Separating Key Concerns in Query Processing

Set Orientation, Physical Data Independence, and Parallelism

Vom Fachbereich Informatik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation

von

Dipl.-Inf. Sebastian Bächle

Dekan des Fachbereichs Informatik:

Prof. Dr. Arnd Poetzsch-Heffter

Prüfungskommission:

Vorsitzender:	Jun.-Prof. Dr. rer. nat. Roland Meyer
Berichterstatter:	Prof. Dr.-Ing. Dr. h. c. Theo Härder
	Prof. Dr.-Ing. Wolfgang Lehner

Datum der wissenschaftlichen Aussprache:

14. Dezember 2012

Acknowledgements

After crossing a finish line, one should not miss the opportunity to look back and revive the memory of all the good things that happened along the way. During my time in Kaiserslautern, I had the privilege to work with many smart, creative, and valuable persons, which supported my work and enriched my life in manifold ways.

First and foremost, I want to express my deepest gratitude to my advisor Prof. Theo Härder. His offer to join the DBIS research group opened me the door to an experience of personal and professional development from which I will benefit the rest of my life. He gave me the freedom to pursue my own research and I could always count on his support and learn from his great experience.

At this point, I also want to thank Prof. Wolfgang Lehner and Jun.-Prof. Roland Meyer for accepting the role as second examiner and head of the examination board, respectively. Their feedback and interest in my work before, during, and after my defense have been a great compliment to me.

Due to the great cooperative culture and personal atmosphere in Kaiserslautern, I can look at long list of colleagues and friends from which I have learned and who shared their valuable thoughts with me.

First, I want to thank the whole team of the *XTC* project, especially Christian Matthias, Andreas Weiner, and Ou Yi. Starting my research work in collaboration with you was a great experience. My special thanks go to my friend Karsten Schmidt for his consistent support in all areas of professional and personal life. His thoughts and advice greatly helped me to improve my work.

It is my pleasure to thank further members of the "old" and the "new" Teeecke generation, namely Andreas Bühmann, Jürgen Göres, Volker Höfner, Thomas Jörg, Daniel Schall, and Boris Stumm, for the many insightful and entertaining discussions about research and the real life. Furthermore, I appreciate the advice and experience that Prof. Stefan Deßloch shared with me.

My former students, the "Brackiteers" Max Bechtold, Martin Hiller, Roxana Gomez, Caetano Sauer, and Henrique Valer, passionately contributed improvements and extensions to the prototype implementation *Brackit* used in this thesis. I had a great time with all of them.

Outside the university, the strong and unconditional support of my parents Magda and Walter and my brother Matthias helped me to keep the balance during the hard times of this journey. I am blessed to have a family who always gives me the opportunity and the confidence to go my own way.

Finally, I owe my deepest gratitude to my beloved wife Susanne. She always believed in me and her patience and encouragement gave me the strength to stay focused during all the ups and downs. Knowing such a wonderful person at my side is invaluable.

Sebastian Bächle

Homburg, February 2013

Abstract

Declarative query languages are the most convenient and most productive abstraction for interacting with complex data management systems. While the developer can focus on the application logic, the compiler takes care of translating and optimizing a query for efficient execution. Today, applications increasingly call for declarative data management for many novel storage designs and system architectures.

The realization of a query processing system for every new kind of storage, language, or data model is a complex and time-consuming task. It requires considerable effort to design, implement, test, and optimize a compiler, which utilizes the system optimally. Thereby, a large part of the work is devoted to porting and adapting proven algorithms and optimizations from existing solutions.

This thesis studies the design of a compiler and runtime infrastructure for consolidating these development efforts. It aims at a decoupled organization of the main concerns of every query processing system:

Set orientation is a key concept for efficiently processing large amounts of data. We develop an intermediate representation for compiling queries and scripts with arbitrary nestings. It bases on the idea of composing higher-order functions to relational-style processing pipelines, which allow us to apply common set-oriented optimizations independently of the concrete data model used.

Physical data independence is mandatory for building a portable compiler and runtime. Our approach generally abstracts from physical aspects to cover a wide range structured and semi-structured data models. For efficiency, we present compilation techniques for tailoring and optimizing a query for a concrete platform.

Parallelism is crucial for exploiting modern hardware architectures. We present a novel push-based operator model, which uses divide-and-conquer and self-scheduling techniques for creating and controlling parallelism dynamically at runtime.

Contents

1. Introduction

The term *data management* classically centered around businesses and their information needs. In this setting, most data is homogeneously structured and data management is aligned to standardized processes, usually in form of sequences of short and well-defined business transactions. Along with data volumes and workloads, which grew fast but still at feasible rates, relational databases formed a long time the perfect basement for almost any kind of data-intensive application.

The internet changed this situation substantially. The scope of data management was and is more and more widened, reinterpreted, and redefined by the uprise of new applications with new types of data and workloads. Accompanied by tremendously increasing data volumes, new hardware, and mandatory connectivity and availability, this acts like a centrifugal force upon the industry. It led to the emergence of a whole zoo of new system designs, languages, and APIs, each optimized to store and process data for certain types of applications. The days of the ubiquitous relational database management systems seem to be over. In fact, they fall out of favor and diminish to a solution for their own niche of well-structured business data.

For developers of data management systems, this brave new world quickly turns out to be a curse rather than a blessing. Proven query processing concepts must be ported to every new platform, but without SQL and the relational model as given building blocks, they lack two of the most successful abstractions for reducing the complexity of managing large amounts of data. Furthermore, increasing numbers of paradigms, patterns, and APIs must be coordinated in the application code. Both, academia and industry, are therefore challenged to contain the growing complexity and consolidate the achievements of decades.

1.1. Language-supported Data Processing

Database management systems (DBMS) play a key role in the ecosystem of data-intensive applications. However, in face of the great diversity of technologies today, applications increasingly have to be built around highly specialized systems without knowing whether or not they will also be appropriate as the application evolves. A second, wide-spread limitation of current DBMS is their focus on plain query processing. They are not designed as a platform for running multi-step data processes. Extension mechanisms like user-defined functions/procedures (UDF/UDP), ECA engines (*E*vent, *C*ondition, *A*ction), and external library routines come to help, but in most cases developers will write separate programs and scripts to implement application logic and data analysis jobs, or just to provide the glue code for simple query and update sequences. This is not only cumbersome for ad-hoc tasks, it may also be less efficient if data must be transferred between the program and the DBMS.

In practice, both limitations require system architects and developers to carefully consider the properties and capabilities of each subsystem very early in the development process to choose the right mix of technologies. An appealing approach in the current situation is therefore to look at data processing from the opposite perspective, i.e., no longer bottom-up from the point of view of a particular data store, but top-down from the point of view of the application. Application design is then primarily driven by the application's *logical view* on data and not by technical aspects of a certain platform. Ideally, one does not need to decide about how to map application data to a specific storage model until all application logic is defined.

The growing number of data formats and systems available calls here for a suitable abstraction to implement data processing tasks without having to cope with proprietary or low-level aspects of a specific data management system. A unifying platform providing the ease and expressiveness of declarative query processing and naturally supporting the flexibility and scalability needs of modern data processing through proven database technology seems to be the new gold standard. The benefit of such an abstraction will be huge. Among others, it increases productivity, simplifies testing and tool development, and it reduces the coupling between applications and backend systems.

From an abstract point of view, a language for processing both structured and semi-structured data needs to fulfill some key requirements. First, it needs a set of primitives to compose and decompose data, i.e., to construct and access data of different granules and shapes. Second, it must conveniently support a common set of processing tasks present in almost any data-intensive application. The quadriga of operations is *filter*, *transform*, *combine*, and *aggregate*, which imply the support of basic features like arithmetics, Boolean logic, comparisons, etc. Third, all language constructs must be composable for implementing data-intensive multi-step processes and, of course, the language must support extensions to make custom functionality available. Virtually, this resembles to a marriage of basic programming and query processing.

Aside the difficulties to design a language for *data programming*, its success is clearly tied to the ability of translating it into meaningful operations on the underlying data, which utilize the data management system and its processing capabilities effectively. In other words, as the burden of dealing with system-specific and thereby also performance-critical aspects is moved away from the developer to the compiler, effective compilation and optimization techniques for the particular target platform are needed. However, with a great diversity of target platforms, less common features can be abstracted and exploited by a compiler. Aiming here solely for the least common denominator of all systems is certainly also the wrong approach. This work strives for a pragmatic solution for a reasonably large and practically relevant set of target platforms.

1.2. Motivation

The literature knows dozens of categorization schemes for programming languages. A language can be declarative or imperative, follow a functional philosophy or a procedural one, imply a structured or object-oriented style, etc. Real-world languages often incorporate traits of several paradigms at the same time, which makes it impossible to give a well-defined taxonomy. The actual difference between languages is explained best with the provided level of abstraction.

At this point, it seems a bit difficult to characterize a certain class of languages as *data programming languages*, because any program, independent of the language it is written in, consumes, processes, and produces some form of data. In this thesis, the term denotes languages which provide a level of abstraction according to the following definition:

> "A *data programming language* is a declarative query and scripting language for processing and manipulating structured and semi-structured data in various formats and in a broad spectrum of data processing systems."

The role of XQuery in this thesis should be only understood as that of a widely-known data programming language, which provides tailored syntax, data abstractions, type concepts, and auxiliary functionality for querying and processing XML data. In that sense, XQuery serves us as a concrete front-end language, which embodies the general concepts of a data programming language. In the following, we will contour the intention of a data programming language and substantiate the central points of the above definition.

Queries and Scripts

Conventional programming languages focus on algorithms and data structures, which rely on a properly-specified flow of control. In contrast, query languages provide a data-centric and result-oriented perspective without any notion of intermediate state or execution order. Hence, it is feasible to look at a query as a function, which returns a result for given input data set. In that sense, a functional program or a query respectively, is a formal description of a computational goal, which has to be realized by the language's runtime.

Scripts or batches of queries are handy for automatizing tasks consisting of multiple steps. Typically, individual steps will build on each other, e.g., input data is filtered first, then cleansed, transformed, combined, and finally aggregated. The result is returned to the application or written out to stable storage. Often it is possible to express everything concisely in a single query, but sometimes it is not, e.g., because the query language cannot integrate external tools for processing intermediate results. Sometimes it is just inconvenient to write queries which refer multiple times to the same data.

The essence of scripting-style query processing is the ability to chain operations and share intermediate results. In contrast to complex, event-driven workflows, most data processing scripts have a rather simple, linear structure and can run autonomously without waiting for additional user input. To some degree, this resembles functional programming languages, which model problems, i.e., programs, as series of function calls. Intermediate results appear here in form of function arguments and as local variables.

