of my master studies.

Furthermore, many thanks go to the DAAD for their financial support.
Abstract

Querying large data sets is a challenging task in today’s information systems. Users are typically interested in the $k$ most relevant results, namely the first page (e.g., the Google search engine) of the given result set. That is, given a dataset $D$, and user defined similarity function $f$, we are interested in calculating the top-$k$, i.e., the $k$ highest ranked results (answers). Finding the top-$k$ most relevant results is one side of the medal, in this domain query language plays an equally important role, as it is basically the main component for user interaction with the system.

We append a new query language - XQuery Full-Text [13] to the existing set of query languages supported by the TopX [37] search engine. XQuery Full-Text poses new challenges, as the evaluation of XQuery in top-$k$ fashion is a challenging task due to its complexity and expressiveness.

We tackle the complexity of XQuery evaluation by introducing a new novel approach in top-$k$ computation, namely operator based query evaluation, with holistic top-$k$ computation, and the latter is the main contribution of this thesis work. We break down XQuery into multiple nested XPath [12] queries, where each nested XPath query is assigned to an independent operator for evaluation. Operators are part of the system as whole, where each operator shares the workload, and contributes in fast and flexible top-$k$ result computation. Moreover, our operator-based environment offers flexibility in terms of computing top-$k$ results in parallel due to operator independence abstraction, this way increasing efficiency, flexibility, robustness and last, but not least, the response time.

This architecture is novel in the sense that it increases the extensibility of the overall architecture and the robustness. The same architecture would fit well in distributed systems, making it easy to share the workload into distributed nodes, and having a super-node acting as broker for top-$k$ computation, and coordinating the execution of the query.
Contents

1 Introduction .......................................................... 1
  1.1 Motivation ......................................................... 1
  1.2 Contributions ................................................... 3
  1.3 Outline .......................................................... 5

2 Related Work .......................................................... 7
  2.1 Overview on related work and comparison to our approach ... 7
    2.1.1 Index-based XXL Search Engine .......................... 7
    2.1.2 FleXPath ...................................................... 8
    2.1.3 Pathfinder .................................................... 9
    2.1.4 PF/Tijah ....................................................... 10
    2.1.5 GalaTex ....................................................... 10
    2.1.6 BaseX ........................................................ 11
    2.1.7 Threshold Algorithms .....................................

3 Existing Infrastructure .............................................. 15
  3.1 TopX 2.0 .......................................................... 15

4 Data and Query Model ............................................... 21
  4.1 Data Model ........................................................ 21
    4.1.1 Extensible Markup Language 1.0 .......................... 21
  4.2 Query Model ....................................................... 23
    4.2.1 XPath 2.0 Full-Text ........................................ 24
    4.2.2 XQuery 1.0 with Full-Text Extensions .................. 25
    4.2.3 Narrowed Extended XPath I (NEXI) ...................... 27
    4.2.4 TopX Query Representation and Query Parsing ........ 28
    4.2.5 XQuery Parser ................................................ 29

5 Relevance Scoring Model .............................................. 33
List of Figures

1.1 XML tree distinction ................................. 3
2.1 Different modes of index list accesses ............... 12
3.1 TopX architecture .................................. 17
4.1 XML Data Tree ..................................... 22
4.2 Twig pattern ........................................ 25
4.3 XQuery Full Text Constructs ....................... 30
4.4 XQuery flow-chart diagram ......................... 32
5.1 Data Model ........................................... 36
5.2 DAG Representation of the query .................. 37
5.3 Index Structure ...................................... 39
6.1 Overview of TopX\textsuperscript{XQ} .................. 42
6.2 Top-\(k\) and the candidate queue ................ 46
6.3 XQuery architecture ................................ 46
6.4 Inverted Index Structure comparison ............. 49
7.1 Query Evaluation .................................... 57
7.2 Min-\(k\) threshold increase ......................... 62
8.1 Query results comparison between queries evaluated in single and multiple operator mode ......... 67
8.2 Query results for queries starting with the \textit{article} structural constraint ........................................... 71
8.3 Query results for queries starting with the \textit{*} structural constraint ........................................... 72
8.4 Total run-times comparison for single and multiple operator mode ...................................................... 73
8.5 Total run-times comparison for single and multiple operator mode ...................................................... 73
Chapter 1

Introduction

1.1 Motivation

In recent years, semistructured data repositories have emerged from many disciplines, and XML \cite{11} was adopted by many organizations as the de-facto standard for data storage and data-exchange. Bibliographical databases such as DBLP \cite{40}, IEEE, the ACM SIGMOD Record collection, or the XML version of the Wikipedia encyclopedia used in the INEX benchmark series \cite{29}, are just a few examples of semistructured data repositories. These repositories combine text with rich semantic and structural annotations. With such ubiquitous repositories, the demand for querying and answer retrieval is apparent.

Beginning with XPath 2.0 \cite{12} and XQuery 1.0 \cite{13}, one can seamlessly query XML repositories that are rich in structure. These two languages are very expressive database-style (DB) languages, but not originally designed for querying text-dense semistructured repositories. Querying in strict database-style yields answers only when all query conditions are matched. DB-style or conjunctive querying with boolean search predicates, when applied to semistructured repositories, often results with either too many or too few results.

In database-style querying, the underlying structure of the XML data should be known to the user a-priori. It is usually the case that the underlying structure of the XML database is unknown or of no interest to the user, therefore automatic invocation of query relaxation can serve as helpful technique in overcoming such issues.
Application of IR techniques on query evaluation would call for a form of approximative query answering or “fuzziness”, that is, the results are returned according to some ranked-retrieval paradigm and do not strictly adhere to the exact-match semantics. When too few results are returned, the IR system relaxes the query conditions, in order to increase the recall.

Query relaxation is a well-know technique in the information retrieval (IR) community. It is used to weaken the query requirements when too few or no results are returned. This entails that in IR-style, the results are not strictly required to match all the constraints in order to be qualified as valid results.

Of interest to this work is the non-conjuctive – IR-style of querying, in particular XQuery with full-text search primitives such as ftcontains [14], or the “about” operator used in NEXI [41] language. The latter is a path oriented language, it permits only descendant ‘//’, and self axes ‘.’. For illustration purposes, a simple NEXI-style query is shown bellow:

//article//section[about(., XQuery)]

This query looks for articles that have section elements, which in turn contains the keyword “XQuery”.

Full-text search is an extension to the XQuery query language, offering text-search primitives that enables IR-style search over semistructured repositories. In addition, XQuery 1.0 and XPath 2.0 Full-Text [14] are set to become a W3C standard. The full-text extension to the XPath 2.0 and XQuery 1.0 languages has instituted the IR world in the path-oriented search paradigm. XPath and XQuery are powerful and very expressive query languages in a similar sense as the SQL is to the relational databases.

Fig. [1.1] a) and b) depict two trees with slightly different structure, but they both sketch the same article element in terms of textual content. However, with DB-style queries, and their implied “exact-match” semantics, it is a non-trivial task to show that these two article elements are in fact equally relevant. This is not the case when queries are evaluated in the IR-style. By making use of query relaxation, the non-matching structural constraints can be relaxed, and the same query would result with answers in both of the trees. However, this does not imply that in IR style, structure plays no role, in fact items that match fewer structural constraints rank lower in the resulting list of answers, for the reason that such items don’t gain any score for the non-matched constraints (i.e., structural constraints).
TopX \cite{37} is a top-$k$ retrieval engine that supports query evaluation in IR style over non-schematic XML repositories \cite{37} and XPath-like queries with full-text search. For this thesis work, TopX serves as implementation basis with the goal of implementing a subset of the XQuery Full-Text query language in a top-$k$ aware system – TopX$^{XQ}$.

### 1.2 Contributions

The current version of TopX 2.0 supports only one XPath expression per query. We extend this functionality of TopX to XQuery Full-Text with multiple XPath expressions towards a – “holistic” top-$k$ processing for XQuery with multiple XPath-Full-Text conditions.

We propose a genuine technique for top-$k$ computation over a subset of XQuery expressions with full-text search predicates. We present a technique for decomposing XQuery expressions $q_i$ into multiple XPath expressions $q_i = (x_1, \ldots, x_n)$. The system computes the overall top-$k$ elements on top of the nested XPath expressions $x_i$. Each $x_i$ is processed independently as a nested operator, and the scores for each of the $x_i$ operators are propagated upwards to their next immediate parent operator. We provide a multi-threaded architecture for enabling parallel execution of each $x_i$ operator. Although, each operator runs in parallel, our architecture forms a tree of operators, where each operator has one parent operator, except the top operator, and parent operator can have many nested operators in turn. Following this suite, nested operators periodically report scores for their items.
to their parent operator. Nested operators are light-weight operators, therefore they need not compute top-k items, nor score bounds for their items in the queue, which otherwise would have been a major burden in terms of memory requirements.

Global scores for the top-k items are calculated by applying monotonic score aggregation functions, and finally we apply Fagin’s variant of TA [18] algorithm – NRA (see Sec. 2.1.7), for threshold test and search space pruning. Score aggregation generalized well in our operator based infrastructure, where non-leaf operator aggregates scores over score bounds, while score bounds are provided by the nested operators. This way of score aggregation is still monotonic, with distinction that instead of working directly with scores fetched from the index lists, we work over the score bounds (i.e., best score, and worst score bounds provided by the nested operators).

Since score aggregation over bounds (i.e., scores that nested operators reported) is still monotonic, we can then infer that the score of an item $T'$ is at least as high as the score of $T$, if the monotonic function $t(s_1, \ldots, s_n) \leq t'(s'_1, \ldots, s'_n)$ holds, that is, whenever $s_i \leq s'_i, \forall i$. This way NRA algorithm is extended to the operator architecture without violating it’s instance optimality.

Our contributions can be summarized as follows:

- **XQuery Full-Text Parsing Infrastructure** (Section 4.2.4) We produce DAG based query patterns for both conjunctive and non-conjunctive search. DAG patterns are passed from the query parser via an interface to the query processor for query evaluation in parallel, by assigning each pattern to an operator, thus constructing an operator tree.

- **XQuery decomposition into a number of XPath expressions** (Section 4.2.5) We break XQuery expressions into a number of XPath expressions, where each XPath expression is further broken into a number of atomic structural and content constraints for efficient query evaluation. For each such constraint the appropriate index lists are scanned in distributed fashion.

- **Operator based XPath execution in parallel** (Section 6.3) We process queries in parallel in our operator based infrastructure, where each operator is responsible for handling one XPath expression. To this end, we differentiate between two types of operators, that is, one top operator and a number of nested operators.
• “Holistic” optimization of XQuery (Chapter 6) Our query processing infrastructure allows for holistic optimization with top-k-style early pruning across multiple operators. Operators feed with score bounds their parent operator, this way the scores reach the top operator, which in turn maintains the score bounds, and accordingly terminates the algorithm safely (i.e., when score bounds permit the termination in top-k style).

• Instance Optimality (Section 6.4) We show that our operator based query processing infrastructure is instance optimal with small constant overhead in worst case scenario.

1.3 Outline

This thesis work is organized as follows: In chapter (2), we elaborate the related work, that is, we present some existing systems and compare them with ours, as well as the family of threshold algorithms that are widely used for ranked-retrieval query evaluation. In chapter (3): the existing TopX [37] system and the underlying architecture are explained. In chapter (4): we present the query and the data model that our system supports. Starting with XML document structure, and continuing with different query types that we support, and last but not least, in this chapter we explain our parser, and our parsing infrastructure. In chapter (5): we present the scoring model used in our system for ranked retrieval, the XML specific scoring model, which is used for our query evaluation.

Chapter (6) is the main chapter of this thesis work. In this chapter we show our query evaluation strategies for our operator based architecture. We explain in details the operator based architecture, and we conclude this chapter with the proof of instance optimality. In chapter (7): we show the inner details of the system, as well as query processing from the initial steps until the final results are computed. This chapter involves technical details, such as operator API, which is explained too.

Chapter (8) is about the concluded experiments and the final results, to continue with chapter (9) where we present the conclusions and the future work.
Chapter 2
Related Work

2.1 Overview on related work and comparison to our approach

2.1.1 Index-based XXL Search Engine

XXL [34] is one of the earliest XML search systems that supported ranked retrieval. XXL supports pattern matching along paths in XML documents, furthermore XXL supports ontological reasoning. A nice feature of XXL is that it exploits XML annotations and structural information, that is, it does not adhere to only boolean search paradigm.

XXL models XML documents as data graph, and evaluates queries by traversing this data graph, furthermore it supports XLink and XPointer, which are represented in the datagraph with edges, but not only XLink and XPointer, XXL models relationships among elements by using edges too.

The XXL evaluates queries based on its three precomputed index structures (1) element path index, (2) element content index and (3) ontology index. These indexes are constructed and maintained as the XML repositories are crawled. Path patterns in the XXL are evaluated by using element path index, whereas content conditions are evaluated by using element content index.

For improving the quality of similarity search, the XXL tries to exploit ontological information, by considering element names that were accumulated during indexing as the most semantically expressive components of XML
documents. For this purpose, the XXL maintains an ontology index as mentioned above. The ontology index organizes all the element names into an ontological graph, for details on how this index is constructed please see [34]. The XXL query processor consults the ontology index for term expansions. For a given expression containing an element “e” in the form “∼e”, that element is expanded in disjunctive paths expressions t_{1}, \ldots, t_{k}, that is, k terms were returned from the ontology index, each with relevance score s_{1}, \ldots, s_{n} to the element “e”. Then this expanded expressions “e” will be evaluated by the XXL query processor by invoking element path index. We mentioned the ontology index in the XXL system in more details, as it was one of the earliest systems to implements such index in order to improve the quality of search, that is, users do not always type their queries with the best keywords, and an ontology index in such cases is a helpful tool for increasing the quality of the search results.