For most scripting tasks, it will be sufficient to support simple sequences of query statements. The result of each query q_i can be assigned to a variable v_i so that it can be used as input by any following statement $q_{j,j>i}$. Accordingly, we intend to support scripts of the following form:

$$\begin{array}{lll} v_1 & := & q_1; \\ v_2 & := & q_2; \\ \ldots & \ldots & \ldots \\ v_n & := & q_n; \end{array}$$

The restriction of scripts to linear sequences of statements is a concession to simplify reasoning about and optimization of data flows in a script. The abandonment of non-linear scripting constructs yet does not imply that the language will not be capable to carry out more complex tasks. Conditional branching, for example, can be realized by decomposing a script into separate parts; loop constructs and recursion can be encapsulated in recursive, user-defined functions.

Semi-structured Data

The nature of semi-structured data is not precisely defined in the literature. The long list of describing attributes reaches from structural properties like complex, nested, and irregular to classifications like schemaless, self-describing, partial schema-conform, and "fairly structured but rapidly evolving" [Abi97, AQM+97, Bun97]. One gets an intuitive understanding by explaining what is *not* considered as semi-structured. Structured data is widely seen as synonymous to relational data, i.e., well-specified collections of flat and uniformly-typed tuples, which can be queried and processed in a structured fashion. In contrast to this, unstructured data is coined to data which does not follow a partic-

ular structure and, thus, requires statistical analysis and information retrieval techniques. Examples are text in natural language and multimedia content like audio, video, and images. Semi-structured data locates somewhere between both extremes, but transitions are blurry.

The Web is a tireless producer of incredibly large amounts of semi-structured data in form of news feeds, data interchange messages, structured documents, log data, web pages, etc. Accordingly, one could characterize semi-structured data as collections of data items of a common super-structure but with individual diversity in form of irregular, incomplete and sometimes deeply-nested structures. The excerpt of an XML log file in Figure 1.1 gives an example of semi-structured data. All log entries share a common base structure, but individual entries differ in size and content.

Super-structures lend itself as anchors for locating data of interest. Common properties like the timestamps and IP addresses in the sample log data allow for systematic filtering and grouping of related objects. Structurally heterogeneous parts are better matched against patterns to extract information. In the worst case, the actual shape of data is a priori completely unknown and one must fall back on full-text search and similar techniques.

The XQuery in Figure 1.2, for example, uses the basic structure of log entries to filter them by date and severity, but uses a path expression as pattern to search for message strings in the contents. The resulting overview reports for each system are shown in Figure 1.3.

Formats, Systems and Platforms

The processing of semi-structured data reveals substantial similarity to relational query processing with explicit filter, sort, and aggregation steps as shown in the example query above. Accordingly, it is worthwhile to reuse the algorithms and techniques from this area to reduce the number of additional concepts to a minimum. Nevertheless, a query processor has to go some new ways abroad the classic relational path to cope with structural complex and heterogeneous data. The goal of a data programming language is to find a compromise between data abstraction, language constructs, and typical processing needs that is applicable to various formats, data stores, and platforms.

```
<log tstamp="2012-04-27+08:49:012" severity="critical">
  <src>192.168.12.31</src>
  <watchdog pid="47232">
    <event>possible attack detected</event>
    <msg>lock user "admin" for domain "evil-empire.org"</msg>
  </watchdog>
</log>
<log tstamp="2012-04-27+09:00:002" severity="low">
  <src>192.168.14.132</src>
  <msg>finished system update: no update required</msg>
</log>
<log tstamp="2012-04-27+08:48:991" severity="high">
  <src>192.168.12.201</src>
  <watchdog pid="3241">
    <event>service not responding</event>
    <msg>restarting system service "clustermgnt"</msg>
  </watchdog>
</log>
<log tstamp="2012-04-27+08:48:733" severity="mid">
  <src>192.168.12.31</src>
  <auth action="login" result="fail">
    <user>admin</user>
    <domain>evil-empire.org</domain>
    <protocol>ssh</protocol>
    <msg>authentication failure: user "admin" ; attempt: 5</msg>
  </auth>
</log>
```

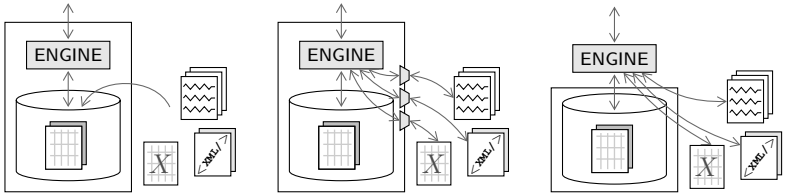
Figure 1.1.: Example of semi-structured log data in XML format.

```
for $e in collection("log.txt")
let $src := $e/src
let $info := <info>{$e//msg/text()}</info>
where $e/@tstamp > "2012-04-01+00:00:000"
  and $e/@severity = ("critical", "high", "mid")
order by $e/@severity
group by $src
return <report system="{$src}" incidents="{count($e)}">
          { $info }
       </report>
```

Figure 1.2.: Sample XQuery for XML log data.

```
<report system="192.168.12.31" incidents="2">
  <info>lock user "admin" for domain "evil-empire.org"</info>
  <info>authentication failure: user "admin" ; attempt: 5</info>
</report>
<report system="192.168.12.201" incidents="1">
  <info>restarting system service "clustermgnt"</info>
</report>
```

Figure 1.3.: Output of sample query on XML log data.



(a) transform and load (b) wrap and extend (c) extract and process

Figure 1.4.: Query processing across different data sources.

Todays relational DBMS are sophisticated experts for processing tabular data residing in their own data store. Traditionally, they provide only limited support for data in foreign structures and formats. If an application needs to process external, possibly heterogeneous data, e.g., in form of plain text files, CSV data, XML, or binary files, it requires considerable effort to facilitate the processing capabilities of an SQL-based DBMS.

Conceptually, there exist three basic strategies for joint processing of external data and data in a DBMS as depicted in Figure 1.4. If the query engine is embedded in the DBMS, either all data needs to be converted into an appropriate relational representation and imported into the system, or the query engine interacts with external data through a mix of adapters, extension modules, and stored procedures. In practice, however, it is often simpler to process all data outside the DBMS with a dedicated ETL tool (*E*xtract, *T*ransform, *L*oad), shell script, or custom application code.

A data programming language will be deployable in all three scenarios rendered. It abstracts from a concrete data format and allows to express data processing tasks at an abstract level without the need to take low-level and system-specific aspects into account. Ideally, a platform-independent compiler kernel will focus on the optimization of storage-independent query logic and delegate the final mapping from the logical view on data to its physical representation, e.g., in a DBMS, to platform-aware extensions. Such extensions may not only implement basic data access routines, but also delegate subqueries or entire queries to the respective data sources, thus, maximizing potential performance gains of native processing. Uncovered operations or otherwise lacking functionality of the storage layer needs to be filled in by generic variants implemented in the language runtime.

1.3. Contributions

This thesis studies the realization of a portable compiler and runtime platform for processing structured and semi-structured data. Concretely, it addresses the compilation of an extended variant of XQuery as being a well-suited candidate for a real cross-platform query and data programming language. Although it is primarily intended to query XML data, it is nevertheless a suitable basement for this work, because it is capable to represent, interpret, and process different kinds of information from diverse sources.

The main parts cover the translation and evaluation of basic language concepts like bulk operations, loop nestings and data access primitives. Special consideration is given to three central challenges: set-oriented processing of structured and semi-structured data in heterogeneous formats, support of platform-specific features, and automatized parallel processing. Accordingly, most of the findings and concepts developed are not inherently specific to XQuery/XML and can be generalized.

An additional aspect of this work is the extensive evaluation of the former concepts, e.g., inside a native XML database management system, where XQuery/XML are first-class citizens. Besides providing deeper insights into application and effectiveness of certain compilation techniques, this part puts the consideration of environmental aspects like external memory in the center of interest.

In summary, the main contributions of this thesis are:

- A retargetable compiler framework and runtime environment for an XQuery-based data programming language with a clear-cut separation of language-inherent aspects from physical aspects of the underlying data store
- A hierarchical intermediate representation for queries and script-like processes inspired by functional paradigms
- A novel push-based operator design for dynamic parallelization of queries based on divide-and-conquer and work stealing techniques
- Experimental evaluation of all concepts presented in various settings including a full-fledged prototype of a native XML database management system

The major intention of this work is the development of architectural strategies and design principles rather than a specific system or algorithm. Nevertheless, we will sometimes need to discuss certain aspects of the latter in detail because it helps to clarify the rationale behind design decisions or because common algorithms are accompanied with substantial implications on runtime, which must be taken into account.

1.4. Limitations

This work does not claim that the presented compilation techniques always yield to optimal solutions across all logical and physical aspects of a query. A holistic approach may have the potential to outperform some of the presented optimization strategies, but it would be highly complex and therefore of doubtful practical relevance.

Furthermore, this thesis does not proclaim XQuery as well-suited for any data processing task. Instead, it tries to explicitly disclose use cases which do not fit well into the picture like some applications for scientific data. Also, this work does not attempt to provide a solution for schema mapping and information integration in general.

To avoid any misconception, some topics should also be mentioned, which are related to this thesis, but *not covered*, because they are considered as orthogonal or out of focus:

- General programming language design and compiler techniques (e.g., typing, memory management, or code generation in general)
- Statistics-based and cost-based query optimization as well as techniques for automatic tuning and robustness of query plans
- Distributed data and distributed query processing
- Machine-specific optimization (e.g., memory, I/O, SIMD) and specialized hardware (GPU, FPGA, etc.)
- Intrinsics of parallel shared-memory architectures like scheduling, core-to-core communication, and memory management

Whenever necessary or helpful for presenting or understanding the material in this thesis, however, some of this aspects will be commented on, but without intending to discuss an issue exhaustively.

1.5. Outline

Chapter 2 introduces design principles and core language constructs for declarative query processing and scripting. Thereafter, Chapter 3 gives an overview of XQuery, which will be used in this thesis as concrete instance of the abstract data programming language defined before.

In Chapter 4, we present a hierarchical query representation which is derived from the functional core of the abstract language. We also outline individual query rewriting and optimization phases of the compilation process.

A detailed discussion of the central compilation stages addressed by this thesis follows in the chapters 5 and 6. They present techniques and rewriting rules for optimizing queries with respect to set-oriented and physical aspects, respectively.

Chapter 7 introduces a novel pull-based operator model, which allows to automatically exploit data-parallel sections in a query.

The concepts and techniques developed in this thesis are empirically evaluated in Chapter 8. Related work is reviewed in Chapter 9.

Finally, this thesis closes with a summary and an outlook on future work in Chapter 10.

2. Anatomy of a Data Programming Language

For a better understanding of the various challenges addressed in this thesis, we start with an introduction of the ingredients of a data programming language. The primary goal of this chapter is to provide insight into central aspects like a basic data model, core operations, and bulk processing logic rather than giving a full language specification or syntax. Thus, the presentation will stay at an abstract level, but with references to the XQuery language specification as source for concrete specifications. In Chapter 3, we will find the application of the concepts presented in a more specific fashion, geared for XML support.

2.1. Data Model

The heart of every query language lies in the underlying data model. In the classic definition, a data model consists of three parts: a set of types and structures, a set of operations, and a set of constraints [Cod80].

The types and structures available in a data model determine its expressiveness, i.e., the ability to represent information in a concise and logically coherent form. Practically, they define the shape of data and thereby influence the design of applications and physical storage.

To work with instances of types and structures, the data model must define operations on them. All operations must be closed within the data model, i.e., all values produced must also be valid instances of the data model. A few basic primitives are usually sufficient to provide the logical basement for building convenient APIs and query languages at a higher level of abstraction.

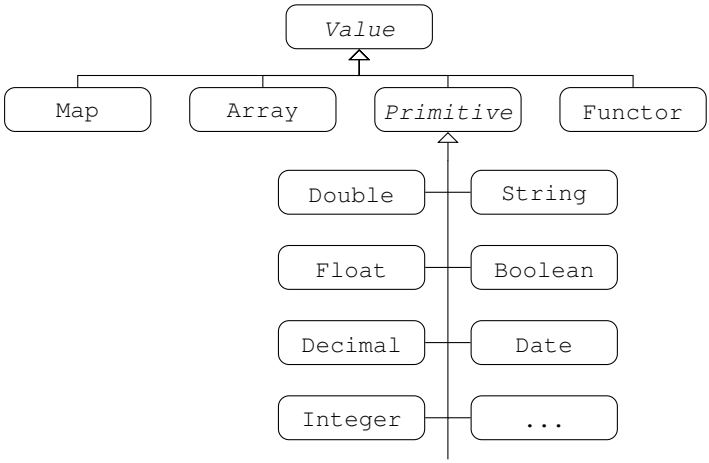


Figure 2.1.: Value types in a data programming language.

Constraints enforce well-formedness and integrity of data in a data model. Generally, they are divided into two classes. Value-space constraints restrain the domains of primitive values like integers, strings, etc. Structural constraints specify the range of valid combinations of structures and the relationships between them. Some data models also build on combinations of both.

For a data programming language, it is of utmost importance that the underlying data model conceptually suits common shapes of data as found in programming languages and serialization formats. Otherwise, the language loses too much of its advantages and expensive user-defined mechanisms are required to convert the input to a suitable representation beforehand.

2.1.1. Values

A value is the logical unit or granule of information in the data model. An overview of the basic value kinds for modeling semi-structured data at different levels of abstraction is given in Figure 2.1.

Primitive Values

Atomic units of data are strings, Boolean values, integers, floating point numbers, etc. In principle, many kinds of values may be considered as primitives when necessary. Query languages, for example, often include support for dates, timestamps, and binary data. As this thesis builds upon XQuery, we stick to the atomic value types defined in [W3C04b].

Complex Values

Complex values are composites of primitive values and other composites, which represent abstract concepts in applications like accounts, log records, and orders. The only two composition types required in our abstract language are *array* and *map*.

An array is an ordered sequence of values. Individual values may be of heterogeneous types and may occur multiple times in an array.

A map represents an ordered mapping of primitive values, called keys, to other values. Keys are required to be distinct and the order of all keys in a map is stable. The enforcement of primitive keys is a concession to facilitate efficient implementations of map values. In cases where the application requires map values with non-primitive keys, one can define primitive surrogate keys for complex values.

Some query languages for semi-structured data include bags and sets as explicit, order-insensitive composite types [ORS+08]. Practically, they can be considered as special cases of arrays, where conceptual differences primarily become relevant when constructing new values. In this sense, arrays could also be interpreted as a special case of maps with densely ascending integers as keys. Because of their practical relevance and performance-relevant implications, however, it seems wise to treat arrays as basic building blocks, too.

Composite types may be arbitrarily nested, but the nesting must not be cyclic. For example, an array cannot be a field value of itself. In other words, complex values must form a tree of values; pointers are not supported.

Note, depending on the context, a complex value may be considered as a value of its own or as a composite of individual values. An XML element, for example, can be referred to as an individual node and as representative of a whole XML subtree.

Functors

A functor or function value is a callable function object. It embodies a function definition or algorithm that can be applied to a given set of arguments and can be used like any other value type, e.g., as field value in arrays and maps. The arguments and the return value of a function can be values of any type, including other functors.

The explicit representation of function values enhances the flexibility and extensibility of the language, but does not introduce higher computational expressiveness. It is also not a typical requirement in a query language, but it is of high practical value in a data programming language. Higher-order functions can, for example, be exploited to plug-in custom functionality in standard query constructs. In combination with complex values, they allow to encapsulate related data and operations similar to object-oriented programming languages [Weg87].

Null

Incomplete or unknown information is represented through the special value *null*. Its use may or may not be tolerated in individual situations or data structures. In comparisons and predicates, *null* is used for implementing a three-valued logic [Cod90].

Identity

Neither primitive nor complex values have an identity. Primitive values are considered equal when they represent the same value of the same basic unit type. Equality of complex values is defined recursively. Two complex values are considered equal when they are "deep equal", i.e., their sub-components are pair-wise equal.

Identity concepts at a higher level of abstraction like primary keys or node identity in XML can be achieved by adding and enforcing explicit or implicit identity values to a structure.

Relationships

A complex value stands in a natural parent-of relationship to the individual parts it consists of. In conjunction with the ordering of child values, the relationship defines the basement for any operation in the

data model. Further structural relationships can be inductively defined whenever necessary.

The parent-of relationship is asymmetric, e.g., it is possible to get the values of a map by their keys, but it is not possible to obtain all maps referring to a particular value. If reversely-directed relationships are needed, e.g., if values must be able to refer to a complex value as their parent, it must be modeled by adding value-based identity and foreign-key concepts to the parent and the children, respectively. Alternatively, the relationship can be emulated with an operation, which returns the parent value by selecting the value which contains the respective value as a child.

2.1.2. Types and Schema

The presence of a type system is – as in any language – devoted to numerous advantages. Most importantly, it serves as an abstraction to reduce complexity within the language itself and to simultaneously increase the modularity of software written in that language. Besides, static and dynamic type checking help to ensure safety and correctness of a program and enable compilers to reason about data and exploit this information for optimization.

Types appear as an additional dimension of values. Every value has an annotated type, which becomes an integral part of the value itself. A type system defines a hierarchical is-a relationship among types. This facilitates techniques like polymorphic subtyping and implicit conversion of values, e.g., for function calls [Lie87, LS88]. The basic type hierarchy is identical to the value hierarchy shown in Figure 2.1. Extension points allow the introduction of additional application-specific or platform-specific subtypes through domain restriction and structural constraints for primitive values and composite values, respectively.

Extending the Type Hierarchy

To leverage typing information of a specific data model like the relational model or XDM, it must be tightly integrated in the type hierarchy. The straightforward solution models the building blocks of a concrete data model as subtypes of the existing primitive and complex types. But if a new type should reflect a completely distinct kind of data, a

more intrusive integration strategy must be pursued. The new type is then modeled as a separate top-level branch in the type hierarchy. Values thereof then appear as separate types, even if they embody the same composition types and primitive values as types in other branches. The extended XQuery data model presented in Section 3, for example, incorporates with XML and JSON two completely different abstractions of semi-structured data. Conceptually, however, XQuery treats all values as sequences and thus operates only on array composites.

Technically, both extension strategies are feasible, because types are solely relevant in the context of type-sensitive operations. At the physical level, a single implementation of maps and arrays is sufficient to represent values of any type. Ideally, however, the compiler is equipped with tailored implementations for concrete abstractions (e.g., XML) and leverages the capabilities of specialized storages.

Schema Information

The integrity of a value is not automatically guaranteed. Assume a type that represents relational tables as two-dimensional arrays of primitive values. For values of this type, we should enforce consistent sizes and value types across rows and columns. Normally, such conditions will have to be checked when a value is instantiated. However, application-level aspects like types of column values and foreign-key constraints, are usually out of reach of the instantiating operation. Informally, we denote such a set of domain-specific type definitions and invariants as schema.

In many query languages, especially those for homogeneously structured data, schema information is mandatory and often an integral part of the storage layout. Invariants like uniqueness constraints and foreign keys may then be eagerly exploited to derive better execution plans. In the field of semi-structured data, however, proper schema descriptions are less common. In most scenarios, e.g., ad-hoc data analysis, schema knowledge is encoded directly in the data processing logic. Schema descriptions and schema languages are merely adopted in standardized application domains or large applications. Consequently, a query processor should be able to leverage complete or partial schema information when available, but must primarily deal well with schema-less data.

This work focuses on query compilation and evaluation without explicit schema information, but the presented techniques can be anytime refined and complemented for the integration, verification, and exploitation of schema knowledge. For our purposes, we assume that the burden of ensuring data integrity and validity of queries is put on the developer.

2.2. Basic Language Features

The core features of a data programming language can be grouped into three categories: Creation and manipulation of data, primitives for computations and calculations, and extensibility aspects.

2.2.1. Composition and Decomposition of Data

Almost every operation in a query involves the inspection, interpretation, and combination of data. Accordingly, creation, composition and decomposition of primitive and complex values are key tasks.

Constructors

Operations for creating primitive values and complex values are called constructors. In a query, constructors are invoked explicitly via function calls and dedicated language constructs, or implicitly when a value is computed as intermediate result. If the type of a newly created value needs to obey certain constraints, the constructor must also perform the respective consistency checks.

Primitive values are either directly parsed or converted from a text or binary representation, e.g., for literals in the query string or for results of standard operations like arithmetics, string operations, type conversions, etc. Complex types may be composed from any given set of primitive and complex argument values. In every case, however, complex values must base on the composition types array and map.

In principle, the same kinds of values can be created through an arbitrary number of different constructor functions. Multiple constructors for a particular value kind may be provided, e.g., to support syntactical variants, to enforce certain type properties, or to expose performance- and storage-relevant properties.

Value Type	XQuery/XDM	JSON

Primitive	`1`
	`3.14159e0`
	`"a string"`
	`xs:boolean("true")`
	`xs:date("2012-07-21")`
		`1`
		`3.14159e0`
		`"a string"`
		`true`

Null value	`()`	`null`
Array	`(1,"2",3.0)`	`[1,"2",3.0]`

Map	`<e a="1">`
	`  <c>a child<c/>`
	`  some text`
	`</e>`
	
	`element e {`
	` attribute a { 1 },`
	` element c {`
	`  "a child"`
	` },`
	` text { "some text" }`
	`}`
		`{`
		` "e": {`
		`  "attrs" : [{ "a" : 1 }],`
		`  "chlds" : [`
		`  { "c" : {`
		`  "attrs" : [ ],`
		`  "chlds" : ["a child"]`
		`  }`
		`  },`
		`  "some text"`
		`  ]`
		`  }`
		`}`

Table 2.1.: Examples of data constructors for XQuery/XDM and JSON.