2.1.2 FleXPath

FleXPath [3] is another XML search engine, which combines boolean and IR-style query evaluation. That is, FleXPath supports db-style querying such as XPath with exact-match semantics, and IR-style querying with ft-contains full-text search predicates. FleXPath tries to integrated these two paradigms, by considering the structure of the queries as a “template”, therefore this search engine tries to find the best matching template to structure and keyword search.

FleXPath uses two separate index structures for storing and retrieving the content and structural conditions of a query, namely the structure and content index. For resolving the final structure of the query, FleXPath is eager on random-access scans on the above mentioned disk-resident index structures. Query answering in FleXPath adheres to top-k-style, furthermore, FleXPath also supports query relaxation techniques.

2.1.3 Pathfinder

Pathfinder [24, 32] is relational query processor for XQuery, and does not support XQuery queries natively, but it adds one more layer of abstraction. Pathfinder works over MonetDB, the latter serving as database back-end. XML documents in Pathfinder are shredded and stored in the database. Pathfinder compiles XQuery expressions into “equivalent” SQL statement,
or variants of table algebras \cite{24} that can be optimized for execution in the relational back-end. The translated SQL statements are then processed by the relational back-end. Finally, the results are returned as a table. This form of rectangularization of the XQuery expressions obviously incurs overheads in the query processing.

The idea behind the Pathfinder is to leverage from the mature database technology, the latter has been proven to be a very well engineered and reliable technology in managing data, in the last 30 years.

In our approach, we have no such intermediate layers, but all it is required is the translation of XQuery to the internal native DAG (directed acyclic graph) representation, which can be efficiently processed by the TopX.

\subsection{2.1.4 PF/Tijah}

PF/Tijah \cite{26} is text search system that works on top of the existing Pathfinder XML/XQuery search engine. PF/Tijah is a compiled module on top of Pathfinder, with the goal of supporting text search capability. PF/Tijah is backed-up by MonetDb \cite{31} as database back-end. It can process queries only in the main memory, this way PF/Tijah is limited to collections that are not larger than several gigabytes \cite{26}. In contrast, the TopX search engine is capable of handling very large repositories, due to its genuine way of reading the compressed indexes in chunks of blocks incrementally.

PF/Tijah supports stemming and ranking of the returned results according to some relevance criteria. PF/Tijah supports searching collections without the notion of document, that is, this search engine can search in ad-hoc fashion (any arbitrary part of the document) without having to define the document granularity upfront.

Pathfinder was not originally designed to process XQuery with text search extensions, for this reason PF/Tijah introduces another index (thus increasing maintenance costs). A disadvantage with this approach is that it inherently requires a translation of index entities from one index to another and vice-versa. In one way, a sequence of nodes created from XQuery expression, that are to be used in the later stage of processing textual query part, (identifiers in pre-order) have to be passed to the PF/Tijah index for further processing. After the PF/Tijah finishes processing the query, it has to reversely translate back the identifiers to the Pathfinder, for finally returning the ranked results, thus an extra overhead.
2.1.5 GalaTex

GalaTex \cite{15} is one of the first implementations of XQuery Full-Text according to W3C \cite{11} standard specifications. Curtmola et al. \cite{15} state that their implementation is more about XQuery Full-Text language conformance rather than the efficiency, that is, their implementation is built having on mind the completeness of the language itself. GalaTex was build to serve as a test platform for new research ideas, but mainly as reference implementation. GalaTex supports the ranked retrieval paradigm, and it supports scoring mechanisms too. GalaTex furthermore supports probabilistic scoring models for relational algebra, in order to satisfy XQuery’s Full-Text scoring requirements. It’s core engine supports XQuery. Full-Text search predicates in GalaTex are handled by translating them to a function calls that the core engine recognizes.

In contrast to GalaTex, we focus on efficiency of the query processing rather than language completeness (although we support the main building blocks of XQuery Full-Text — FLWR expressions). We strive to achieve optimal XQuery Full-Text evaluation methods, with high precision and recall. Our implementation is about a new novel approach in XQuery evaluation, aiming “divide-and-conquer” strategy when processing complex XQuery queries, and processing parts independently, whereas aggregating scores as a whole. This form of query evaluation increases the flexibility, response times as well as robustness of the system.

2.1.6 BaseX

BaseX \cite{21} is another search engine that supports XQuery Full Text. BaseX supports all the latest XQuery FT language specifications as introduced by \cite{14} with high conformance 99.9 \% \cite{21}. BaseX relies on database back-end for storing and managing its indexes, and it relies in a number of indexes structures, namely index structures for names, paths, values, and full text. BaseX support three major query evaluation mechanism, sequential scans, index-based processing and the hybrid approach.

One major drawback with BaseX is that it does not support scoring mechanisms, therefore it does not support ranked retrieval (i.e., no support for top-\( k \)), that is, results in BaseX are returned adhering only to a strict database-style query evaluation. As a consequence, BaseX does not support early termination nor pruning, thus yielding extensive result set, an disadvantage well known in database-style of query evaluation.
Such disadvantages are not present with our approach, as we process queries in both conjunctive and non-conjunctive modes, furthermore we support advanced scoring mechanisms, and in addition to that, we extend the whole XQuery Full Text processing approach to the operator based approach, that is, XQuery FT processing in parallel.

### 2.1.7 Threshold Algorithms

Fagin’s [18] family of threshold algorithms (TA) are widely used as choice for efficient top-$k$ computations, originally designed for multimedia databases. These algorithms work typically over precomputed inverted indexes with items stored in descending order of their scores. TA algorithms rely on making smart decision on index scans (scheduling decisions) and terminate the algorithm early by pruning the non-promising candidates, without having to scan the inverted indexes exhaustively. Such algorithms rely on clever booking of score intervals for their accumulated items, thus terminating early, that is, as soon as the top-$k$ items were found. These algorithms are proven to be instance optimal (see [18]), therefore can compute the exact final top-$k$ results, unlike some heuristics used in web search engine counterparts.

Threshold algorithms are attractive for ranked-retrieval applications, but their application to XML centric ranked-retrieval applications is all but straightforward. In XML data retrieval, it has to be taken into account the fact that XML repositories are rich with structural annotations and textual content. In order to efficiently answer queries over such repositories, one has to build smart indexes, capturing all relevant information, a process yielding more than one index structure, typically structure index and content index. The latter complicates further the ranked retrieval in general, not only in the domain of TA algorithms. Query answering over XML centric data repositories requires complex joins, in order to evaluate path axis (i.e., element containment), queries over such domain typically form twig patterns, therefore requiring complex joins over more than one index structure and complex path evaluation.

The main goal of TA algorithms is to find the overall best items without having to scan exhaustively the inverted lists. The same principle holds true for database objects, too [19]. TA algorithms scan relevant index lists in an interleaved fashion, with the goal of computing global scores for the items encountered. Such scores are computed monotonically from the index lists where such items were seen, i.e., by means of monotonic aggregation function, such as weighted sum. TA algorithms maintain two score bounds,
the *bestscore* that an item can possibly gain, and the *worstscore* which is the true score of the item. Worstscore bound is especially useful for terminating the algorithm, that is, when the worstscore of the $k$-th ranked item cannot be exceeded by the bestscore of any candidate item, then it is safe to stop the algorithm and present the $k$ items accumulated.

![Diagram showing sorted vs. random index list accesses](image)

**Figure 2.1:** sorted vs. random index list accesses

State-of-the-art TA algorithms are divided in three flavors: (1) the original TA algorithm [17] scans eagerly all the relevant index lists for an encountered item in a *random-scan-fashion* (see illustration in Fig. 2.1 showing the difference between sorted and random accesses), therefore the full score of each item is known immediately at the point when that item is encountered. However, such aggressive access of hard disks in random fashion is not a particularly cheap option, that is, because of expensive costs associated with random accesses over hard disks (i.e., the hard disk hand has to travel from one sector to another rapidly), furthermore there can be cases when random access is not feasible. (2) The NRA (*No-Random-Access*) algorithm relies only on sequential scans in order to compute the global best items, or so called *stream-combine* in [19]. This algorithm is attractive in scenarios when no random accesses are allowed, or not desired due to performance issues. The NRA algorithm maintains two priority queues, namely the candidate queue and the top-$k$ queue. It stops when the bestscores of all candidate items are lower the the worstscore of the $k$-th ranked item in the top-$k$ queue. (3) Combined Algorithm (CA) is a hybrid approach. It is similar to the NRA, with the difference that it periodically issues random accesses for the most promising candidates in order to resolve their final scores, i.e., for the candidates that have a small gap between bestscore and worstscore bounds. Random accesses are carefully scheduled based on some simple cost-models.
The latter denotes a environment-specific cost ratio, between random accesses $c_r$ and sequential accesses $c_s$. Basically after a number of sequential accesses, this algorithm preforms one random access, that is, whenever the cost of sequential accesses has reached the cost of the random access, but this is system specific parameter allowing for many tuning options.

The TA algorithm is the most effective algorithm in terms of index scans, that is, this algorithms performs a very low number of overall index scans, and typically terminates after a small number of such scans, but relies only on expensive random scans. NRA performs only sequential scans, thus no expensive random scans, but scans more index items than the TA. However NRA typically achieves better run-times than TA. Finally, CA algorithm is the most versatile algorithm, this algorithm is implemented by TopX \cite{36, 37, 38} as baseline, but extended in various ways. CA tries to minimize the overall access costs $c_R/c_S$, therefore allows for many constellations in terms of scheduling expensive random accesses and the not-expensive sequential counterpart.
Chapter 3

Existing Infrastructure

3.1 TopX 2.0

TopX is a search system that works over semi-structured data and plain text repositories. TopX evaluates queries in both IR-style - non-conjunctive, and in DB-style conjunctive mode (so called strict mode). IR style query evaluation leads to memory problems and queue managements issues. In IR style of query evaluation many potential candidates are returned, as every matching element is considered a potential candidate. This issue is endemic in IR systems, and requires efficient bookkeeping strategies, in order to optimize the immense memory requirements.

TopX is a mature top-k search engine for ranked retrieval of XML documents. With the ranked retrieval paradigm, it is implied that the results are returned according to some relevance score of measurement assigned to the each of the satisfied query keyword and/or structural annotation. Intuitively, the elements that score higher are placed higher in the result list. To this extend, TopX supports two main granularities in XML retrieval, document retrieval as well as the element retrieval.

TopX’s efficiency has been well proven in the INEX benchmark series, where TopX ranked among the very top search engines. TopX supports NEXI style queries as well as XPath with full-text search primitives. TopX embraces a novel block-organized index structure, coined IO-top-k [37]. The block-organized index structures allow TopX for better and more efficient IO throughput, better scheduling strategies, as well as improved scoring models.

In contrast to the RDBMS based search engines, TopX has advantage in
terms of merging index lists, due to its block-structured index lists, where TopX merges document blocks based on IDs. This is a significant improvement over hash joins, yielding more performant run-times.

However, TopX 2.0 does not support XQuery language, and it can process only one XPath query at the time. In this thesis work we refine the TopX architecture similarly to Eddies-like\[5\] tuple routing, allowing for multiple nested operators to run in parallel. Our solution generalized well to multiple XPath query evaluation, with one global top-$k$ list.

In this thesis work, we extend the existing TopX engine with XQuery Full-Text processor.

TopX will serve as implementation platform for the intended subset of XQuery Full-Text infrastructure. TopX makes an ideal platform due to its maturity, and support for a number of IR properties, such as advanced scoring models \[37\], different modes of query evaluation, enumerated as follows:

- queries with combined *structural and content conditions*
- automatic query expansions, and
- query relaxation, if too few results are found in the strict query evaluation mode

As already mentioned, TopX evaluates queries in non-conjunctive “*andish*” mode as well as in *exact-match* mode. TopX can be considered a bridge between DB-style and IR-style query evaluation. *Query relaxation*, is a helpful mechanism in relaxing query conditions, when not all the tag-terms and structural constraints can be satisfied. From an IR perspective, this component is of benefit to our infrastructure.

With such a wealth of functionalities, TopX makes an ideal platform for implementation of the proposed work. FLWR expressions (*for-let-where-return* expressions in XQuery language \[14\]) with full-text primitives such as the “*ftcontains*”\[14\] predicate are the foundations of the query language that we append to the existing set of languages in the TopX engine.

In the Fig.3.1 components of TopX system have been depicted. The system’s main components are:

- The Indexer/Crawler:
  As the name indicates, the indexer perform the job of indexing, i.e.,
Figure 3.1: TopX System Architecture

Ontology/Large Thesaurus
WordNet, OpenCyc, etc.

Non-conjunctive Top-k XPath Query Processing

Probabilistic Index Access Scheduling

Probabilistic Candidate Pruning

Dynamic Query Expansion

Index Metadata
- Selectivities
- Histograms
- Correlations

Inverted Index
Unified Text & XML Schema

Expensive Predicates
- Path Conditions
- Phrases & Proximity
- Other Full-Text Op’s

Sequential Access

Top-k Queue

Scan Threads

Candidate Queue

Frontends
- Web Interface
- Web Service
- API

TopX 2.0 Query Processor

Indexer/Crawler
parsing semistructured data and building the index, as well as updating the index.

- **Query Processor:**
  It parses the query, decomposes it into a number of XPath expressions, and invokes a top-$k$ operator per each of the XPath expressions. It is also responsible for maintaining a queue of top-$k$ items, as well as a candidate queue.