Whenever feasible, examples in this thesis use the simple syntax of the data serialization format JSON [Cro06] for representing array and map values. Arrays are specified as comma-separated lists of values enclosed in squared brackets `[]`. Maps are specified as comma-separated lists of key/value pairs enclosed in curly braces {}. Keys and values are separated by a colon (`:`), respectively. Table 2.1 exemplifies syntax variants for constructing primitive and complex values in XQuery/XDM and JSON.

Path Operations

The counterpart to constructors are functions and other built-in facilities for decomposing complex values, i.e., accessing individual parts. When they are combined, they virtually reflect navigation through tree-like structures of complex values. Field accesses are therefore commonly called path operations.

As the data model consists of only two composition types, there are solely two corresponding types of access operations. Both, arrays and maps, support associative field access by position and key, respectively.

The fields of an array can be accessed with the binary operator `[[]]`. The first parameter is the respective array, the second is the position of the array field. In typical array syntax, the second argument is enclosed inside the squared brackets[1], i.e., `x[[y]]` returns the `y`-th element of the array `x`. As usual, the numbering starts with 0 so that `["a","b"][[0]]` will return `"a"`.

A mapped value can be obtained through the binary operator `=>`, which uses the second argument as lookup key for the map provided as first argument. The expression `{x:2,y:3}=>y`, for example, yields the value `3`. Left-precedence and infix notation foster convenient chaining to navigate arbitrarily deep paths of the form x_1`=>`x_2`=>...=>`x_n. Accordingly, `{point:{x:2,y:3}}=>point=>y` yields the `y` value `3` of the given point record.

Advanced Path Operations

As said, accesses to map values and array fields mimic a drill down into a tree-like structure. Generally, they are sufficient to explore arbitrary nestings and relationships between values. However, if a query needs to evaluate more elaborate search patterns, it quickly gets complicated to express this as a combination of simple down steps and supporting predicates. Very likely, the evaluation of complex compound patterns has an impact on performance, too.

To facilitate a crisp notation of complex search patterns and to open the door for better performing search routines, front-end languages can augment further access operations, as it is the case in XQuery, respectively its subset XPath. For navigating XML trees, they define 7 forward axes like `child`, `descendant`, and `following-sibling`, and 5 backward axes like `parent`, `ancestor`, and `preceding` [W3C10a, W3C07]. The same way, path operations can be enriched with support for wildcards, type matching, etc.

[1] The notation with two squared brackets was chosen to avoid confusion with XQuery predicates and filters.

Auxiliaries

Sometimes it is necessary to inspect unknown structures in a query. To complete the picture, we make therefore use of two auxiliary functions `entries()` and `length()` to get a list of key/value pairs and the number of elements for maps and arrays, respectively.

At the physical level, both constructors and path operations must operate on a common representation. However, if multiple data sources and different storages are used within the same query, there are always pairs of data construction operations and data access operations that belong together. In such cases, a query processor may utilize a generic access interface and appropriate adapters for each storage. A performance penalty through adapters may be avoided if operations can be safely confined to a particular storage. However, this requires careful analysis of the data flow in a query as detailed in Section 6.

2.2.2. Core Operations

As in any query language, a comprehensive tool set of core operations like arithmetics, comparisons, conditional branching, and type conversion, is necessary to carry out meaningful tasks. Without going into details, we assume to have common standard functionality available as defined in XQuery/XPath [W3C10a, W3C11a].

Coercion

The working principles of basic operations on primitive values, e.g., arithmetics, is folklore. However, their application for non-primitive values requires further specification. For example, a comparison `a>b` is well-defined if `a` and `b` are numeric values, but its semantics is unclear if one or both of them are complex values.

If an operation is considered invalid or forbidden for particular combination of value types, an error should be raised. Alternatively, a complex argument value can be coerced to a primitive value for which the respective operation is defined [Abi97]. For example, if `a` and `b` in the above example were two arrays `[1,2,3]` and `[2,3,4]`, one could coerce both arrays transparently to the sum of their elements, yielding the valid comparison `6>9`.

Clearly, automatic argument conversion is primarily a comfort feature and not a technical necessity. A front-end language may specify meaningful coercion rules for certain operations and value types but also raise typing errors. Again, we assume in this work intuitive type conversion and coercion rules like in XQuery/XPath [W3C10a, W3C11a].

Platform Specifics

Another peculiarity of core operations arises at the implementation level. The support of heterogeneous target platforms requires consistent treatment of all system-specific aspects like arithmetic overflows, rounding, character sets, character encodings, byte ordering, etc. XQuery refers here to widely-used industry standards to define the expected behavior for operations on built-in primitives [W3C04b, W3C11a].

2.2.3. Function Calls

The support of built-in and user-defined functions is of essential utility. Built-in or system-defined functions primarily implement basic or low-level functionality like I/O, string manipulation, date operations, etc. Additionally, built-in functions may be used to make proprietary capabilities of the target platform accessible. User-defined functions encapsulate parameterized queries and utility operations, which allows for better code reuse and maintainability.

A function can take any number of input parameters and produces a single result value. When a function is called, the runtime is allowed but not required to handle the arguments in a non-strict fashion. For large input arguments, the query processor can then facilitate memory-efficient pipelining techniques instead of computing each argument entirely beforehand. In addition to that, lazy argument handling also brings the freedom to use arbitrary forms of directly and mutually recursive functions. For performance reasons, a function implementation may also take advantage of lazy evaluation and compute large results incrementally, too.

```
declare function sum-greater-3-rec($val, $pos, $end, $acc) {
  if ($pos > $end) then
    $acc
  else if ($val[$pos] <= 3) then
    sum-greater-3-rec($val, $pos + 1, $end, $acc)
  else if (empty($acc)) then
    sum-greater-3-rec($val, $pos + 1, $end, $val[$pos])
  else
    sum-greater-3-rec($val, $pos + 1, $end, $acc + $val[$pos])
}
```

Figure 2.2.: Recursive function to sum up all values greater than 3.

2.3. Bulk Processing

The language features described so far are already very powerful. Conditional branching and recursive functions yield a Turing-complete language, which allows to perform arbitrary computations. However, it is not yet sufficient for effective data processing.

The handling of serious data volumes depends on efficient algorithms and careful resource management. Hence, a compiler must be able to reason about a query, its use of data, and its runtime properties. It must recognize data flows, patterns, and dependencies to assemble filter, join, and grouping algorithms to an efficient execution plan.

It is difficult, if not impossible, to extract the actual intention of a query from arbitrary functions. Even the most basic query pattern is hard to identify without appropriate support from the language itself. Consider the function in Figure 2.2. It uses a non-trivial recursion to compute the sum of all elements in the input sequence `$val` that are greater than `3`. It requires sophisticated analysis to recognize that parameters `$pos` and `$end` are used to scan sequentially through the input sequence, and that parameter `$acc` is the accumulator variable holding the current sum.

Obviously, the excessive use of recursive functions is barely efficient and often cumbersome to write, too. Dedicated language constructs for expressing data-intensive processes without recursion can fill this gap. The explicit specification of bulk operations and data flows simplifies analysis, optimization, and code generation dramatically. Internally,

bulk operations look like higher-order functions whose special role is known by the compiler. This avoids the need to introduce completely different language concepts and simplifies the compilation process.

A bulk processing function operates in a relational fashion over sets of tuple-like structures. We assume here the simplest form of relational structures, i.e., two-dimensional arrays of values. Each function accepts at least a two-dimensional array as input and produces a two-dimensional array as output. This makes the mapping to efficient tuple-oriented techniques and algorithms straightforward.

Without intending to provide an exhaustive list of all advisable bulk operations, this work focuses on three groups of standard operations as listed in Table 2.2. In the following, we will denote bulk functions as operators to emphasize their special role as internal query constructs and use upper-case names to distinguish them from normal functions. An upper-case type parameter `T` stands for a whole list of columns, i.e., a function `[T]→[[S]]` takes an array with columns t_1, t_2, ..., t_n and returns an array of arrays with columns s_1, s_2, ..., s_m.

2.3.1. Transformation and Filtering

The first group of operators transforms and filters a given input relation. The functions can be characterized as encapsulations of looping primitives, which iteratively process the input relation row by row.

The `ForEach` operator works similar to the map pattern in functional programming. It computes a sequence of Cartesian products where one input can be functionally dependent on the other. The function `bind` is applied on each tuple `t` in the input relation `in` to produce a second relation, which is used to compute the cross product with the respective input tuple. Figure 2.3 illustrates the working principle of `ForEach` for a simple `bind` function, which maps the first column value of a tuple `[x,y]` to a relation of two tuples `[x+1]` and `[x+2]`.

The relative order between input tuples is preserved in the output, reflecting a nested-loops-like evaluation strategy. However, if a query permits permutations of tuples in the output relation, the query processor may exploit this, e.g., for parallelism, and produce a randomly ordered output.

Operator	Parameter		Return Type
ForEach	in	[[T]]	[[T,S]]
	bind	[T]→[[S]]	
Project	in	[[T]]	[[S]]
	proj	[T]→[S]	
Select	in	[[T]]	[[T]]
	pred	[T]→bool	
OrderBy	in	[[T]]	[[T]]
	cmp	[T]×[T]→bool	
GroupBy	in	[[T]]	[[S]]
	grp	[T]→int	
	agg	[[T]]→[S]	
Count	in	[[T]]	[[T,int]]
Join / LeftJoin	in	[[T]]	[[T,S,R]]
	left	[T]→[[T,S]]	
	right	[T]→[[T,R]]	
	pred	[T,S]×[T,R]→bool	
Concat	in	[[T]]	[[T,S]]
	left	[T]→[[T,S]]	
	right	[T]→[[T,S]]	
Union / Intersect	in	[[T]]	[[T,S]]
	left	[T]→[[T,S]]	
	right	[T]→[[T,S]]	
	cmp	[T,S]×[T,S]→bool	

Table 2.2.: Overview of bulk processing operators.

The bind mechanism can be used to feed additional data into the output relation. In a chain of bulk functions, `ForEach` will therefore often play the role of an input provider.