- **Index Access Scheduler:**
  It basically deals with the strategies for accessing the index lists. It makes use of some simple heuristics about cost models, which in turn are shown to be fruitful for reducing index access costs. Basically this component helps the query processor to scan those index lists which in turn result with increase of the threshold, thus yielding less candidates in the queue and faster termination. This component is also the main actor in spawning the expensive random accesses (RA), all in the name of early termination. In this thesis work we do not employ this existing component in the TopX, as we rely only on in-expensive sequential accesses.

- **Probabilistic Candidate Pruning:**
  This component is used for score prediction of candidates by making use of histogram convolutions and correlations estimates. If it is estimated early that a candidate cannot qualify for the top-$k$, then such candidates can be pruned. Intuitively, this component plays an important role in optimizing memory requirements. However, this component can not be used as it is in the operator-based architecture, therefore remains to be appended to the system in the future work.

- **Dynamic Query Expansion:**
  Users do not always formulate their queries with the best keywords. This component reformulates the query by mapping the keywords to synonyms, hyponyms, etc. available in a thesaurus. There can be cases when the user entered keywords are not contained in the index lists, in such cases, this component’s invocation can be helpful towards increasing the quality of search results. In general, this component helps to increase recall (i.e. completeness of search results), whereas the factor of precision (i.e. soundness) may drop. This component as well is not implemented in our operator based architecture, but is mentioned as an existing feature of the TopX, a feature which remains to be implemented in the future work.
Precision and recall are the main *metrics* that help measure the effectiveness of an IR system. IR search engines therefore are ranked according to such measurements. Precision and recall are also used at INEX benchmark series [29] for ranking search systems according to their effectiveness.

For a given input query, we measure precision (P) as the fraction of retrieved items relevant to the input query:

\[ P = \frac{|\{\text{relevant items}\} \cap \{\text{retrieved items}\}|}{|\{\text{retrieved items}\}|} \]

When measuring precision in such a way, we are measuring it considering the set of all retrieved documents, however one could also measure the precision for top-k results, that is precision at \( k \).

Whereas for a given query, the recall (R) is the fraction of relevant items to the query, that are successfully returned:

\[ R = \frac{|\{\text{relevant items}\} \cap \{\text{retrieved items}\}|}{|\{\text{relevant items}\}|} \]

Now, for the recall, consider the repository of all XML documents, we measure the recall as the number of elements returned divided by total number of elements that should have been returned. In practice there is always a trade-off between such two measurements, although not theoretically. If we were to focus more on precision, that is, if we increase the precision (so we are interested in a smaller specific set of returned elements), then the recall drops, and the other way around.
Chapter 4

Data and Query Model

4.1 Data Model

For data model, we follow the W3C XML standards 1.0 and 1.1. We model each XML document as a tree. Attributes in XML documents are simply considered a children of the node element where they occur. More details about how we model XML data are shown in the following section.

4.1.1 Extensible Markup Language 1. 0

Ever since XML emerged in 1998, the interest in semistructured data has grown rapidly. XML data can be rich in text, contain highly-structural data (e.g., floating numbers, number ranges), and semistructured data (e.g., text mixed with markup). Usually markup (tags) describe the content, therefore XML is known as meta-markup language. Text in XML is of type CDATA, and this simply implies character data i.e., simple strings.

XML data is represented as a labeled and ordered tree [20]. Edges represent relationships among the element nodes in the tree, and the tags represent the elements nodes. Values in the tree represent content of the node, which in turn can be CDATA. In our settings, we model semistructured data as DAGs (Directed Acyclic Graphs), although XML allows to have links from elements in form of ID/IDREF, XLink, XPointer, in our implementation this information is discarded.

In Fig. 4.1 an XML document has been depicted, in the left side – (a), and its corresponding tree model on the right side of Fig. – (b). As shown
in the figure, elements are connected with edges, where edges denote their relationship (descendant/parent) among elements. In addition, elements can also contain text, like the elements depicted in the Fig. 4.1 in the leaf level, where their textual content is shown beneath.

An important labeling scheme in our XML tree representation is the pre/post ordering. In Fig. 4.1 this ordering of elements is denoted in-between “[” and “]” brackets. For instance, element vehicle has its pre/post ordering corresponding to [0,8]. This information is invaluable for path traversals when evaluating XML queries. Pre/post ordering are useful for determining relationships among elements (helping evaluate XPath axis) such as:

- parent
- ancestor
- ancestor-or-self
- descendant
- descendant-or-self
- following
- following-sibling
- preceding
- preceding-sibling
- child
- self
- attribute

Figure 4.1: XML Data Model
In the Fig. 4.1 part b), according to the pre/post ordering, we can easily determine that the element tagged: car is descendant of the element vehicle, by making use of a very simple pre/post ordering comparison of these two elements (but not limited to only these two, the same technique is applicable for any two elements):

\[ \text{car.pre} > \text{vehicle.pre} \land \text{car.post} < \text{vehicle.post} \]

Pre/post ordering are the main building blocks in TopX for XPath axis evaluation, and in this thesis work the same concept is inherited for XQuery FT, for the reason that XQuery will work over the same index structures that already store such a labeling for all the indexed elements.

Furthermore, for each node in an XML tree, we can also associate additional information, that is, its full-content \[35\].

**Definition 4.1.1 (Element’s Full-Content):** Full content of an XML element is concatenation of its all descending elements’ text nodes in document order.

The definition above is useful for defining the partial score that an element e obtains when that element is found in the ftcontains predicate, or in the about operator in nexi style querying. For an element e to match the tag-term condition, the element e must match the tag, and the term must be contained in the element’s e subtree, or according to the definition, in element’s e full-content.

### 4.2 Query Model

Due to the nested structure of XML data, queries in XML should be structure and content aware. We categorize XML queries into two categories:

- **Database-style** queries (conjunctive)
- **IR-style** queries (non-conjunctive or so called “andish”)

Database-style queries follow the *exact match semantics*, similar to the SQL queries in databases. Queries belonging to this type, produce results only if all conditions of the query have been matched, which is a rather restrictive way of query evaluation, therefore this is rightfully known as *strict* query evaluation. Queries evaluated in this mode are not particularly useful when they are used to query large XML corpora. Such queries will result with too many results (or no results at all), and for the first case, it would be a very
demanding task to have users examine the result set exhaustively in order to find their desired items.

In the second case, and more important case for this thesis work, the queries do not comply with exact match semantics, i.e., query results do not necessarily fulfill all conditions present in the query (i.e., structure conditions or/and content conditions). Query results in IR mode are ranked according to their relevance to the query, and only top items with highest score are returned.

XPath \[12\] and XQuery \[13\] are today’s main stream querying languages for XML data sources recommended by the W3C Consortium \[11\].

4.2.1 XPath 2-0 Full-Text

The XPath FT query language is supported in the current TopX system. XPath is basic but a flexible query language allowing to select nodes, such that the path from the root to the desired node must be specified in the query’s pattern.

XPath is composed of a number of interleaving axes and tags. Most common axes are the descendant “//” and the child “/” axis. A simple query “vehicles//brand” will select all brand elements denoted with the tag brand that are descendants of a vehicle element (vehicle element is located higher in the data tree, and we call it ancestor element). If such a query was applied to the example in Fig. 4.1, there would be yet two nodes labeled brand satisfying the query condition. The query “vehicle/car” would select all car elements that are children of article element. In Fig. 4.1 there are only two such elements, and this can be easily seen by following the immediate edges from the vehicle element.

The W3C Full-Text extension was added to XPath in order to adapt to the IR-style querying, hence making it applicable for XML sources that contains large text chunks. XPath Full-Text is a more flexible query language than the classical db-style XPath, in that it allows to compose more complex queries with full text search predicates. In this thesis work, queries with full-text predicates yield more interesting cases. Consider the following query:

```
/\texttt{/article[./sec ftcontains "Max-Planck-Institute"]}
/\texttt{//title[./par ftcontains "Nobel Price"]}
```

24
This example query forms the twig pattern shown in Fig. 4.2.1. This XPath query returns all paragraphs \texttt{par} containing the keywords “Nobel Price” which belong to the \texttt{title} elements, and the latter is descendant of the \texttt{article} elements that contain a child element, namely the \texttt{sec} element with the keyword constraint “Max-Planck-Institute”. The path \( /\texttt{article}//\texttt{title}/\texttt{par} \) is the main path in the above shown query, whereas the content between “[” and “]” is the full text predicate. The query above has only one target element (output node), namely the \texttt{par} elements. The target element has been marked with bold a circle in the twig pattern shown in the Fig. 4.2.1.

Twig patterns are the main building blocks of the XPath Full Text query. As shown in the query example above, the XPath example returns only one target element, but this is a limitation in the XPath language, that is, XPath expressions can return only one target element. Important to this thesis work is that twig patterns are well supported in TopX, not only in the XPath language, but in the NEXI querying language too. In this thesis work, we extend this functionality to many twig patterns, therefore we support more than one target element. Full-text extensions are especially useful for querying text-dense repositories.

4.2.2 XQuery 1.0 with Full-Text Extensions

XQuery [13] is a more complex and more expressive language than XPath. Valid XPath expressions are considered valid XQuery expressions too, for the reason that XQuery borrowed path expressions from XPath. The main building blocks of XQuery are its \texttt{For-Let-Where-Return} (FLWR) clauses.
- **For** clause - allows user to define variable and bind it with nodes selected by XPath expression

- **Let** clause - is similar with the **For** clause, but adds another extra feature by allowing user’s defined variable to bind with the nodes selected by the already declared variables in the **For** clause

- **Where** clause - is used for selection, namely in this clause, users can specify the **ftcontains** predicates as well as join predicates, and boolean comparisons.

- **Return** clause - can be used to format the returned items in XML format, instead of simple tuples.

XQuery is a very expressive language, it allows for multiple nested XPath expressions, as well as nested **for** and **let** binders. Nested XPath expressions are the core of our system implementation in this thesis work, as we are interested in the evaluation of such expressions in parallel and aggregate results as a whole, hence “holistic” XQuery processing. Furthermore, XQuery allows nested FLWR expressions. As mentioned, XQuery is a more expressive language than XPath, but nested and compositional syntax of XQuery comes with the cost of increased complexity, making XQuery queries hard to optimize.

```xml
for $a in index(wiki.xml)//article/
let $s := $a/sec
let $par := $a//title/par
where $s ftcontains "Max-Planck-Institute" and $par ftcontains "Nobel Price"
return $par;
```

The query above, is the same XPath query rewritten in XQuery syntax, thus it yields the same twig pattern as the former, shown in Fig. 4.2.1. XQuery queries in contrast to XPath queries (which can have only one target element) can create more than one target elements [20] (although the example above creates just one), therefore query evaluation in XQuery might require evaluation of multiple twig patterns. In addition to that, queries in XQuery allow for inclusion of more than one output node, whereas XPath queries allow only for one.

We have already mentioned that XQuery allows for multiple bindings to XPath expressions, this way creating multiple twig patterns. In the example above there are three user-declared variable binders, namely $a bound to
path //article/, $s is to the path of variable $a and the trailing path /sec, resulting with the path //article/sec/, and last but not least, variable $par is to the path //article/title/par. Notice, that in XQuery we can easily bind variables to other already declared variable, this way creating complex path expressions in XQuery is an easy task, but the same cannot be said for XPath. One important difference between XPath and XQuery is that XQuery returns elements sequence (i.e., it lets you “loop” over the sequence), whereas XPath may return element set.

4.2.3 Narrowed Extended XPath I (NEXI)

NEXI is a new query language introduced by the INEX [29] work group. NEXI is a simplification of XPath, it looks similar, but IR amendments were introduced. INEX work group was aiming to reduce verbosity of the XPath, by allowing only the descendant “//” and “.” self operators and removing other XPath features. However, NEXI introduced the new – “about()” IR-style operator, similar to the “ftcontains” in XPath and XQuery FT. The main and the most important divergence of NEXI from XPath is that of semantics. In XPath, the semantics are well defined, whereas in NEXI that is left up to implementors, namely by deducing the semantics from the query [41].

TopX supports NEXI queries, which can also be used in our approach, that is, our operator-based architecture allows for evaluation of NEXI queries in parallel.

NEXI queries can be classified into two classes:

1. (CO) – Content only queries
2. (CAS) – Content and Structure queries

**Content Only (CO) queries** – are especially useful when users are not familiar with the XML structure of the documents (i.e., tags), and they can simply issue only keywords and phrases for search.

```
  // *[about(., John Nash)]
```

This query assumes no structure, only keywords (john nash) were used. For such queries, the wildcard operator (*) is used, namely instructing the search engine to ignore the structure.
Content and Structure (CAS) queries – may contain structural information, that is, users can refine their queries, and ask for specific parts of documents (e.g., specific article or section). And if such documents contain the structure given in the query, then these documents yield higher scores, and rank higher in the top-\(k\) results list.

//article/[//sec[about(.),Game Theory]]]

In this example, the user is interested about a section in an article, containing the keywords “Game Theory”.

4.2.4 TopX Query Representation and Query Parsing

As previously stated, we decompose XQuery queries into multiple nested XPath expressions, basically we separate twig patterns in XQuery into a number of single twig patterns, this way forming a framework for twig pattern evaluation in parallel.

Algorithm 1 shows steps that the system takes in order to transform XQuery expressions to XPath expressions. We create one XPath expression per one XQuery variable binder, given that the variable is binded with the full-text \texttt{ftcontains} clause in the \texttt{where} clause.