The operators `Project` and `Select` are equivalent to projection and selection in classic relational settings. The former strips specific columns from the input relation. The latter discards tuples from the input relation, which do not fulfill the given predicate. Note, `Select` can be seen as a specialized version of `ForEach`. Examples for both are given in Figure 2.4.

in	
1	"c"
2	"b"
3	"a"

ForEach(in,bind)		
1	"c"	2
1	"c"	3
2	"b"	3
2	"b"	4
3	"a"	4
3	"a"	5

with bind := [x,y] → [[x+1],[x+2]]

Figure 2.3.: Exemplified output of ForEach operator.

in	
1	"c"
2	"b"
3	"a"

Project(in,proj)
1
2
3

Select(in,pred)	
2	"b"
3	"a"

with proj := [x,y] → [x]
pred := [x,y] → x > 1

Figure 2.4.: Exemplified output of Project and Select.

2.3.2. Sorting, Grouping, and Aggregation

The second group of operators hosts sort and aggregation functions. In contrast to the first group, they operate on a per-relation basis and evaluate relationships and correlations between individual rows. Without surprise, the sort operator OrderBy is identical to its relational counterpart. It reorders the tuples in the input relation according to the ordering scheme determined by the comparison function cmp. Without loss of generality, we assume that cmp evaluates a less-or-equal relationship between tuples. An example of OrderBy is given in Figure 2.5.

In a language like SQL, which initially bases on the order-agnostic relational model, a sort operator can basically employ any sort algorithm. In array and map values, however, order is a fundamental aspect. Hence, OrderBy must implement a stable sort to guarantee deterministic results. Similarly to ForEach, this requirement can be relaxed if a query tolerates non-stable sorting.

in	
1	"c"
2	"b"
3	"a"

OrderBy(in,cmp)	
3	"a"
2	"b"
1	"c"

with cmp := $[x_1,y_1]\times[x_2,y_2] \rightarrow y_1 \leq y_2$

Figure 2.5.: Example of `OrderBy`.

The `GroupBy` operator divides the input relation into separate partitions and aggregates each of the latter to a single output tuple. The partitioning scheme is defined by the given grouping function[2]. The function `agg` aggregates the tuples in each non-empty partition. To obtain deterministic results, we require again that the relative order of input tuples is preserved within each partition. The ordering of the aggregated tuples in the output relation, however, is implementation-dependent to permit efficient grouping algorithms.

`GroupBy` is a generalized variant of standard grouping operators for plain relational data. As in SQL, the tuples within each partition must only be aggregated column-wise. However, the aggregation function itself is not limited to primitive aggregates like `count`, `sum`, `min`, `max`, etc. Any aggregation function yielding a primitive or non-primitive aggregate value is allowed. A very simple aggregation function, for example, could combine all column values into a single array.

Figure 2.6 demonstrates the output of `GroupBy` for a simple summation function `agg1` and a concatenation function `agg2`, which are both defined as specializations of the higher-order aggregation function `foldl` [Hut99]. Note that both aggregation functions obey the mentioned standard behavior of a grouping operation. The columns of a partition are aggregated separately and the column that served as grouping key is reduced to a single representative value.

The operator `Count` enumerates a relation and attaches to each tuple its position in the input relation. The numbering is dense and starts with `1`. An example is shown in Figure 2.7.

[2]For illustration purposes, Table 2.2 assumes numbered partitions and a grouping function `[T]→int`, which assigns a tuple to a specific partition.

in	
1	1
2	4
1	3
3	2
2	3

GroupBy(in,grp,agg1)	
1	4
2	7
3	2

GroupBy(in,grp,agg2)	
1	[1,3]
2	[4,3]
3	[3]

with `grp` := [x,y] → x
`agg1` := [[x,y]] → `foldl(add-right,`[∅,0], [x,y])]
`agg2` := [[x,y]] → `foldl(concat-right,`[∅,[]],[x,y])]
`add-right` := $[x_1,y_1]\times[x_2,y_2] \rightarrow [x_2,y_1+y_2]$
`concat-right` := $[x_1,[y]]\times[x_2,y_2] \rightarrow [x_2,[y,y_2]]$

Figure 2.6.: Examples of aggregation types in `GroupBy`.

in
"a"
"b"
"c"

Count(in)	
"a"	1
"b"	2
"c"	3

Figure 2.7.: Example of `Count`.

In some situations, the compiler presented in Chapter 4 will use `Count` just to assign each tuple a unique label, which can be used later on as sort or grouping key. In this case, an implementation may optimize the labeling process in favor of parallelism and permit "holes".

2.3.3. Joining and Combining

The last group of bulk operators provides means to combine two relations in various ways. In contrast to the aforementioned operators, however, neither operator of this group is essential. Each one can be replaced through combinations of other operators to yield the same result. However, because efficient algorithms are usually indispensable for queries which combine data, the operators in this group are included to enable the use of optimized algorithms and simplify the compilation process.

The operators `Join` and `LeftJoin` implement standard join functionality[3], but in a more general manner. The table functions `left` and `right` are consecutively applied to the tuples of the input relation `in` to produce left and right join inputs, respectively. The predicate function decides which pairs of tuples from these inputs will be combined to an output tuple. In case of `LeftJoin`, every tuple from the left input will be included in the result. If a tuple does not find a join partner in the right input, it is padded with null values to ensure homogeneous tuple width in the output relation. As always, the input order is preserved in the output if not stated otherwise. Order precedence is given to the left input, which reflects a nested-loops-style evaluation strategy. Note that this does not imply that implementations cannot use other join algorithms like hash join or merge join.

Examples for both join variants are given in Figure 2.8. The input relation `in` consists of a single empty tuple. Accordingly, the input functions `left` and `right`, which return the static relations `left-in` and `right-in`, respectively, are invoked only once. These two relations are joined with a simple equality predicate over the first column values.

It is important to emphasize that the computation of multiple left and right join inputs, one for each input tuple, is a fundamental difference to the structure of a classic relational join. Effectively, `Join` and `LeftJoin` compute sequences of joins and concatenate the results. However, typical situations will often allow the query processor to take advantage of constant input functions. For example, if the right join input is functionally independent of the input relation as in Figure 2.8, a hash join implementation must load the hash table for the right input only once. In contrast, functionally dependent inputs as in Figure 2.9, require to rebuild the hash table for each input tuple. Section 5.2 addresses joins and related optimization strategies in more detail.

As their names suggest, `Concat`, `Union`, and `Intersect` cover further standard operations on relations. Like the join operators, they operate on two relations, which are step-wise computed for each tuple in the given input. As always, input order is preserved if not stated otherwise. Illustrations are given in Figure 2.10.

[3]Inner joins and left joins are particularly interesting for the material in this work. Further kinds of outer joins and self joins are left out for brevity, but not for practical or technical reasons.

in

left-in	
1	"a"
2	"b"
3	"c"

right-in	
3	"x"
1	"y"
1	"z"

Join(in,left,right,pred)			
1	"a"	1	"y"
1	"a"	1	"z"
3	"c"	3	"x"

LeftJoin(in,left,right,pred)			
1	"a"	1	"y"
1	"a"	1	"z"
2	"b"		
3	"c"	3	"x"

with $\texttt{pred} := [x_1,y_1]\times[x_2,y_2] \rightarrow x_1{=}x_2$
$\texttt{left} := [x] \rightarrow \texttt{left-in}$
$\texttt{right} := [x] \rightarrow \texttt{right-in}$

Figure 2.8.: Exemplified output of `Join` and `LeftJoin`.

2.3.4. Composition of Operators

The consistent use of relational structures in operators facilitates their composition to accommodate typical query patterns. A composition can be established by chaining operators via output/input parameters, e.g., to form sequences of scan, filter, and transformation steps, or in form of nestings inside joins, unions, etc.

The summation function from Figure 2.2 can be rewritten with operators as shown in Figure 2.11. In contrast to the recursive solution, a compiler can easily deduce that the data flows from inside out within this composition. Starting with the empty tuple, the auxiliary function `transpose` converts the input array to a single-column relation which is also the result of the respective `ForEach` function. `Select` filters this result by evaluating the predicate `first-greater-than-3` for each tuple. `GroupBy` assigns the remaining tuples to the partition `0` by using the constant partitioning function `zero`, and finally computes the sum of all values in this partition.

The equivalence to a relational query plan is obvious. By taking recourse on relational query optimization, we can therefore employ similar rewriting techniques for operator compositions. `Select` filters should

in
1
2

Join(in,left,right,pred)		
1	2	2
2	3	3

left([1])	
1	2
1	3
1	4

left([2])	
2	3
2	4
2	5

right([1])	
1	2
1	0

right([2])	
2	3
2	1

with `pred` := $[x_1,y_1]\times[x_2,y_2] \rightarrow y_1{=}y_2$
`left` := [x] → [[x,x+1],[x,x+2],[x,x+3]]
`right` := [x] → [[x,x+1],[x,x-1]]

Figure 2.9.: `Join` with functionally-dependent input functions.

be moved upwards in a chain to reduce the size of intermediate results, a filtered Cartesian product should be replaced by a join, etc. However, to perform such restructuring, we must first complete the picture and expose hidden dependencies in compositions of higher-order bulk functions.

So far, we tacitly assumed that composition knowledge is encapsulated in supplementary functions like `first-greater-than-3`. The predicate function expects an input relation with rows having at least one column and compares the value at position `0` of a given input tuple against the constant `3`. Similarly, the function `sum-first` accumulates only the values at position `0`. Any reordering of operators may easily break such dependencies.

To get rid of auxiliary functions and hard-coded tuple positions, we introduce a naming scheme for columns as it is natural in relational settings. If every column in the output relation of a function is labeled so that the consuming operator can access individual columns by referring to the respective label, the compiler can easily trace dependencies during optimization.

In Figure 2.12, we labeled the single output columns of `ForEach` and `Select` with `$a` and `$a'` as indicated by the suffixes `[$a]` and `[$a']`, respectively. The two unary helper functions `sum-first` and `first-greater-than-3` were replaced with parameterized versions.

in

left-in	
1	"a"
2	"b"
3	"c"

right-in	
3	"x"
2	"b"
1	"a"

Concat(in,left,right)	
1	"a"
2	"b"
3	"c"
3	"x"
2	"b"
1	"a"

Union(in,left,right,cmp)	
1	"a"
2	"b"
3	"c"
3	"x"

Intersect(in,left,right,cmp)	
1	"a"
2	"b"

with $\texttt{cmp} := [x_1,y_1]\times[x_2,y_2] \rightarrow x_1{=}x_2 \wedge y_1{=}y_2$
$\texttt{left} := [x] \rightarrow \texttt{left-in}$
$\texttt{right} := [x] \rightarrow \texttt{right-in}$

Figure 2.10.: Exemplified output of `Concat`, `Union`, and `Intersect`.

The summation function is replaced by `sum($a)` and the predicate is replaced by the function call `greater-than($a,3)`. The output column of `GroupBy` is labeled with `$a'` instead of `$a` to indicate that values in this column are not the original values produced by `ForEach`, but aggregates thereof.