**Algorithm 1:** XQuery decomposition into XPath expressions

<p>| Input: A XQuery Full-Text |</p>
<table>
<thead>
<tr>
<th>Output: XPath Queries ({q_1, \ldots, q_n})</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 foreach $var \in XQuery do</td>
</tr>
<tr>
<td>2 $temp \leftarrow \text{path}($var);</td>
</tr>
<tr>
<td>3 if $var \in \text{where} then \quad \text{// where clause in FLWR Expression}</td>
</tr>
<tr>
<td>4 $\text{keywords} \leftarrow \text{getKeywords($var)};</td>
</tr>
<tr>
<td>5 $\text{temp} \leftarrow \text{temp} \cup [\text{. ftcontains &quot;$\text{keywords}&quot; }];</td>
</tr>
<tr>
<td>6 return $\text{temp};</td>
</tr>
</tbody>
</table>

The main idea is to decompose the XQuery into number of XPath expressions, such that we can evaluate XPath expressions in parallel, in our operator based architecture. After we have obtained XPath expressions, we can then pass XPath queries to the query processor, so that it can create internal query representation (for internal query representation see [37]), and then proceeds with the query evaluation.

To better illustrate Alg. 1, we will re-examine the query example from Sec. 5.1.
for $book in index(wiki.xml)//article/section/
let $l := $book/p/
where $book ftcontains "hercule poirot" and $l ftcontains
"private detective"
return $book

In the above query, there are two variable binders, namely $book and $l corresponding to for and let binder. This should give a clear indication that our XQuery might contain two XPath expressions. However, we can’t build XPath expressions blindly by relying only on the number of variables present in the query. We inspect further the where clause, to see if the variables are present (see Line 3 in Alg. 1) there too. That is, we are interested in valid XPath expressions with target elements, and contain the ftcontains predicate. We deviate slightly from the XQuery syntax, by allowing variables to bind to the index (in this case, the index corresponds to the Wikipedia XML corpus), in the standard W3C XQuery, for variables are typically bound to XML documents. The new index construct, allows us to increase the search space from document granularity to an entire XML corpus.

In our example, both variables appear in the where clause. Variable $book appears in the where clause with content “hercule poirot”, and according to our algorithm, we obtain the XPath expression:

//article/section[. ftcontains 'hercule poirot']

Basically, we create such XPath expressions by combining paths assigned to the user-declared variables and the ftcontains clause corresponding to those variables. The above generated XPath query was created in this way, namely by appending to the path assigned to the variable $book, and the full-text ftcontains clause. Similarly, variable $l, first obtains the path assigned to variable $book as prefix, then appends its path with (/p/), and finally the ftcontains clause, resulting with:

//article/section/p[. ftcontains 'private detective']

4.2.5 XQuery Parser

Since TopX could parse only NEXI and XPath style expressions, we extended that functionality with XQuery parser, so now it supports XQuery expression as well. Not all the features present in XQuery Full-Text [13] are supported by our parser, as the XQuery FT is a very expressive language, and this thesis
work is rather about efficient query processing than language completeness. Although we support the most used and crucial language constructs such as \textit{For-Let-Where-Return} (FLWR) expressions.

![Figure 4.3: XQuery FLWR Model Tree](image)

For parsing XQuery we use \textit{java cup} [28], which is LALR (Look-ahead Left to Right), bottom-to-top parser. In what follows, we show a small part of the grammar that our parser uses to evaluate XQuery FT queries (in BNF:\footnote{BNF: Backus-Naur Form} format):

\begin{verbatim}
start with XQuery;

XQuery ::= FLOWR:flwr {: RESULT = new XQueryExpr(flwr); :} |
Path:pa {: RESULT = new XQueryExpr(pa); :};

FLOWR ::= ForExpr:fe {: RESULT = new FLOWRExpr(fe); :} |
LetExpr:le {: RESULT = new FLOWRExpr(le); :} |
WhereExpr:we {: RESULT = new FLOWRExpr(we); :} |
ReturnExpr:re {: RESULT = new FLOWRExpr(re); :};

ForExpr ::= FOR VAR:v In:index Path:pt FLOWR:flwr |
{: RESULT = new ForExpr(v.toString(), index, pt, flwr); :};
\end{verbatim}

\footnote{BNF: Backus-Naur Form}
The production rules shown above represent the main production rules for our recursive query parser. In Fig. 4.2.5 we have also shown the visual representation of these production rules. Numbers in square brackets ‘[’ and ‘]’ show how many times that particular expression may occur in the query, in order for that query to be considered a valid query. If such numbers are not present, it is understood that expression by default may occur only once.

The code in between curly braces ‘{’ and ‘}’ shown in the production rules above, is simply Java code that is executed whenever the hosting production rule has been matched during the parsing. In other words, first the text taken from the query input goes through the lexer (lexical analyzer), which then removes white spaces and applies a number of regular expressions, this way converting the input to a number of tokens, i.e., tokenizing the input query. Now, each token must go through the parser for syntactical evaluation, and whenever the parser matches the token or sequence of tokens against the production rule (a subset of such production rules have been shown above), it fires the Java code embedded in such a rule, otherwise an exception is thrown. We show XQuery processing steps (flow-diagram) including the above mentioned steps in Fig. 4.2.5.
Figure 4.4: XQuery flow-chart diagram
Chapter 5

Relevance Scoring Model

5.1 Probabilistic Scoring Model – Okapi BM25

In IR, users are typically interested in items that have the highest score with
respect to the query, therefore it is the score that distinguished the items
that will make it into the top-k list, this usually normalized between [0, 1].

TopX employs the XML specific the Okapi BM25 [37] scoring model, which
has been proven effective for more than 4 years of INEX benchmarking par-
ticipation. Okapi BM25 is a predominating ranking function in the world
of search engines. In the context of the TopX, we use this function to rank
documents according to their relevance with respect to a given query.

Okapi BM25 is composed of approximate 2-Poisson model for term relevance,
and RSJ – the Robertson Sparck-Jones [33] Model for term specificity. Okapi
BM25 has been shown empirically as a very successful model in INEX bench-
mark series as well as in TREC [39] (text retrieval benchmark).

Okapi BM25 works by invoking probability ranking principle. Using this prin-
ciple, documents are ranked according to the probability of relevance of the
query. However, because Okapi BM25 was designed for scoring text repos-
itories, TopX employs the XML variant of the model. This model captures
the influence of both structural and content conditions of the query. XML
variant of Okapi blends well in terms of the XQuery Full-Text too, as the lat-
ter is a query language intended to query XML repositories, and it contains
content constraints as well as structural constraints, therefore we can inherit
the scoring features from the already supported XPath Full-text language in
TopX.
In this example, this XQuery contains more than one content constraints, namely the \texttt{ftcontains} predicates (e.g., \texttt{ftcontains "hercule poirot"}), as well as the structural constraints (e.g., \texttt{//article/section/}). As can be seen from the example query, the relevance of the structural constraints as well as the content constraints are important to the user, and they should influence the weight of the resulting items returned by the system.

5.2 Retrieval Modes in Semistructured data

XQuery Full-Text queries contain both structural and content conditions. It is important to distinguish two ways of evaluation of content conditions in such queries. The first way of evaluation is known as non-conjunctive or so called “andish” mode, and the latter is known as conjunctive mode respectively.

For a given query \( q \) with content conditions: \texttt{// ["A_1, \cdots, A_n"]}, XML node could match the query if it contains any of the terms \["A_1, \cdots, A_n"]\ in its \textit{full-content} (see Def. \[4.1.1\] for the definition of full-content). In the second case, all term conditions need to be fulfilled, i.e., an element must contain all the terms in its \textit{full-content}. This should not be confused with phrase match, as full-content doesn’t necessary imply phrase match and vice-versa. The queries in first mode are harder to evaluate, as this form of query evaluation yields with many potential resulting candidates, which in turn require significant portion of the memory allocation. Therefore, the non-conjunctive mode requires considerable memory space for bookkeeping the candidates in the candidate queue.

5.2.1 Content Scores

For calculating content scores, TopX uses statistical measures, that is, each node’s content in XML tree is viewed as a \textit{bag-of-words}. In this application, an important role plays the \textit{full-content} of a node \( n \), which includes content
of the node $n$ itself, in addition also the content of all other descendant nodes of $n$, where to $n$ is referred as context node.

$$\{e \mid \forall e \in N \lor \forall e \in D, \forall D \subset N\}$$

(5.1)

e : term
N : context node
D : descendant node(s).

We have shown a small graphical illustration of the concept of full-content in Fig. 5.1.

Text in a node, as mentioned earlier, is considered as a bag-of-words, hence to this extend, it is only important the number of times the term appears, whereas the order of terms is irrelevant, so are no punctuational and structural annotations.

For content scoring, TopX employs three main building blocks:

1. $ftf(t, n)$ – denotes the full-content frequency of a term $t$ contained in XML node $n$, which resembles the number of occurrences of term $t$ in the full-content score of $n$.

2. $N_A$ – denotes the tag frequency of a tag $A$, which is the number of all nodes in the corpus with tag $A$.

3. $ef_A(t)$ – denotes the element frequency of a term $t$ contained under a tag $A$, i.e., number of XML nodes tagged $A$ that contain the term $t$ in their full-content with respect to the entire corpus.

In Fig. 5.1, $ftf(\text{article, database})$ is 5, which resembles occurrence statistics of the term - “database” (term is shown in bold in the figure) in the scope of element - “article” (we count the term also if it occurs in the descendant nodes, see Eq. (5.1)).

In Fig. 5.1, XML nodes are shown in rounded gray-shaded rectangles, whereas their content is shown beneath the node. The tag label is shown inside the rectangle, as well as pre-post element ordering.

For a given tag-term pair in (XPath) in form of $//T[.\text{contains "t"}]$ ($T$ can be a tag match, or a wildcard character * that matches any tag), the score is calculated by invoking the following BM25 formula:
A Database is an integrated collection of logically related records.

These databases store data and information extracted from selected operational and external databases.

Figure 5.1: Data Model

score(e, //T[.ftcontains “t”]) = \left(\frac{k_1 + 1}{k + 1}\right) \frac{ftf(t, e)}{ftf(t, n)} \cdot \log \left(\frac{N_A - e_A(t) + 0.5}{e_A(t) + 0.5}\right)

having \( K = k_1 \left(1 - b\right) + b \frac{\sum_{e'} \text{avg}(\sum_{e''} \text{ftf}(t, e''))}{\text{ftf}(t, e)} \) with tag A.

\( k_1 \) and \( b \) are Okapi BM25 tuning parameters, and in TopX they are used with values: \( k_1 = 1.25 \) and \( b = 0.75 \)

For queries with multiple content conditions, the score is evaluated as sum over content conditions of satisfying elements:

\[ \text{score}(e, //T[.ftcontains “t_1, \cdots, t_n”]) = \sum_{i=1}^{n} \text{score}(e, //T[.ftcontains “t_i”]) \]
5.2.2 Structural Scores

For the reason that users normally are not familiarized with the structure of the data, they usually neglect to annotate their queries with structural conditions, and put more efforts entering the content conditions. The system should not penalize users by requiring them to know a-priori the structure of the corpus, as this would require the users to learn a large set of tags. In non-conjunctive (IR) mode of query evaluation, if there is structural information in the query, then each matched conditions is awarded with a predefined score mass, but the absence of such conditions would not result with empty result set, due to the query relaxation techniques.

To this end, in our TopX settings we employ a system predefined static score mass $c = 1$ [35]. Since the constant $c$ denotes the relative weight of the structural conditions, then one could lower it if more emphasizes is desired for content conditions and vice-versa.

Consider the query:

```
//article[/sec[about(../title, Game Theory)]]
//author[about(../name, John Nash)]
```

The matching target elements title and name are required to be descendant of both sec and article, and author and article tags respectively. Query
evaluation in such sense yields a DAG (directed acyclic graph) representation, where each tag-term condition is placed at the bottom of the query tree, i.e., leaf level. For this example query, its DAG representation has been shown in Fig. 5.2.

The elements which can satisfy the structural conditions are awarded with the score mass $c$, for each of the satisfied structural condition, furthermore this score is later aggregated similarly with content condition using summation. This process yields the final score for the candidate item.

5.3 Index Structures

For evaluating XQuery FT queries, we rely on the existing index structures supported by TopX 2.0. TopX combines two major indexes, namely the structure index and the content index for evaluating the XPath Full-Text queries. Both indexes are implemented as binary files. Disk I/O operations are known to be very expensive, hence they bottleneck the performance of the algorithms. Actually in TopX, the disk I/O operations are the most expensive operations. For this reason, it is more efficient to have the indexes organized in blocks, and fetch the elements from index lists in block units, rather than tuple units. This offers advantages in terms of reducing the number of I/O operation and at the same time increases the disk throughput.

The index structures in TopX are composed of element blocks, consisting elements that share the same tag-term pair, and in turn element blocks are organized in another layer, the so called – document blocks. Document blocks are of fixed size, size is a system parameter, e.g., can be set to the size of a disk sector. Elements in document blocks are organized in descending order of their document IDs, allowing for efficient $m$-ary merge joins, when queries have more than one dimension. When using merge joins, items are visited at most once, whereas hash joins would require nested loops. Merge joins offer better runtimes, as merging lists using $m$-ary merge joins is very fast, items are accessed at most once in round robin fashion from each of the $m$ dimensions. Furthermore, the document blocks are sorted in descending order of their maximum element block score, a property making possible the institution of Fagin’s threshold family of algorithms (in this thesis work, the NRA [18] algorithm), which in turn enables early termination in top-$k$-style of query evaluation.