Effectively, a column label is a variable binding for a specific column position and a reference to it translates into a respective access operation on the current input tuple. As will be shown in Section 4.2.1, the abstraction mechanisms of the lambda calculus will provide the right tool for defining symbolic column names and modeling the dependencies between individual operators. However, recall that we introduced dedicated operators only as internal representation of common query constructs and data flows. At the syntax level, we aim for dedicated language constructs which allow to express bulk operations in a concise and natural fashion. In our front-end language XQuery, this part is greatly accomplished by FLWOR expressions.

```
declare function sum-greater-3-bulk($val) {
  GroupBy(
     Select(
       ForEach(
          [],
          transpose($val)
       ),
       first-greater-than-3
     ),
   zero,
   sum-first)
}
```

Figure 2.11.: Bulk function to sum up all values greater than 3.

```
declare function sum-greater-3-bulk($val) {
  GroupBy[$a'](
     Select[$a](
       ForEach[$a](
          [],
          transpose($val)
       ),
       greater-than($a,3)
     ),
   zero,
   sum($a))
}
```

Figure 2.12.: Use of named column positions in bulk operations.

2.4. Data Manipulation

In the elementary data model, data manipulations will reflect modifications of map and array values in form of insert, update, and delete operations. An insert adds new key/value pairs and fields to maps and arrays, respectively. An update replaces mapped values and array fields. A delete shrinks composite values by removing key/value pairs or array fields. Because the effect of these basic update primitives is obvious, we can omit a detailed presentation. However, a detailed specification of update operations is necessary in the context of a concrete frontend-language or data model, e.g., the XQuery Update Facility [W3C06b].

Conceptually, data manipulation requires clarification at two different levels. It must be specified how updates are applied and propagated to persistent storage and how queries and scripts cope with mutable data.

2.4.1. Update Queries

Declarative update statements enable users to evaluate a query for specifying how data should be updated. The canonic example in SQL is the withdrawal of money from a bank account. The account is identified by the account number and its current balance is used to compute the update value:

```
UPDATE accounts
SET balance = balance - 100
WHERE account_no = 270612 .
```

Update queries like this must be processed with care because the update itself may interfere with the predicates evaluated. Otherwise, updates have ambiguous semantics or can result in uncontrollable behavior at runtime. This is independent of the language, the shape of data, and the kind of updates performed. Thus, appropriate precautions must be taken in a data programming language, too.

Snapshot Semantics

To circumvent problems with update queries, they must be carried out in two phases. In the first phase, only the query part of the update is evaluated to compute the necessary update operations. In the second phase, the latter are applied. This prevents that a declarative insert like the following ends in an infinite loop of insertions:

```
INSERT INTO entries
SELECT account_no, max(no)+1, -100, current-date()
FROM entries
WHERE account_no = 270612 .
```

The splitting of update queries in two phases is also referred to as snapshot semantics, because the updates are computed on a snapshot of the data. Clearly, this property is mandatory for specifying any meaningful update.

Transactions

Transactions are the second major concern of update processing, particularly in database systems. A transaction is a programming abstraction that shields the users of a database system from reliability and consistency issues arising from concurrency[4], failures, and distribution [GR93]. Broken down into key words, transactions guarantee atomicity, consistency, isolation, and durability for queries and updates.

Conceptually, transactions do not affect the outcome of queries and updates[5]. They solely provide a transparent execution context, which can span over multiple queries and updates. Consideration or even enforcement of transactional properties is therefore out of the scope of a language specification. Nevertheless, efficiency and correctness concerns usually require a tight integration of transaction processing and query evaluation.

2.4.2. Immutability

For the time a query is evaluated, each value read or created must be logically immutable. This guarantees that query predicates hold once they are evaluated and, thus, it prevents chaos.

In principle, immutable values must not be enforced explicitly. In the first place, it is a direct consequence of two-phased update processing or follows from the general absence of update primitives. In fact, snapshot semantics will guarantee that even those values that will be modified in the update phase appear to be immutable in the preceding query phase.

Immutability is also in many other ways a valuable property. It simplifies reasoning about data and enables the compiler to perform various kinds of optimizations like constant folding and memoization. For parallel processing, it is paramount as it rules out race conditions[6]. Concurrency and transactions (see 2.4.1) are orthogonal aspects on a concrete target platform.

[4] Note that concurrency control mechanisms must not be confused with the aforementioned snapshot semantics of update queries.

[5] Practically, optimizations like non-serializable isolation levels and savepoints relax this property in favor of faster response times and higher throughput.

[6] In fact, immutability of data is the major reason for the current renaissance of functional principles in the context of highly parallel architectures.

Note that, to benefit from immutability in scripts, the first phase of update processing must be extended over all subqueries, i.e., all updates specified therein are deferred until the end of the script. Obviously, this is a rather rigid restriction, but it is necessary to avoid arbitrary complex analysis of which intermediate results are affected.

2.5. Runtime Aspects

After having described the general structures of a data programming language, it must be clarified how queries and scripts are actually evaluated. In the following, we focus on aspects, which may influence the outcome of a query. Concrete evaluation techniques and technicalities are subject to other chapters.

2.5.1. Evaluation Model

In principle, every part of a query, from simple arithmetic to function calls, is treated as an expression, which yields a result value. An expression may consist of several subexpressions and even individual statements in a script are subexpression in a sequence of expressions. Accordingly, queries and scripts form trees of expressions, which are evaluated recursively.

Individual expressions may be evaluated in an eager or a lazy fashion. With eager evaluation, the result value is computed and materialized instantly, whereas with lazy evaluation, a placeholder value mimics the materialized result and computes it incrementally on demand [LS88, Lie87]. The query processor is free to choose the best evaluation strategy for an expression with respect to CPU consumption and memory overhead.

The query processor is in principle also free to evaluate the expression tree in arbitrary orders or even in parallel – as long as the respective expressions are independent of the surrounding context. The context of an expression is an implicit environment of variable bindings, which influences its outcome. Variable bindings refer to user-specified input parameters, but also reflect results of preceding statements, function arguments, and, primarily, the lambda abstractions in operator compositions as mentioned in Section 2.3. As will be shown, modeling of this

context is key to scalable query processing. Thus, large parts of this thesis will primarily focus on how to organize the dynamic environment and how to evaluate context-dependent expressions.

Inside expression trees, sections with basic operations like data accesses, function calls, and general computations can be arbitrarily interleaved with bulk processing operators. Clearly, the high data volumes intended to be processed by the latter demand special treatment. To leverage relational bulk algorithms, an operator composition is treated like a pipeline of relational operators. Within the whole query, however, the composition acts as a single expression.

The localization of bulk processing logic offers various possibilities for optimized processing. At runtime, the representation as higher-order functions can be completely dissolved, e.g., to pipeline intermediate tuples between operators. Depending on the shape of a query, the input data, and available system resources, a composition can be processed as a pull-based operator tree or a push-based data-flow graph, sequentially or in parallel, with or without materialization of intermediate results.

2.5.2. Side Effects

Functions that directly or indirectly interact with the system or external resources may cause side effects or deliver indeterministic results. This is critical for the same reasons as mutable values are. Indeterminism can result in manifold problems including silent errors and crashes. Unfortunately, it is not possible to prevent side effects and indeterminism in practice at all. For example, if a query or a script reads input from a file or another uncontrolled resource, then there is no guarantee that this input is stable and can be read repeatedly. Therefore, functions causing or relying on side effects should be avoided whenever possible. In consequence, the direct use of algorithms that rely on mutable data structures is restricted or at least discouraged.

Functional programming languages face exactly the same problems and developed strategies to cope with side effects. The most obvious solution is the specification of an evaluation order for certain constructs, which allows the developer to reason about the occurrence of side effects. Pure functional languages do not permit side effects by design and help themselves with monadic type constructs, which encapsulate indeterminism in a proper functional value (see Section 4.2.1). Internally, monadic values enforce a specific ordering of side-effecting actions.

The application of strict execution policies deprive a query language of many optimizations including the most important ones: operator reordering, lazy evaluation and parallelism. As a compromise, we establish a few basic rules, from which happens-before relationships can be derived, which in turn allow to reason about program execution.

Generally, a query term may be evaluated in any order, which seems appropriate to the query processor. The summation `a+b`, for example, may evaluate operand `a` before `b`, but also `b` before `a`. This rule applies to all kinds of terms except the ones described in the following.

Conditional Branching

Language constructs which implement conditional branching like an `if-then-else` must always evaluate in the same sequential order. In a term `if (c) then a else b`, the condition `c` is evaluated first and, depending on the outcome of `c`, either `a` or `b` is evaluated next. It is neither allowed to evaluate the branches before the condition, nor it is allowed to evaluate both of them. The same rationale applies to all supported variations of conditional execution like `switch-case`.

Conjunction and Disjunction

Boolean expressions consisting of conjunctive and disjunctive clauses are evaluated sequentially, but the evaluation stops as soon as the final truth value is fixed. The conjunction `a and b` evaluates `a` first and `b` only if `a` yields `true`. The disjunction `a or b` evaluates `a` first, too, but `b` only if `a` yields `false`.

Statements

The individual subqueries in a query script are evaluated in their order of appearance, i.e., a statement $v_i := q_i$ is evaluated before the following statement $v_{i+1} := q_{i+1}$. Effectively, this allows to serialize most practically relevant cases where side-effect-afflicted I/O occurs, e.g., when writing and reading intermediate results to and from disk, respectively.

Lazy Evaluation

Despite the above evaluation rules, there are still some aspects left open, which influence whether and how often a side-effecting action will take place or not. Namely, it must be clarified what it actually means if one expression is said to be evaluated before another.

The evaluation process embodies two main sources of uncertainty. If an expression is evaluated in a lazy fashion, it may happen that a suspected side effect has not yet occurred while another expression is evaluated that is supposed to happen logically after the side-effecting one. Related to this problem is the question of whether or not intermediate results are materialized. If the result of an expression which produces side effects is not materialized but recomputed every time it is required, e.g., in a recursive function, then the respective side effect can occur multiple times. Because there exists no general solution to this problem except insisting on eager evaluation with materialization for all intermediate results, we simply negate the naïve assumption that an expression is evaluated exactly once and in its entirety. Instead, the developer is required to wrap a critical expression in a special construct, e.g., `strict{write-to-file(...)}`, which instructs the compiler to evaluate it eagerly and only once, and to materialize the result.

Optimization

With regard to side effects, optimizations conducted by the compiler can lead to unexpected results. Reordering of operators is one of them. It causes problems if, e.g., two operators in a chain of operators are swapped, whereby the formerly first one causes side effects on which the second one depends. Other major optimizations that may silently suppress side effects are the removal of unused subexpressions, the replacement of expressions with output-equivalent alternatives, and early exits, i.e., the partial evaluation of query parts, which stops as soon as the final result can be constructed. Last but not least, parallelism is another source of indeterminism that jeopardizes all chances to reason about side effects.

A general banning of the mentioned and similar optimizations is not an option, because optimization is one of the central strengths of a declarative language. Furthermore, functions with side effects are rarely

necessary in data processing and should be avoided anyway. Therefore, the compiler is allowed to generally assume that a function does not cause side effects unless it is annotated accordingly by the developer. In the latter case, the compiler must not attempt optimizations which reorder or suppress the occurrence of side effects with respect to the strict evaluation of a non-optimized query. Passive but side-effect-dependent sections must be annotated similarly to avoid reordering, too. Although this does not determine the outcome of queries with side effects in general – the language remains declarative and does not base on a fully specified execution model – this allows the developer at least to reason about the query with respect to the canonical evaluation strategy of a specific target platform, which, for example, evaluates operators in a pipelined nested-loops style.

In a certain sense, annotating a function as a source of indeterminism taints a – potentially large – part of the query, which cannot profit from advanced optimization or parallelism. Thus, a front-end language may alternatively offer a special language construct, which prohibits optimization of critical regions, e.g., `restricted{...}`.

2.5.3. Error Handling

Compilation and evaluation of a query may lead to various kinds of static and dynamic errors. Most of them are related to typing issues, e.g., when function is called with illegal argument values. Without delving into details, we assume the compiler to perform respective type checks at runtime and statically, if possible. Other kinds of runtime errors concern a broad spectrum reaching from unavailable resources and dynamic failures (e.g., division by zero) to constraint violations in a specific data model.

Errors are always reported immediately and lead to the abortion of a query. However, expressions may be nested `try-catch` constructs as in programming languages to enable automatic recovery from runtime errors, e.g., to work around partially invalid inputs.

If the compiler restructures and optimizes a query, it may happen that some errors will not be raised by the optimized version, because the erroneous situation does not occur anymore. For example, an illegal comparison in a search predicate might not be evaluated in an optimized query, because the data fulfilling the query predicate is accessed directly

using an additional index. Without restraining optimization efforts, we resort here to the rule set of XQuery [W3C10c], which allows non-detected errors in optimized queries as long as occurring errors are not purposely suppressed. The other way around, i.e., optimizations that yield errors which would not occur in the unoptimized query, are not allowed.

3. Extended XQuery

XQuery is a query language for XML data that was specified by the W3C almost a decade ago to consolidate various efforts in declarative XML processing [W3C98b, W3C98c, RCF00, W3C10a]. In the course of several revisions, the language grew to a versatile functional programming language, but still has its major strengts in XML bulk processing.

For this thesis, the most important aspect of XQuery is the ability to specify query patterns and control flows in a data- and platform-independent manner. In combination with the composition features of a functional language, this makes XQuery a decent demonstration candidate for the working principles and compilation aspects of a data programming language. This chapter presents the key aspects of XQuery and introduces a few extensions, which help to draw a more complete picture of our vision.

To illustrate the conceptual independence of data processing concepts and physical data representation, this work augments XQuery with data structures and operation primitives for the competing serialization format JSON [Cro06]. With a small syntax extension, we also uncover already built-in scripting capabilities.

3.1. Data Model

The XQuery and XPath Data Model (XDM) [W3C11b] is built around the family of XML standards, e.g., [W3C06a, W3C04a, W3C04b]. As a result, XDM is a relatively complex basement for a query language but, fortunately, most subtleties of XML can be confined to node construction, schema validation, and serialization. For this thesis, it is sufficient to concentrate on the main aspects of XDM and show the correspondences to the basic data model described in the previous chapter.

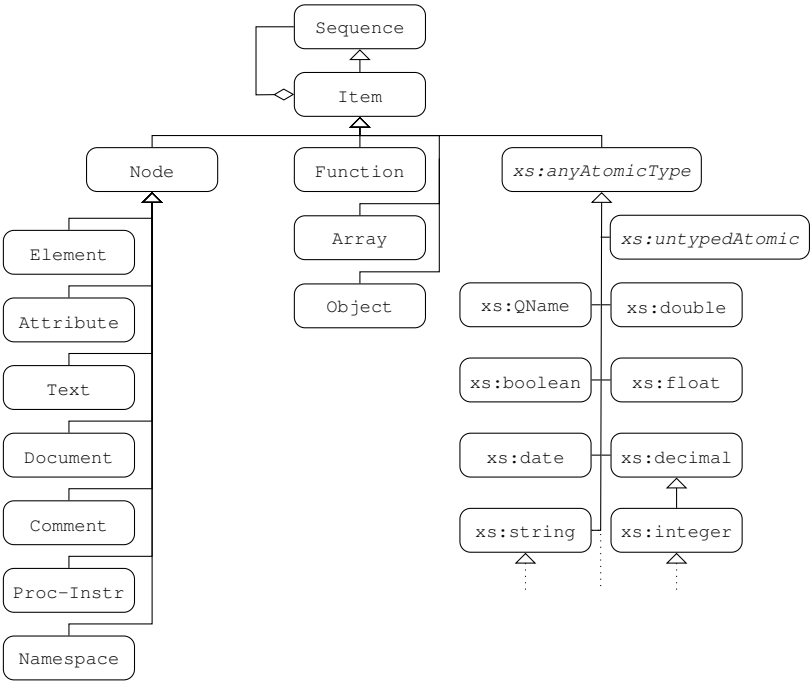


Figure 3.1.: Items and sequences in the extended XQuery data model.

3.1.1. Items and Sequences

The central concept in XDM is the notion of items and sequences as illustrated in Figure 3.1. Sequences are ordered lists of zero or more items. Conceptually, they are array composites with the speciality that nestings of sequences are always flattened, i.e., it is not possible to build hierarchical structures of sequences. Single items are treated as equivalent to a sequence of one item. Because the empty sequence `()` is also used to represent missing data, it serves as null value in XDM.

Originally, items are distinguished into three classes. Atomic items are XDM's notion of primitive values like strings and numerics. Node items represent the nodes of an XML document tree. Function items are objects representing XQuery functions, which can be evaluated for

a set of arguments. In our extended version, array items and object items reflect the respective data structures from the JSON data model.

XML

Node items are the standard abstraction for representing structured and semi-structured data in XML. Nodes appear in seven different kinds, e.g., as elements, attributes, or text, and have various kind-specific properties like a name, a type, a typed value, a text value, a set of children, and a parent. Furthermore, nodes have an identity, which also has implications for XML processing. Altogether, these properties are defined in the XML Infoset specification [W3C04a], which specifies how XML nodes can be composed to model data as XML documents or fragments.

Conceptually, a node item reduces to a complex composite of array and map values, but choosing a concrete composition form is not necessary for our discussion. Neither necessary for this thesis and, thus, generally omitted are advanced XML specifics like serialization, whitespace handling, and XML namespaces. For details we refer to the respective standard documents [W3C11c, W3C09].

Objects and Arrays

Object items and array items are extended variants of the respective structures in JSON, which, in turn, represent incarnations of the two fundamental composition types map and array of Section 2.1.1. For a seamless integration with common idioms of XQuery, array items are indexed with atomic integer values and maps are keyed by QNames, respectively. QNames are the standard atomic type for object names in XDM. Furthermore, nodes and sequences are legal values, which de facto extends the basic JSON model. Null and the JSON value types `boolean`, `string` and `numeric` are mapped to the empty sequence and atomic value types, respectively.

A basic constructor syntax for static JSON items was already exemplified in Table 2.1. Examples for dynamically computed objects and arrays are given in Table 3.1.

Value Type	Examples

Array	`[ 1=1, substring("banana", 3, 5), () , <a/>]`
	evaluates to
	`[ 2, "nana", null, <a/> ]`

	`[ (1 to 5), [ "x", true() ] ]`
	evaluates to
	`[ (1,2,3,4,5), [ "x", xs:boolean(1) ] ]`

	`[ =(1 to 5), 19.54, false() ]`
	evaluates to
	`[ 1, 2, 3, 4, 5, 19.54, xs:boolean(0) ]`

Object	`{ "a": 1, 'b' : 2, c : 3 }`
	evaluates to
	`{ "a" : 1, "b" : 2, "c" : 3 }`

	`{ a : concat('f','oo') , b : [1,2][[1]] }`
	evaluates to
	`{ "a" : "foo", "b" : [2]}`

	`let $r := { x:1, y:2 } return { $r, z:3 }`
	evaluates to
	`{ "x" : 1, "y" : 2, "z" : 3}`

	`{ x : 1, y : 2, z : 3 }{z,y}`
	evaluates to
	`{ "z" : 3, "y" : 2 }`

Table 3.1.: Dynamic construction of array and object items.

3.1.2. Properties and Accessors

Between items and sequences exist various kinds of relationships. The name of an XML element node, for example, is an atomic value. Both can be part of several sequences and the element can also be parent, child, and sibling of other nodes in an XML fragment. XDM defines many of these relationships explicitly in form of properties.

Operations in XDM are simple accessors for sequences and node item properties. Groups of in total 17 accessors cover the logical aspects of a node. The central ones are accessors for structural properties (`attributes`, `children`, `parent`) and accessors for value and type properties (`node-name`, `string-value`, `typed-value`, `type-name`, `node-kind`). The other accessors cover miscellaneous XML-related properties (e.g, `document-uri`, `is-id`, etc.).

In XPath/XQuery, accessors are either exposed as functions (e.g., `fn:name()`) or as part of higher-level axis step expressions. The latter evaluate whole combinations of item properties. The axis step expression `child::revenue`, for example, fetches the child list of the context node through its `children` accessor and then uses the `node-kind` and `node-name` accessors to filter this list for element nodes having the name `revenue`. Of course, query processors must not strictly follow this procedure and may implement more efficient algorithms for evaluating paths in XML trees.

JSON objects and arrays provide each solely a single accessor, which they inherited from the underlying composition type. Maps expose the property `entries` for enumerating the key/value pairs. Arrays expose their field values via the property `fields`. Note that this is already sufficient to implement the access operations presented in Section 2.2.1, but, as with XML nodes, the query processor may realize most of them with more efficient strategies like associative lookups.

3.1.3. Types

A central design goal for XDM was to support decent XML processing for both properly typed, schema-conform data as well as for schema-less or unvalidated XML. The resulting type hierarchy therefore needed to embody the original type system from XML Schema. In addition to that, XDM was required to model data, i.e., nodes, atomics, and recently also functions, in a way that allows to define a concise and consistent query language. As a result, the type system of XDM is characterized by a dualism created by merging two distinct type hierarchies, the item hierarchy and the XML Schema hierarchy, into one.

XML Schema

The XML Schema part of the hierarchy is rooted by `xs:anyType`. It is shown in Figure 3.2. Notably, the branch `xs:anyAtomicType` overlaps with the item type hierarchy.