Our content index structure is depicted in Fig. 5.3, showing graphically how elements are combined in document blocks, i.e., per tag-term pair. In
5.3.1 Content Index

In TopX, content index is stored in a single binary inverted file [36]. The logical structure of such a binary file has already been shown in Fig. 5.3. Index lists in such binary file are separated by the delimiter ‘L’ as shown in the picture. TopX maintains a dictionary (auxiliary hash-based dictionary), containing information about the offset of such index lists in the binary file, thus allowing for sequential and random access to such indexes. For each tag-term pair there is one such index list.
In the Fig. 5.3 the logical index list for tag-term pair sec[‘xml’] has been shown. Elements in such index lists are organized in triplets – (pre, post, score), and in addition to that, elements also contain a header, storing the document ID, that is, the document ID that the particular element belongs to. Block sizes in TopX settings are typically between 256-512 KB, thus they are fetched in one I/O operation in disk, although block-size is a configurable system parameter, and can be changed accordingly.

TopX supports probabilistic pruning by employing histograms, but not limited, histograms are also used for employing sophisticated cost models as shown in [37], and they are helpful in terminating algorithm early, without having to find the score bounds for all items in the candidate queue (for more on this see Sec. 7.6, or [37, 38]). Inverted files include histograms (histograms are not shown in Fig. 5.3), which can be used to optimize the number of blocks we read from particular index list. Histograms are divided into number of buckets $b$, and for a particular tag-term pair, a bucket $b_i$ stores the number of documents that have their maximal score in the interval $[1 - \frac{b_i-1}{b}, 1 - \frac{b_i}{b}]$.

### 5.3.2 Structure Index

In similar fashion as the content index, TopX stores an inverted file for all matching elements for a given tag. Structure index identically as the content index stores a sequence of document blocks (per tag), which in turn contain nested element blocks with their pre/post labels, but without score entry, as for each matched structural constraint, TopX assigns a static score to such elements, thus there is no necessity for the score entry. There are also no histograms in the structure index, for the same reasons [36].

Analogously as the content index, TopX maintains an auxiliary structure (a dictionary), this way supporting sequential and random scans.
Chapter 6

XQuery Processor

6.1 Top-k threshold algorithms

Query processing over non-schematic XML data is a difficult task when only boolean search predicates are used. In such scenarios, either too many or too few results are returned. Hence, a new retrieval mechanism is called for. XPath and XQuery with full-text search primitives enable application of IR-style querying to XML databases. XQuery Full-Text search predicates have first been introduced in [2], suggesting new powerful predicates intended to perform full-text search in XQuery. IR-style querying works by scoring the elements in accordance to how closely they match the keywords in the query. Users are typically not interested in the whole exhaustive result set, but in the list of a few top relevant results.

TopX evaluates queries by scanning in parallel the pre-computed inverted lists, which are sorted in a descending order according to the document score.

TopX decomposes the query into a number of tag-term pairs and structural constraints. The system then scans the query relevant pre-computed inverted lists in parallel. By employing a monotonic score aggregation function (see Algorithm 2), it computes two score bounds for each candidate item, namely the worst score for the dimensions that the candidate was seen (current top-k), and the best possible score (or an upper bound) that a specific candidate can gain. The algorithm stops scanning the lists, when it has determined top-k objects. Application of the family of threshold algorithms, introduced by Fagin et al. in [18, 17], is an effective way of lowering the number of index accesses (less I/O) and memory requirements, this way making early
termination a very attractive instrument in ensuring efficiency in the top-k computation.

### 6.2 Naive Approach

One naive approach towards top-k computation for XQuery FT queries, would be the one where XPath expressions in the XQuery are executed in parallel and independently without having an upper entity or a broker that would synchronize such threads of XPaths. One could simply have each operator compute its top-k results, and when all the operators have halted, their individual top-k scores are combined, i.e., the scores of the same items appearing in more than one top-k lists are simply aggregated.
This algorithm would not provide us with guarantees that the collected top items are indeed the true top items, thus such algorithm would very likely yield incorrect results, for the reason that the score bounds are not computed “globally”, that is, such scores are computed independently and in isolation.

Moreover, the challenging issue with such an approach arises with the \( \text{min-}k \) threshold computation, since the operators would run in parallel and independently, one would need to guess the stopping conditions for all the operators (one common threshold for all the operators). Calculating the threshold in such a scenario is hard, as the operators are not communicating with each other, furthermore they do not exchange information about their candidates in their respective candidate queues. As an example, operator \( o_1 \) could have a candidate item \( t_1 \) with bestscore of 0.79, an the \( \text{min-}k \) threshold at this operator at that time was 0.8, so the operator \( o_1 \) would have simply dropped this item. But, in parallel, lets say an operator \( o_n \) could have had the same item \( t_1 \) with score 0.1 in its candidate queue. So if the scores of such items were combined, then this candidate could have made it at least to the candidate queue (assuming that the global \( \text{min} - k \leq 0.79 \)). The idea here is that score calculations have to be computed globally, in order to compute a sound top-\( k \) list with guarantees.

### 6.3 New Approach - Operator based XQuery processing

In its incarnated form, the system translates XQuery Full-Text expressions into several XPath expressions (\( n \) expressions). For each of the XPath expressions, the system then initiates \( n \) nested top-\( k \) operators, so that they can start scanning the inverted index lists in parallel. Nested operators report their elements together with their scores asynchronously to the top operator. The latter does not require all \( n \) operators to scan exhaustively their corresponding index lists, which would be a naive way of calculating top-\( k \) results, but rather asks them to halt at the moment when the top operator has accumulated the final top-\( k \) elements.

Nested top-\( k \) operators are light-weight top-\( k \) operators, which propagate and update the score bounds to their top-level top-\( k \) operator. Nested operators do not maintain a top-\( k \) queue, and they’re not required to maintain the min-\( k \) threshold, nor prune the items themselves (see Sec. [7.6] for item pruning at nested operators). Such operations are reserved only for the top operator.
All the nested operators report their elements in a non-blocking manner to their top operator. In following the Fagin’s NRA [18] algorithm, modified for our purpose is shown:

**Algorithm 2: Top-k computation in operator based architecture**

**Input:** XQuery Full-Text Expressions  
**Output:** top-k result list

*We consider each of the nested operators as an independent unit of query processing.*

1. decompose the XQuery into multiple XPath expressions:

   \[ XQ := \{ q_1, \ldots, q_n \} , \]

2. each operator \( o_i \) corresponding to \( q_i \) starts scanning the sorted inverted list sequentially, and in parallel,

3. each operator, maintains the current scan position \( pos_i \) for each list \( L_i \),

4. the operators maintains \( high_i \) scores at \( pos_i \), where \( high_i \) denotes the upper bound for the unknown scores of the remaining items in the index list,

5. each operator maintains a queue of the encountered items,

6. the top operator maintains two priority queues:
   - candidate queue,
   - top-k queue.

7. each queue item \( t_i \), has a flag “isNew”,

8. top operators fetches the queue items from the nested operators:

   \[ \{ t_i \mid t_i.isNew = true \} , \]

9. nested operators maintain a set of evaluated query dimensions:

   \[ \{ E(t) \mid E(t) \in q_i \} , \]

10. nested operators maintains a set of remainder query dimensions:

    \[ \{ \bar{E}(t) \mid E(t) \in q_i \} , \text{where the score of item } t \text{ is unknown}, \]

11. top operator aggregate the scores of the items if they belong to the same document, and for each item it:
   - maintains \( bestscore(t_i) \) and \( worstscore(t_i) \) bounds,
worstscore is the lower bound for an item \( t_i \) (i.e., total score accumulated for the item so far)

\[
worstscore(t) = \sum_{i \in E(q_i)} worstscore(t), \forall q_i
\]

bestscore is computed dynamically:

\[
bestscore(t) = worstscore(t) + \sum_{b \in E(q_i)} high_i, \forall q_i
\]

12. \( \sum_{b \in E(q_i)} high_i, \forall q_i \) at the leaf level operators, simply calculates the upper-bound for an item based on scan position in the index lists, therefore \( q_i \) at such level represents the index lists,

13. the lowest worstscore in the top-k list of the top operator, coined \( min-k \) serves as threshold,

14. when the highest bestscore in the candidate list is less than the \( min-k \), the top operator can stop the algorithm and return the top-k items. See Eq. 6.1

\[
\max_{t \in \text{top-k}} \{bestscore(t)\} \leq \min_{t \in \text{top-k}} \{worstscore(t)\}
\]

This way of score calculation, leaves uncertainty about the candidates not in the top-k queue. Therefore, the candidates which have their \( bestscore > min - k \) and \( worstscore < min - k \) are kept in a candidate queue (see Fig. 6.2). The candidates which have their \( worstscore \geq min - k \) are kept in a top-k queue (see the left diagram on Fig. 6.2). Elements in candidate queue are considered potential candidates, these candidates can gain more score, if they are encountered in other query dimensions/operator (i.e., such candidate could be reported by other operators). These candidates can still make it into the top-k list, thus they cannot be pruned prematurely, unless some probabilistic estimators are employed (for more on this see [37]).

Our solution generalizes to multiple operators, where each of the nested operators, propagates and updates bounds for its new and updated items to its upper level operator. This way the scores are propagated all the way to the top operator. The score aggregation at the top-level operator works over
Figure 6.2: top-\(k\) and candidate items illustration

Figure 6.3: XQuery operator-based query evaluation
bounds instead of simple scores (aggregation functions over such bounds are still monotonic), see Fig. 6.3 for a snapshot of our operator-based query evaluation.

In Fig. 6.3 there are three operators represented in gray-shaded triangles. The lower level, or leaf level operators, as we refer to them, scan their index lists in round-robin fashion per each tag-term pair and structural constraints. As they encounter items they build their queues with such items. The top operator, then works over the queues of the items previously built by the nested operators, also the scores of the items at the top operator are computed over the bounds of the scores at the nested operators. See for example the item D7, where top operator has aggregated its score to 1.2, as this item appeared in both queues. We will show in details our operator-based architecture at work on Chapter 7.

6.4 Instance Optimality

The notion of instance optimality is introduced by Fagin et al. in [18] for their family of threshold algorithms. Let’s quickly recall this notion. Let \( A \) be the class of all algorithms, and let \( D \) be the set of all legal inputs to \( A \). By legal inputs we refer to all index lists that have their items sorted in descending order of their scores.

We measure the cost for running algorithm \( A \) over \( D \) as \( \text{cost}(A, D) \). We say that an algorithm \( B \) is instance optimal over the class of algorithms \( A \), and class of legal inputs \( D \), if for every \( A \in A \) and every \( D \in D \):

\[
\text{cost}(B, D) = O(\text{cost}(A, D))
\]

Following the above equation, there are constants \( c \) and \( c' \), such that:

\[
\text{cost}(B, D) \leq c \cdot \text{cost}(A, D) + c'
\]

for every \( A \in A \) and \( D \in D \), with \( c \) denoting optimality ratio.

In the context of TopX XQuery, we define \( O(\text{cost}(A, D)) \) to be the cost for index accesses, i.e., number of document blocks that we fetch from index lists, and for that purpose we rely only on sequential accesses, and no random accesses. Therefore, for our architecture we work on top of the the NRA (no random access) algorithm, the latter presented as follows:
We first present the original NRA algorithm, and then we show that it’s instance optimality holds for our operator based architecture. We now present the NRA algorithm in more details, as we have already mention it briefly in the related work section:

1. Perform sorted access in each of the inverted lists $L_1, \cdots, L_m$, maintain depth $d$ cursor, where $d$ objects have been accesses in the sorted lists.
   - Maintain the bottom values $x_1, x_2, \cdots, x_m$ for the items already encountered under sorted access.
   - Let the function $\tau$ be a monotonic aggregation function, and $E(R)$ set of dimensions where an item $R$ has been seen.
   - Calculate worstscore $W^{(d)}_S$ and bestscore bounds $B^{(d)}_S$.
     - Worstscore $W^{(d)}_S(R) = \tau(x_1, x_2, \cdots, x_m)$, $\forall x_i \in E(R)$.
     - Bestscore $B^{(d)}_S(R) = \text{aggr}\{W^{(d)}_S(R), \tau(x_1, x_2, \cdots, x_m)\}$, $\forall x_i \notin E(R)$.
   - Let $T^{(d)}_k$ be the current list with top-$k$ items, that is, the list with $k$ items with largest $W^{(d)}_S$ values seen so far (if two items have the same $W^{(d)}_S$, ties are broken arbitrarily, wins the item with highest $B^{(d)}_S$), let $M_k$ be the $k$th largest $W_S$ value in the $T^{(d)}_k$.

2. We call an item $R$ viable if $B^{(d)}_S(R) > M^{(d)}_k$, algorithm halts when at least $k$ objects (distinct) have been seen, and there are no viable objects left, that is, for all $R \notin T^{(d)}_k, B^{(d)}_S(R) \leq M^{(d)}_k$.

**Theorem 6.4.1: NRA correctly finds top-k items if the function $\tau$ is monotone.**

We deviate slightly from the concept of NRA algorithm to document block index structures, instead of record tuple notion used by Fagin et al. [18, 17]. Fagin et al. [18] in their proof of instance optimality for NRA, assume that the algorithm fetches one tuple per index scan at each index lists in parallel, that is, NRA in its original form works on tuple granularity.

In ToXXQ we fetch blocks of documents, sorted by document IDs in ascending order (a property required for efficient $m$-ary merge joins) and within document, elements are sorted with their scores in descending order. In contrast to Fagin et al. [18] NRA algorithm, our extended NRA algorithm fetches one document block per one index scan. The difference between granularites has been shown in Fig. 6.4 for illustration purposes.
Proof of the NRA has been shown in [18], since we rely on NRA, we show that TopX^{XQ} is also instance optimal, but with a small constant overhead.

**Definition 6.4.2 (Size of document block):** The size of a document block is tunable parameter, and rarely is larger than the size of the largest element block.

**Definition 6.4.3 (Document’s block worst-score):** A document’s block worst-score is the highest element’s worstscore (W_S) encapsulated in that particular document.

A document block’s worst-score is particularly useful when determining min-k threshold test in our block-based index architecture.

**Theorem 6.4.4:** Let the aggregation function \( \tau \) be monotone. Let \( D \) be the class of all legal inputs, and let \( \mathcal{A} \) be the class of all algorithms that correctly find \( T_k^{(d)} \) items for \( \tau \). Furthermore, let \( \mathcal{A} \) be the class of all algorithms that do not make random access and make no wild guesses. We say the extended NRA for TopX^{XQ} is instance optimal over \( \mathcal{A} \) and \( D \) with small constant overhead in worst-case scenario.

By “no wild guesses” it is meant that the algorithm makes no random access for resolving the score of any encountered item, in contrast to TA algorithm [17], which resolves the final score of each item encountered sequentially, by means of random accesses.

\[
\begin{array}{c|c|c|c|c}
\text{Doc} & \text{Pre} & \text{Post} & \text{Score} \\
1 & 23 & 48 & 0.8 \\
2 & 15 & 0.7 \\
10 & 8 & 0.5 \\
5 & 24 & 0.4 \\
3 & 11 & 0.3 \\
\end{array}
\]

- **Figure 6.4:** Inverted Index Structure comparison

Proof: Let \( NRA \in \mathcal{A} \). For computing \( T_k^{(d)} \), suppose that the \( NRA \) halts at
scan position $d$ (depth). By position $d - 1$ it saw at least $k$ distinct objects. NRA must reach scan at depth $d$ at least in one of the lists in order to halt the algorithm, from here it follows that the optimality ratio of the extended NRA is $m$. Let $b$ denote the number of elements in one document block in TopX index structure. For every index access TopX$^Q$ fetches one document block, that is, $b$ entries. If NRA halts at position $d$, and by this position it has accessed index lists $m \times d$ times, iff $(d \mod b = 0$, then TopX$^Q$ has accessed exactly $m \times \frac{d}{b}$ document blocks. Whereas in worst-case scenario if $d \mod b > 0$, then TopX$^Q$ has accessed exactly $m \times \lceil \frac{d}{b} \rceil$ document blocks. Therefore it follows, that TopX$^Q$ accesses at most $m \times (b - 1)$ entries in our operator based architecture. That is, our algorithm accesses at most $m \times (b - 1)$ more index entries than the NRA, given that $m$ denotes the the overall number of inverted lists that operators have accessed. We conclude that TopX$^Q$ is instance optimal with small overhead in worst-case scenario for any algorithm $A \in \mathcal{A}$.

It is important to emphasize that the notion of instance optimality is a stronger notion compared to only worst-case or average-case optimality.

From practical point-of-view, TopX$^Q$ performs much better than the classical NRA, due to its index organization in document blocks, if the index access was to be considered the only cost. In recent years, main memory has been following a very fast growing trend, whereas disk I/O speed has improved by very slow pace. Therefore IR and database systems that operate over large disk-resident index structures focus on reducing the number of I/O scans, and many times such systems rely heavily on caching and data redundancy. It follows, that TopX$^Q$ is much more efficient in terms of I/O operations compared to the systems that perform one such operation per tuple fetch. Classical NRA makes one index access per tuple, where TopX$^Q$, accesses $b$ elements per one index access, therefore TopX$^Q$ will make at most $\frac{d}{b} + 1$ index accesses for finding top-$k$ items, given that NRA halts at scan position $d$. 

50
Chapter 7

System Implementation

7.1 TopX\textsuperscript{XQ} Operator API

Listing 7.1: Operator API (C++)

```cpp
class Operator {
    public:
        virtual void getNextBlock() = 0;
        virtual QueueItem* getNextItem() = 0;
        virtual float getHighBound() = 0;
        virtual void prune(QueueItem* item) = 0;
};
```

The virtual functions shown above, are the functions that are used by an operator in our system implementation. We can call each of these functions on any operator, and they are the main interface for interacting with an operator, as we shall see in the following section. In order to have more generalized architecture, and increase robustness of the architecture as a whole, we treat index lists as operators, too, i.e., they also implement the above shown virtual functions, entailing that we can call any of these function on index lists objects.

The names of the methods shown above are self-explanatory with regards to the functionality of the methods.

1. `getNextBlock()` function call results with operators scanning index lists for document blocks, and queue up the document blocks encountered, after path validation has been performed over element blocks contained withing the document blocks. After the call of `getNextBlock()`,
system spawns a number of threads depending on operator, where each operator scans index lists, as well as merges the items from this lists in parallel, furthermore path evaluation is also performed in parallel. We show in the sequel the pseudo-code for parallel query evaluation:

**Algorithm 3:** Dynamic document merge, and path evaluation

**Input:** getNextBlock()

**Output:** document queue \{d_1, \ldots, d_n\}

1. while \(\text{maxCandidateScore} \leq \text{topkList} \rightarrow \text{min} \rightarrow k\) do
2. \quad thread.start(blockMerge());
3. \quad thread.start(doXPath());

The method calls blockMerge() and doXPath() are executed in parallel, the first method merges all documents from document queues at nested or index list operators, merge is done based on document IDs, whereas the second method call evaluates XPath (staircase-join) only for leaf-level operators. The Alg. 3 is showing only logically how each method call is executed in parallel, therefore all details have been omitted for the sake of clarity. However, our system consists of much more complicated code and synchronization mechanisms for the multi-threaded architecture. Because the multi-threaded environment is very a non-deterministic environment, we have engineered carefully synchronization mechanism (i.e., my means of events and mutexes) in order to make the query execution flow coherent and deterministic, as if it was executed single-threaded.

2. getNextItem() function call returns items from the queue of the operator, if the queue is not empty, and if there were items that were not returned before. Actually this function call is very efficient, as it doesn’t make a copy of the object it returns (the queue item), but returns only the pointer to that item i.e., the memory address.

3. getHighBound() function results with the operator reporting the sum of the best-scores of the items it has in its queue.

4. prune(QueueItem *item) this function simply prunes the item it received in the argument list, namely by setting the item’s pointer to “null”.

52
7.2 Multi-Threaded Query Evaluation

In Sec. 6.3 we have presented the algorithmic backbone of the system for XQuery Full Text query processing. In this section we will go through some more details, namely distinct steps that the system takes in order to evaluate XQuery queries in ranked retrieval paradigm.

The user inputs XQuery FT expression, the system invokes XQuery FT parser (see Sec. 4.2.5) and decomposes that given query into a number of XPath expressions as shown in Alg. [1]. For a number of derived XPath queries $p_n$, the system creates $o_n + 1$ operators. Since our query parser is build in Java, after the system has finished parsing the query and decomposed it to $p_n$ XPath expressions, it then passes $p_n$ to the query processor (built in C++ for performance reasons) via a JNI interface. The query processor then fetches $p_n$ XPath expressions and creates $n$ light-weight operators and an additional fully-fledged top operator, hence $o_n + 1$, where 1 stands for the one extra top operator on top of the nested operators.

The difference between the top operator and the nested operators is in that, the top operator manages the execution flow of the algorithm, and maintains top-$k$ and the candidate queue. Nested operators serve the top operator with queue items, they only maintain one queue, that is, queue with items accumulated from the lower levels (i.e., index lists), but they do not maintain candidate nor top-$k$ queue.

The top operator is the main operator in the operator tree, it controls the execution of the query and halts the algorithm when top-$k$ items have been found. It is this operator that is responsible for pruning items, as only this operator can safely determine when to prune an item (i.e., when it is safe to prune an item and release valuable memory space). However, if the XQuery produces only one XPath expression, this can be the case when in the XQuery there is only one variable binder, then the query processor creates only one top operator and no nested operators, hence $o_1$. As an example, consider the query:

```xml
for $a$ in index(wiki.xml)\\ //article/
where $a$ ftcontains "Michael Schumacher"
return $a
```

This query results with one only XPath expression: //article[. ftcontains "Michael Schumacher"]. For our application, of more interest are the

1JNI : Java Native Interface
queries that produce more than one XPath expression, that is, we are interested in processing queries in multiple-operator fashion in parallel. Now consider the last query but refined to produce more than one XPath expression:

for $a$ in index(wiki.xml)//article/
let $k := $a/sec/
where $a$ ftcontains "Michael Schumacher"
and $k$ ftcontains "formula 1"
return $a$

Thus, this query yields the following XPath expressions:

//article/sec[. ftcontains "Formula 1"]
//article[. ftcontains "Michael Schumacher"]

Semantically, this query is looking for articles containing the Formula One driver “Michael Schumacher”, and preferably articles that in addition contain the keyword “Formula 1” in their child element section.

Since there are two XPath expressions, the system yields $o_2 + 1$ operators, namely two nested, light-weight operators and the additional top operator. Each of the XPath expressions is assigned to one of the nested operators. Each of the derived XPath expressions contain $m$ content, and $n$ structural constraints. After initialization, top operator calls the operator API function getNextBlock() on its nested operators $o_n$. Each operator, owning one XPath expression then starts scanning index lists $(L_1, \ldots, L_n)$ in interleaved manner, namely by calling the same function, but this time on index list objects, that is, the function getNextBlock() initiated by the top operator is propagated all the way down to the index list object. This function is executed as an independent thread, so that operators can run the same function independently, relying on thread-synchronization mechanisms applied at the higher level (i.e., by the top operator).

**Incremental merge-joins and path evaluation**

In what follows, we present the core algorithm for merge-joins and path evaluation in detailed steps. We demonstrate the functionality of operators’ API as shown in Sec. 7.1.
• Top operator propagates the method call `getNextBlock()` down on the operator tree, reaching index lists, i.e., each nested operator calls the very same method on its children operators.

• The first round of `getNextBlock()` method call, builds a candidate queue at each of the nested operators (nested operators execute this method in parallel). The method `getNextItem()` then iterates over the entire queue. Initially, all items in this queue are returned and propagated upwards via `getNextItem()`.

• Parent operator joins items based on document ID as they are fetched from the queues of its nested operators. We perform this join step by applying in-memory efficient $m$-ary merge joins in parallel. Where $m$ stands for query dimension in the case of leaf operators, otherwise $m$ denotes the number of nested operators.

• From the second round on, only a subset of newly inserted items or updated items are propagated upwards. If an existing queue item is encountered in another query dimension, its score is updated (gains more score), therefore this item needs to be propagated upwards to its parent operator, and all the way to the top operator. The latter needs to be informed about the new score, otherwise top-$k$ list does not reflect true top items. Subset of the new items are propagated anyways for exactly same reasons.

• New items need to be propagated upwards in any case, this way informing the upper operator about the new candidates.

• Each item has two flags, namely `isNew` flag which is set to true for all new items, and the `isUpdated` flag which is set true only for items that have their score updated.

• As document blocks are fetched from the index lists sequentially (by means of `getNextBlock()` method call, which in turn is executed in parallel at each operator), we test such documents against the existing documents in queue, namely by their IDs. If document IDs equal, then we aggregate the scores.

• After one round of merge joins (merge-joins are executed in parallel at each operator), we proceed with path evaluation. We check path on the enqueued documents by checking element’s pre/post order, and perform staircase-joins [22, 37] accordingly. This step is applicable only for operators at the leaf level, for XPath evaluation.
• The top operator maintains two queues, one for the current top-k items and one for the candidates.

• If the best score of some item is lower than the min-k, then this item can be safely pruned. The top operator tells its nested operator to drop this item by means of method call `prune(QueueItem *item)`, and this call is propagated all the way to the index lists. This way a valuable memory space is saved.

• The top operator calculates the bestscore by calling `getHighBound()` from its nested operators. This way it can dynamically compute the best score for each item.

• The `getNextBlock()` is repeatedly called until the stopping condition is reached (see Eq. (6.1)).

It’s worth noting, that we merge documents as we fetch them from the nested operators’ queues by applying efficient in-memory m-ary merge-joins. Because of the property that documents adhere, that is, documents in queues are sorted by their IDs in ascending order, we can perform in-memory merge joins incrementally (after each document block scan). Whenever we fetch a new document per distinct tag-term pair or structural condition, we test that document against existing set of documents already in the queue, if the document IDs match, we then compare the new set of element blocks against the existing set of elements blocks, thus testing structural conditions and aggregating scores. Scores are aggregated only in the case when pre/post conditions allow to do so. As in the previous example, the elements in documents fetched for tag-term pair `sec=formula1`, can be checked against structural constraint of `article`, if elements belonging to `sec` fulfill the condition of having `sec.pre > article.pre` and `sec.post < article.post`.

In the following sections we will show how scores are aggregated incrementally in the operator-based architecture, item pruning techniques, queuing at the operator level, score aggregation over score bounds at the top operator, and last but not least, top-k item filtering and candidate queue maintenance.

For this purpose we have shown a detailed figure in Fig. [7.1] depicting the final items that have made it into the top-k list, the candidate queue before having its items pruned, as well as index lists and queues at operator level. Please note, that at this point the system has already determined the top-2 items. The final index scan positions at each respective list are indicated with a green arrow, i.e., pointing to the last scan depth.
Figure 7.1: XQuery operator-based query evaluation in action
7.3 Queuing in hierarchical architecture

Each operator maintains its own queue for the items it has fetched from the inverted list. Operators are light weight operators, therefore besides the queue, they need not maintain top-k list nor candidate queue, see the queues at operator leaf-level as shown in the Fig. 7.1. At initialization stage, leaf-level operators are asked to start scanning inverted lists, each per tag-term pair and structural condition. In the Fig. 7.1, we have shown two such leaf level operators (i.e., nested operators), and underneath the operators, there have been depicted index lists, each with items in descending order of scores. We have simplified this example so that index lists contain only document IDs and their respective score in format: [document id, score].

After the first index scan round \( d = 1 \), the left operator scanned both its inverted lists in parallel and hence has fetched two documents (at pos. \( d = 1 \)), in Fig. 7.1 these two first documents are shaded in yellow color. The right nested operator in the other hand has fetched one document only (it had to scan only one list), namely \( d_{12} \). Nested operators continue scanning the index lists sequentially round after another in parallel, until the threshold conditions has been reached at the top-level operator, that is, until they are notified to stop scanning by the top operator. Normally by the time they are asked to stop scanning, they have accumulated a number of documents, and its crucial to prune early the items which cannot make it into the top-k list, this way we free valuable memory.

Each queue item has two flags, isNew, and isUpdated. These flags are the main mechanism for item’s score propagation, as the decision to propagate the score is based on these two flags (see alg. 4). Naively, one could propagate all the items after each index scan round, but this would only contribute to performance decrease and memory overhead.

**Definition 7.3.1 (New item):** If an item has not been seen before while reading items from the index list at each operator separately, then this item is considered new under the domain of the operator that has fetched it from the inverted list(s).

Initially all items are new by definition (see Def. 7.3.1), therefore they are propagated upwards, but once they are propagated their flag is set to false (see alg. 4). However, we could still see the same item in the other inverted lists, therefore such item has gained more score, and needs to be re-propagated upwards in the operator tree. See Algorithm 4 on how operators
Algorithm 4: Queue Management at leaf level operators

Input: query dimensions \( m \)
Output: Queue items

1. \( \text{indexDepth} = 0; \)
2. \( \text{while blockmerge} \land \text{indexDepth} < \text{index} \to \text{size}() \text{ do} \)
3. \( \quad \text{foreach dimension } \in m \text{ do} \)
4. \( \quad \text{item } \leftarrow \text{scanNewItem(indexDepth)}; \)
5. \( \quad \text{if Queue } = \emptyset \text{ then} \)
6. \( \quad \quad \text{Queue } \leftarrow \text{Queue } \cup \text{item}; \)
7. \( \quad \text{else} \)
8. \( \quad \quad \text{foreach it } \in \text{Queue do} \)
9. \( \quad \quad \quad \text{if it } = \text{item then} \)
10. \( \quad \quad \quad \quad \text{it } \to \text{score } \leftarrow (\text{it } \to \text{score } + \text{item } \to \text{score} ); \)
11. \( \quad \quad \quad \quad \text{it } \to \text{isUpdated } \leftarrow \text{true}; \)
12. \( \quad \quad \text{else} \)
13. \( \quad \quad \quad \quad \text{Queue } \leftarrow \text{Queue } \cup \text{item}; \)
14. \( \quad \text{foreach it } \in \text{Queue do} \)
15. \( \quad \quad \text{if it } \to \text{isUpdated } = \text{true} \lor \text{it } \to \text{isNew } = \text{true} \text{ then} \)
16. \( \quad \quad \quad \text{propagateUpwards(it);} \)
17. \( \quad \quad \quad \text{it } \to \text{isNew } \leftarrow \text{false}; \)
18. \( \quad \text{indexDepth } \leftarrow \text{indexDepth } + 1; \)

7.4 Score bounds computation in hierarchical architecture

Nested operators report the scores of their enqueued items to their immediate upper parent operator after the first index scan round. This way, such scores reach the top operator. The top operator maintains two priority queues, the queue with top-\( k \) items and the candidate queue. Both queues are shown in Fig. 7.1. Each item has its worst score higher than the min-\( k \) threshold is kept in the top-\( k \) queue (ties are broken arbitrarily), whereas the candidate queue is populated with items having their best score greater than the min-\( k \) threshold but having their worstscore lower than the threshold condition (see Fig. 6.2).
Queues at the top operator are highly dynamic, items’ scores change frequently, and items’ bestscore and worstscore start to converge fast, so does the min-k threshold increase. However, score computation at this level is done over score bounds that the top operator receives from the nested ones, which is still monotonic. In the example in Fig. 7.1, the bestscore of the item $d_{15}$ in the top list, has its worst score aggregated over the worstscores of the same item reported by the nested operators, hence $\text{worstscore}(d_{15}) = 0.8 + 0.6$. Best scores are computed dynamically (see alg. 2), since this item was seen in all dimensions, that is, in all operators, its best score is the same as its worstscore.

Since our operator tree allows for more than two levels of operators, and since we consider index lists as operators too (this way we generalize the infrastructure), score bounds of the items in queues are not aggregated at top level only. The same item can be seen in two different inverted lists, and the score of such an item should reflect the weight in terms of score accumulation from both lists. In the example in Fig. 7.1 in left side lower operator, we have denoted such items with the sigma symbol $\Sigma$ on top. Now, consider the document $d_{11}$, this item has been seen in both inverted lists shown under the lower left operator, therefore its score is $d_{11} = 0.3 + 0.5$, namely the score of 0.5 comes from the scan position $d = 3$ (see the green right pointing arrow in the left index list) and the score of 0.3 comes from the index position $d = 1$ at the right index list under the left lower operator.

### 7.5 Queue Item updating in hierarchical architecture

In the example in Fig. 7.1 the updated items are denoted with the ($\Sigma$) sign.

**Definition 7.5.1 (Updated item):** If an item has been seen before while reading items from the index list at each operator separately, then this item is considered **updated** under the domain of the operator that has fetched it from the inverted list(s).

Following Def. [7.5.1] and knowing the property that inverted lists have their documents sorted according the documents’ score in descending order, it is clear that every updated item has increased it’s score if it was seen in another dimension. This entails that the scores of the items that are updated at later index scans are not synchronized with the scores of the items already
reported to the upper operator, therefore the score bounds maintained at the
top operator are to be updated accordingly.

**Theorem 7.5.2:** Let \( L \) be an inverted list from the set of legal inverted lists \( \mathcal{L} \) with the property of having their items sorted in descending order of scores. Let \( n \) denote the number of elements in the list \( L \), that is \( n = \{L\} \). We say that for every item in the inverted list \( L \), its score is less or equal with the score of the item preceding it, but no item has a score 0. It follows that \( t_i \leq t_{i-1}, \forall t_i, t_{i-1} \in L \land \text{score}(t_i) > 0 \). We say that for any item that has been encountered in one of the index lists, if this item is to be encountered in another dimension, then its score must increase, applying the monotonic function \( \tau \).

Proof follows from the fact that items’ score in the list is non-negative and greater than zero. For any item encountered twice in two different indexes, its score is increased, therefore it needs to be propagated to the top operator for a sound top-\( k \) score calculation.

\[
t_i \in L_i \land t_j \in L_j, t_i = t_j \Rightarrow \tau(t_i, t_j) > t_i \land \tau(t_i, t_j) > t_j \quad (7.1)
\]

In the example in Fig. 7.1 there has been only one item updated, that is \( d_{15} \), by scan depth \( d = 3 \) (see green arrow in Fig. 7.1). This item was then again propagated to the top operator with its new updated score, therefore the top operator updates the bestscore and worstscore bounds accordingly. However, item \( d_{15} \) doesn’t make it to the top-\( k \) list, although it’s score was updated (increased), hence it is only listed in the candidate queue of top operator: \( \text{worstscore}(d_{15}) \leq \text{min} - k \).

### 7.6 Queue Item pruning in hierarchical architecture

The top operator knows all bestscores and worstscores of items it has accumulated. Based on the these two bounds, it can decide when to prune items, i.e., when it is safe to prune an item (when \( \text{bestscore}(d) \leq \text{min} - k \)), the top operator drops that item from the candidate queue, and calls the function \( \text{prune}(	ext{QueueItem* item}) \) (see operator API on sec. 7.1) on its nested operators recursively, and so do nested operators prune this item too, this way a valuable memory is freed. This operation is very fast, as we work only over the pointers of items. Nested operators propagate this call for pruning the
item, to their nested operators too, or index list objects, in the same way as the top operator. In the example in Fig. 7.1 by scan depth $d = 3$, the top operator prunes all of the items in the candidate queue, and the nested operators in turn follow the call and remove such items from their queues too.

![Figure 7.2: min-k threshold rise over index run (scan depth)](image)

With the increase of the scan depth, consequently the min-$k$ threshold (see Fig. 7.2) increases too, therefore many candidates “fall off” the queue ($\text{bestscore}(d) \leq \text{min} - k$), and if we were not to prune such items, then the memory requirements would be tremendously high, especially for queries with many content and structural conditions.

Relying only on the NRA algorithm, and dropping the items only when $\text{bestscore}(d) \leq \text{min} - k$ is considered too conservative, i.e., it requires deep index scans, which is a costly operation. The NRA baseline algorithm stops only when there are no more top-$k$ items to add to the top-$k$ list. But, the bestscores and worstscores of items in the NRA converge slowly, as result candidate queue items are kept in the queue for long time. We know that the bestcore bound for such candidate items is overly optimistic, that is, the true score of such items is often much lower than the bestscore. TopX can drop candidate items by estimating the bestscore of such items, i.e., by making use of histogram correlation, and guaranteeing a small deviation from the exact top-$k$ with high probability 38, 35.
\[ p(d) = P[ \sum_{i \in E(d)} s_i(d) + \sum_{i \in E(d)} S_i(d) > \delta], \text{having } \delta = \min - k \] (7.2)

Equation (7.2) is used to estimate the probability of the bestscore of an item being larger than the threshold. Such estimates are helpful for pruning items in advance, thus releasing the memory and increasing the efficiency in terms of run times. This saves a lot of disk I/O overhead. That is, if the probability of the bestscore of an item is lower than the threshold – min-\(k\), we can then drop such an item without having to scan index lists further and determine its true score. Of course such results are only approximative, and it is possible that we can miss some true top-\(k\) item, but query evaluation with probabilistic estimators helps terminate the algorithm faster compared to the conservative NRA algorithm, with a very small deviation from the true top-\(k\) list.

In the example in Fig. 7.1, the pruned items in the operator level have been marked with the red lightning (\(\mathcal{Z}\)) sign, but not based on estimations, but rather on their true bestscore bounds.
Chapter 8

Experiments and Results

In this chapter we show our concluded experimental results. To this extend, we compare the run times of evaluated queries in single operator and multi-operator fashion.

8.1 Index Construction and Data Collection


The original size of the corpus is 4.38 GB and contains more than 30 million elements, and on average an article has 161 nodes. We used no stemming, in order to produce faster and more precise top-k results.

Experiments in this thesis work were conducted on a server running 64-bit Windows 2003, AMD Opteron quad-core 2.6, with main memory - 16 GB.

8.2 Setup for the experiments

Our query processor was built in C++ using Microsoft Visual Studio 2008, with the parameters as shown in the following table:
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOCUMENT BUFFER</td>
<td>16384</td>
<td>reserved space for the number of documents</td>
</tr>
<tr>
<td>ELEMENT BUFFER</td>
<td>32768</td>
<td>reserved space for the number of elements</td>
</tr>
<tr>
<td>MAX BLOCKS</td>
<td>256</td>
<td>reserved space for the number of blocks</td>
</tr>
<tr>
<td>MAX QUEUE BLOCKS</td>
<td>8</td>
<td>reserved space for the number of queue blocks</td>
</tr>
<tr>
<td>MAX NESTED OPS</td>
<td>4</td>
<td>sets the maximum number of nested operators</td>
</tr>
<tr>
<td>NO CPU</td>
<td>8</td>
<td>sets the virtual number of CPUs</td>
</tr>
</tbody>
</table>

Table 8.1: Query processor settings

For evaluating TopX<sup>Q</sup>, we tested the system against two different sets of queries: (1) The INEX efficiency track query set [29], and the (2) composed query set, with \( k = 15 \) for both query sets.

8.3 INEX Efficiency track

The INEX efficiency track contained 115 topics (queries, type (A), classic ad-hoc queries). For each topic we have created one XPath expressions, that is, the keywords of each topic were added to the \( \text{ftcontains} \) clause, whereas for the structural constraint, the “//article” was added as prefix.

Each such XPath expressions was evaluated in single operator and in multiple operator mode. For the single operator mode, each XPath expression was executed as it is, without modifications, whereas for multiple operator mode, each XPath expressions was broken, and three new XPath expressions were derived (see Alg. 5). The derived XPath expressions all had the same structural constraint - “//article”.

The queries that had less than three keywords, we created two (i.e., when there were only 2 keywords present) and respectively only one operator (i.e., for cases when there was only one keyword in the query).

The comparison results for the run-times between the two modes of execution are shown in Fig. 8.1. The vertical axis denotes the runtime in milliseconds, whereas the horizontal axis denotes the number of keywords, in incremental fashion [1-10].
Figure 8.1: Query results comparison between queries evaluated in single and multiple operator mode

<table>
<thead>
<tr>
<th>Mode</th>
<th>Total Execution Time</th>
<th>Average Time per query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>24561 ms</td>
<td>213 ms</td>
</tr>
<tr>
<td>Multiple</td>
<td>40697 ms</td>
<td>353 ms</td>
</tr>
</tbody>
</table>

Table 8.2: Single Operator vs Multiple Operator

For the reason that queries are not very expensive (not that large), the single operator outperformed the multiple operator per same query set. However, as we shall see in the next section, for queries with “*” wild-card, the multiple operator outperforms the single operator when queries have more than 30 keywords, this happens as the query processor starts harvesting better run-times from the multi-threading, this way the synchronization costs are covered by a better load distribution.

The total and average run-times are shown in Tab 8.2. The column “Total Execution Time” denotes the elapsed time from the start of execution the query batch, until the last query from the batch has been executed, whereas the “Average Time” column, denotes the average time per query.
8.4 Composed query set

In addition we have evaluated the system against a composed set of queries (queries created by hand), varying in a number of keywords constraints and path conditions.

For evaluation of system performance, we have compared the run-time results of queries running in a single operator and multiple operator mode. That is, for a given query, we have first evaluated the query as it is in a single operator mode, with its all structural and content conditions, and then we have broken the query into at most three parts, namely we have derived at most three sub-queries from the original query (see Alg. 5).

**Algorithm 5: Decomposition of XPath queries for evaluation purposes**

Input: XPath query \( xp \)

Output: XPath queries \( \{\text{derived}_1, \ldots, \text{derived}_n\} \)

1. \( \text{structure} \leftarrow \text{getStructureFromQuery}(xp) \)
2. \( \text{keywords} \leftarrow \text{getKeyordsLength}(xp) \)
3. \( \text{subLength} \leftarrow \text{keywords}/3 \)
4. If \( \text{subLength} \geq 1 \) then
   - For \( i = 0; i < \text{subLength}; i++ \) do
     - \( \text{xpathQuery}[0].\text{append}(	ext{keyword.at}(i)) \)
   - For \( i = \text{subLength}; i < 2 \times \text{subLength}; i++ \) do
     - \( \text{xpathQuery}[1].\text{append}(	ext{keyword.at}(i)) \)
   - For \( i = 2 \times \text{subLength}; i < \text{keywords}; i++ \) do
     - \( \text{xpathQuery}[2].\text{append}(	ext{keyword.at}(i)) \)
5. Else if \( \text{keywords} == 2 \) then
   - \( \text{xpathQuery}[0].\text{append}(	ext{keyword.at}(0)) \)
   - \( \text{xpathQuery}[1].\text{append}(	ext{keyword.at}(1)) \)
6. Else
   - \( \text{xpathQuery}[0].\text{append}(	ext{keyword.at}(0)) \)
   - \( \text{counter} \leftarrow 0 \)
   - While \( \text{counter} \leq \text{xpathQuery.length()} \) do
     - If \( \text{xpathQuery[counter]} \neq \text{null} \) then
       - \( \text{derived}[\text{counter}] \leftarrow \text{structure} \cup \text{xpathQuery[counter]} \)
     - \( \text{counter} \leftarrow \text{counter} + 1 \)
7. Return \( \text{derived} \)

Consider the XPath query shown bellow:
In a single operator mode, this query would be processed as it is, whereas in the multiple operator mode, according to Alg. 5, this query would be divided into three distinct queries for parallel processing, as follows:

\[
//\text{article}\. \text{ftcontains "NASA space agency launched rocket Mars"}
\]

Furthermore, we have evaluated the query set into two different cache modes, namely in a cold-cache, and hot-cache. In the first case, query processor does not benefit in terms of cached items in the main memory, whereas the latter implies the opposite.

The Table 8.3 shows our complied query set, each query starts with the structural constraint //\text{article}. But we change this structural constraint to two more distinct cases, so that we can have a better overview of the query processing run-times under different constraints. However, the content conditions remain the same for all different structural constraints.

We consider in total three distinct cases:

1. Queries contain only one structural constraint //\text{article}, as shown in Table 8.3.

2. Queries start with the wild-card operator "\*", i.e., the wild-card operator can by matched by any structural constraint, therefore queries belonging to this type are very expensive. These are the queries from the same batch as shown in Table 8.3 but the //\text{article} structural constraint is replaced by the wild-card operator //\*.

3. Queries with an increased number of structural constraints, that is, //\text{article}//\text{sec}//\text{p}, same as in (2), now the //\text{article} constraint is replaced with //\text{article}//\text{sec}//\text{p}.

For the query type (1), the results are shown in the Fig. 8.2. We have evaluated the query batch with total of eleven queries as shown in the Table 8.3. The vertical axis (y) shows the run-times for queries in milliseconds, whereas the horizontal axis shows the number of keywords that queries contained in their \text{ftcontains} clause. Furthermore, in Fig. 8.2 there have been depicted
<table>
<thead>
<tr>
<th>No</th>
<th>XPath Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>//article[. ftcontains &quot;Penrose tiles tiling theory&quot;]</code></td>
</tr>
<tr>
<td>2</td>
<td><code>//article[. ftcontains &quot;tourism in tunisia sfax governorate tourist sousse tunisian tunis arabic agriculture lrm economy arabs sahara cities tunisians arab libya gafsa kairouan algeria coast mediterranean morocco monastir gulf country port governorates countries islamic&quot;]</code></td>
</tr>
<tr>
<td>3</td>
<td><code>//article[. ftcontains &quot;movie Slumdog Millionaire directed by Danny Boyle&quot;]</code></td>
</tr>
<tr>
<td>4</td>
<td><code>//article[. ftcontains &quot;mean average precision reciprocal rank references precision recall proceedings journal&quot;]</code></td>
</tr>
<tr>
<td>5</td>
<td><code>//article[. ftcontains &quot;chemists physicists scientists alchemists periodic table elements&quot;]</code></td>
</tr>
<tr>
<td>6</td>
<td><code>//article[. ftcontains &quot;opera singer italian spanish soprano&quot;]</code></td>
</tr>
<tr>
<td>7</td>
<td><code>//article[. ftcontains &quot;world wide web history hypertext cern protocol berners browser browsers cailliau worldwideweb usenet markup transfer ncsa technologies telnet internet mosaic w3c inria hypercard netscape gopher server spyglass&quot;]</code></td>
</tr>
<tr>
<td>8</td>
<td><code>//article[. ftcontains &quot;israeli director actor actress film festival&quot;]</code></td>
</tr>
<tr>
<td>9</td>
<td><code>//article[. ftcontains &quot;election victory australian labor party state council federal&quot;]</code></td>
</tr>
<tr>
<td>10</td>
<td><code>//article[. ftcontains &quot;applications bayesian networks bioinformatics&quot;]</code></td>
</tr>
<tr>
<td>11</td>
<td><code>//article[. ftcontains &quot;olive oil health benefit&quot;]</code></td>
</tr>
</tbody>
</table>

Table 8.3: Composed Query Set
run times for both query modes, single and multiple, and using cold and hot cache.

In Table 8.4 we have shown the total run-times, that is, the time it took to execute the entire batch. Same as with the INEX queries results (see Sec. 8.3), the column “Total Execution Time” denotes the elapsed time from the start of executing the query batch, until the last query from the batch is executed, whereas the “Average Time” column, denotes the average time per single query, the column “Cache” denotes the state of the cache. We can clearly see that run-times for queries executed with hot-cache are slightly better, although the difference gap would have been higher if we were to re-initialize the query processor for each query individually.

For query type (2), the results are shown in the Fig. 8.3. Clearly, queries starting with wild-card operators are much more expensive than those with
the **article** constraint, as can be seen in the figure. Run-times with query type (2) have significantly increased. Queries with large number of keywords, such as 32 (see Fig. 8.3), outperform the queries evaluated in a single operator mode. This happens for the reason that with such long queries, the multiple-operator gains advantage due to its multi-threaded execution of the queries. With more advanced hardware (e.g. server with 8 cores, recall that we tested on server with 4 cores), here we would have seen even better times for the multiple operator paradigm compared to the single one.

![Figure 8.3: Query results for queries starting with the * structural constraint](image)

For total and average run-times for the query type (2), we present the result in tabular fashion, analogously as we did with the query type (1).

<table>
<thead>
<tr>
<th>Mode</th>
<th>Total Execution Time</th>
<th>Average Time</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>51455 ms</td>
<td>4677 ms</td>
<td>Cold</td>
</tr>
<tr>
<td>Single</td>
<td>49017 ms</td>
<td>4456 ms</td>
<td>Hot</td>
</tr>
<tr>
<td>Multiple</td>
<td>62296 ms</td>
<td>5663 ms</td>
<td>Cold</td>
</tr>
<tr>
<td>Multiple</td>
<td>51747 ms</td>
<td>4704 ms</td>
<td>Hot</td>
</tr>
</tbody>
</table>

Table 8.5: Total run-time for the query type 2

We plot now the total and average run-times for query type (1) – **//article**, in single and multiple operator fashion. The results are combined from Tab. 8.4 and are shown in the Fig. 8.4.
Figure 8.4: Total run-times comparison for single and multiple operator mode

The vertical axis in the Fig. 8.4 similarly to other graphs, it denotes the running time in milliseconds, whereas the horizontal axis simply denotes the query type. It is clearly depicted that queries running in single operator outperform the multiple operator queries by factor of 2.

We plot now the total and average run-times for query type (2) – //*, in single and multiple operator fashion. The results are combined from Tab. 8.5 and are shown in the Fig. 8.5

Figure 8.5: Total run-times comparison for single and multiple operator mode
From the Fig. 8.3, we can see that the gap difference between single and multiple operator execution mode is not so big (i.e., not any longer by factor of 2). This happens for the reason that // queries are expensive to execute, therefore multiple-operator starts to gain momentum thanks to its parallel processing capabilities in multiple operator fashion.
Chapter 9

Conclusion and Future Work

9.1 Open Issues

In the current version, that is, in this thesis work we have implemented operator based architecture for XQuery Full Text evaluation in parallel. However, at the current version, we do not support complicated boolean search predicates, furthermore our query processor does not optimize multiple XPath expressions (the derived XPath expressions from the XQuery FT), so that the redundant paths are evaluated only once. Our query processor evaluates each XPath expression as it is, and does not delve deeper on its path constraints, in order to analyze and result with better query plans, as said the redundant paths could for example be evaluated only once per operator, and then join the results with other operators that evaluate the remaining constraints. Our architecture would perfectly allow for such optimization, and implementing such optimizations is trivial, and can be achieved during the query parsing, by omitting the redundant paths from the DAG trees.

Another more complicated scenario would rise in terms of cost of operation per operator. We could easily assign costs to each operator based on the query constraints, and then balance the load among operators, this way take advantage of the parallel processing that we have already implemented. A typical scenario would be the case when a given query would be very expensive to evaluate, we could in such cases piggyback some constraints for later evaluation, and present the results in different time intervals, that is, a more interactive query processing. If the user wished to have all the query constraints fully evaluated, then the user would be able to instruct the query processor to proceed with the remaining conditions. Another way
of dealing with expensive queries, would be to spawn more threads for a single such expensive query, i.e., segment the query further into more subparts, and evaluated each subpart as an independent thread. This technique would result with equal work units per operator, and our existing operator architecture would easily allow such equi-distribution.

9.2 Conclusion and Future Work

TopX$^XQ$ is a very efficient search engine, allowing for XQuery Full Text queries with nested multiple nested XPath Full Text expressions binded to user declared variables. Each of the nested expressions are evaluated in parallel in our generic operator based architecture, but the scores are computed globally, thus our systems yields with true top-$k$ items. TopX$^XQ$ is not limited to only nested XPath expressions in the XQuery Full Text, we can as well process independent batches of XPath expressions in parallel, thus resulting with much better run times than the run times when each of the XPath expressions were to be executed in singleton mode. Our architecture allows for evaluation of multiple twig patterns allowing for more than one output node, in contrast to the XPath where only one output node is allowed. Because XQuery Full Text allows for user declared variable binders, it is possible to form very complex queries by combining such variables efficiently. Such queries are very tedious to construct in the XPath language. Our architecture allows such complex queries, which in turn are decomposed, thus evaluated efficiently by our query processor in operator fashion.

It is important to note, that we process XQuery Full Text in ranked retrieval paradigm, that is, results returned by our query processors are ranked according to a defined similarity function. Furthermore, our architecture extends the NRA algorithm to the operator architecture, therefore our query processor terminates early, without having to scan exhaustively the inverted indexes, that is, when the “global” top-$k$ results have been found. Our query processor is not limited to processing inverted lists that fit only in the main memory, it can process XQuery Full Text queries over independent size of index lists, due to our novel block-based index structures.

For future work, we are aiming to implement the issues mentioned in the previous section [9.1], furthermore we are aiming to distribute the whole query processing infrastructure to multiple nodes, as our architecture is not tightly bound to the processing unit, hence would perfectly fit in such distributed system. In such a distributed environment, each of the nested operators
would become an independent processes, and the top operator acts as a bro-
er on top of such processes, where each process would be assigned to an
independent processing node.
Bibliography


79

http://www.w3.org/XML/

http://www.w3.org/TR/xpath20/

http://www.w3.org/TR/xquery/

http://www.w3.org/TR/2004/WD-xquery-full-text-20040709/


[28] Scott Hudson. CUP Parser Generator for Java. 
http://www.cs.princeton.edu/~appel/modern/java/CUP/

[29] INEX. Initiative for the evaluation of xml retrieval. 
http://www.inex.otago.ac.nz/.


http://monetdb.cwi.nl/

http://www-db.informatik.uni-tuebingen.de/research/pathfinder/


[40] Universität Trier. The DBLP computer science bibliography.  
http://www.informatik.uni-trier.de/~ley/db/.