The dualism requires to use the merged type hierarchy with the appropriate perspective. The item perspective comes close to the type systems of standard programming languages and is the default in XQuery. The XML Schema perspective comes into play when XML needs to be

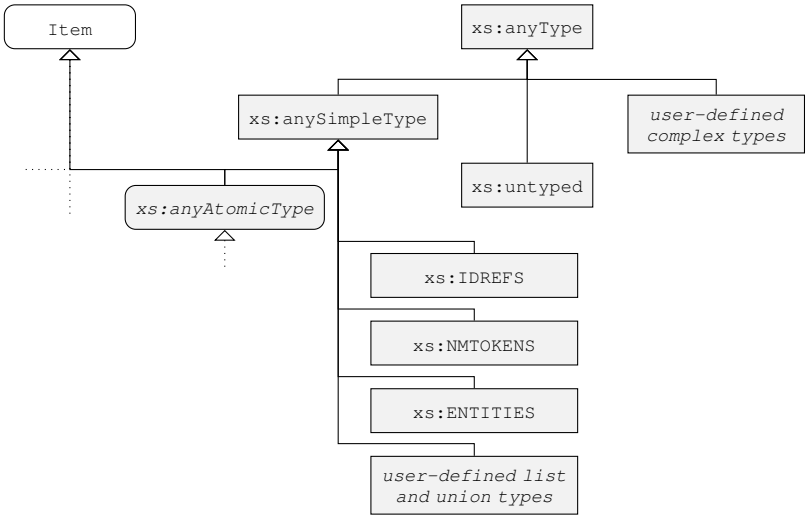


Figure 3.2.: XML Schema part of the dual XDM type system.

validated against a schema definition, which may be the case before, during, or after the actual query. After successful validation, XML nodes are labeled with a corresponding schema type annotation, which effectively determines their (sub-)type in the item perspective. The special types `xs:untyped` and `xs:untypedAtomic` serve as default types for unvalidated XML nodes and their content, respectively. As a comfort feature, values of the latter type are subject to implicit conversion rules, which simplify the work with data originating from textual XML representations.

Like in the item type perspective, every major group in XML Schema has extension points for user-defined types. The shared branch of atomic types is interpreted in both perspectives as hierarchy of primitive values, which supports sub-typing by domain restriction.

Sequence Types

On top of the item perspective, XQuery uses the notion of sequence types for referring to the type of results, function arguments, etc.

A sequence type is a pair of an item type and an optional occurrence indicator `?`, `*`, or `+` meaning zero-or-one, zero-or-many, and one-or-many respectively. The special type `empty-sequence()` represents the empty sequence.

Sequence type matching is a mechanism for ensuring the type of a sequence. A function `foo(xs:date,xs:int*)→ xs:string+`, for example, must only be invoked for two arguments, with the first argument being an atomic value of type `xs:date` or any subtype of `xs:date` and the second argument being a sequence of atomic values of type `xs:int` or any subtype of `xs:int`. Furthermore, the function must only return a non-empty sequence of `xs:string` values. Respective type validation must be performed both statically and dynamically. Typing errors raise exceptions in the runtime system.

3.1.4. Additional Concepts

To simplify common situations, XQuery defines three additional concepts for items and sequences. For the sake of completeness, we name them here, but refer to the language specification for details:

Document Order A total, stable ordering of all node items in a query. Within XML fragments, document order is defined as the order in which the nodes appear in the serialized representation.

Atomization A coercion mechanism for converting arbitrary sequences into sequences of atomic values. It is primarily applied to the arguments of arithmetics, comparisons, function calls, etc.

Effective Boolean Value An implicit truth value of items and sequences for convenient use in logical and conditional expressions. It is based on common programming language idioms, e.g., a non-empty string has the effective Boolean value `true` and the empty sequence has, as representation of the null value, the effective Boolean value `false`.

3.2. Expressions

A query is a tree of expressions that is evaluated to a sequence of items. The spectrum of expression types reaches from basic ones like literals, arithmetics and function calls, to XQuery-specific FLWOR expressions

and expressions derived thereof like filter expressions and path expressions. A full overview of all built-in expression types of XQuery 3.0 is given in Table 3.2.

As mentioned in Chapter 2, equivalents to the basic expression types can be found in almost any programming language. FLWOR-based expressions, however, are the language-specific equivalent to bulk operator compositions and serve fundamental data processing tasks. As such, they play a central role for this thesis and will be introduced briefly in the following.

3.2.1. FLWOR Expressions

FLWOR expressions are the language's backbone. The name stems from their general structure – a sequence of clauses, i.e., composable language primitives for iterating, filtering, reordering and grouping of data. The standard clauses are `for`, `let`, `where`, `order by`, and `return`. In the current working draft for version 3.0 of XQuery [W3C10c], they are accompanied by the clauses `count`, `window`, and `group by`[1].

Inside a FLWOR expression, the individual clauses operate on a logical tuple stream. A tuple is a set of named variables, which are bound to XDM sequence values by individual clauses and visible to all following clauses within the same FLWOR expression. Each sequence of clauses is terminated by a single `return` clause, which reduces the tuple stream to a single XDM sequence – the result of the FLWOR expression. For illustration, consider the sample query and the corresponding expression tree shown in Figure 3.3.

The query consists of two nested FLWOR expressions with `for`-loop clauses. The outer loop binds the values `1`, `2`, and `3` successively to a variable `$a` and evaluates for each binding the nested FLWOR expression, which itself loops with a run variable `$b` over the sequence `(2,3,4)` to compute {`$a`,`$b`} pairs. The `let` clause binds the sum `$a+$b` for each pair that fulfills the `where` predicate `$a=$b` to a variable `$c`. Finally, the `return` clause produces the output by concatenating all values bound to `$c`. Accordingly, we obtain the result `(4,6)`.

The correspondence between FLWOR clauses and bulk operators is obvious. The `for`, `let`, and `where` clauses match to `ForEach` and

[1] All XQuery examples in this thesis base on the syntax and language features of the W3C XQuery 3.0 working draft from December 13, 2011.

Expression Type	Examples
Literal	`11  1.34e10  "a string"`
Variable Reference	`$var`
Parenthesized Expression	`(...)`
Context Item Expression	`.`
Static Function Call	`fn:substring("brothers", 3, 6)`
Named Function Reference	`fn:substring#3`
Inline Function Expression	`function($a) as xs:integer { $a*$a }`
Filter Expression	`(19,81,11,11)[.>11]`
Dynamic Function Call	`$fun("foo","bar")`

Path Expression	`$n//person/address:address_t/city[2]`
	`$n/parent::*/attribute::id`
	`$number!sqrt(.)`

Sequence Expression	`3, foo("bar"), $x+1`
Range Expression	`1 to 500`
Node Sequence Combination	`(<a/>,<b/>) union (<c/>)` (and `intersect`, `except`)
Arithmetic	`3 + 4.8` (and `-`, `*`, `div`, `idiv`, `mod`)
String Concatenation	`"foot" || "ball"`
Value Comparison	`3 eq 1` (and `ne`, `lt`, `le`, `gt`, `ge`)
General Comparison	`(3,1,5)=(9,5)` (and `=`, `!=`, `<`, `<=`, `>`, `>=`)
Node Comparison	`<e>1</e> is <e>1</e>` (and `<<`, `>>`)
And/Or Expression	`"yes" and "no"  false() or true()`

Node Constructor	`<elem attr="val"><child/></elem>`
	`<node>{"con" || "tent"}</node>`
	`attribute "id" { 340 }`
	`text { "some value" }`

FLWOR Expression	`for $p in doc("parts.xml")`
	`where $p/price > 5000`
	`return $p/@id`

(Un-)ordered Expression	`unordered {...}  ordered {...}`
Conditional Expression	`if ($i>10) then "y" else "n"`

Switch Expression	`switch ($i)`
	`case "I" return 1 case "II" ...`
	`default return -1`

Quantified Expression	`some $x in $y satisfies check($x)` `every $x in $y satisfies check($x)`
Try/Catch Expression	`try { calc() } catch * { "oops" }`
Instance Of Expression	`5 instance of xs:integer`

Typeswitch Expression	`typeswitch ($node)`
	`case $n as element(*,address_t)`
	`return $n/city() case ...`
	`default return "unsupported type"`

Cast Expression	`"1.3" cast as xs:double`
Castable Expression	`"1.3" castable as xs:double`
Constructor Function	`xs:boolean("true")`
Treat Expression	`treat $nums as xs:integer+`
Validate Expression	`validate strict { doc('data.xml') }`

Table 3.2.: Overview of expression types in XQuery 3.0.

```
for $a in (1,2,3)
return for $b in (2,3,4)
       where $a=$b
       let $c := $a+$b
       return $c
```

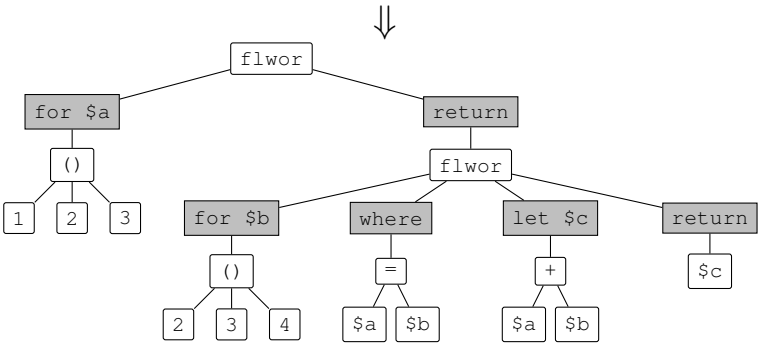


Figure 3.3.: Expression tree for nested FLWOR expression.

`Select` operators, respectively. The `return` clause reflects a combination of `Project` and `GroupBy`. Variable bindings directly translate to tuple positions in the intermediate relations.

3.2.2. Filter Expressions

Filter expressions allow to filter an item sequence by a given positional or general predicate. The filter expression `("a","b","a")[1]`, for example, selects the first item `"a"` from the input sequence, whereas the predicate in `("a","b","a")[.="a"]` matches both `"a"` string items yielding the sequence `("a","a")`. Intrinsically, filter expressions are syntactic sugar for plain FLWOR expressions as shown in Figure 3.4.

$e_1[e_2]$ $\Leftrightarrow$

```
let $tmp := e1
let $fs:last := fn:count($tmp)
for $fs:dot at $fs:pos in $tmp
let $p := e2
return if (fs:is-numeric($p)) then
          if ($p eq $fs:pos) then
            fs:dot
          else
            ()
       else
          if ($p) then
            fs:dot
          else
            ()
```

Figure 3.4.: Equivalence of filter expressions and FLWOR expressions.

The initial expression e_1 is evaluated once and its result is bound to a temporary variable `$tmp`. The size of the result is bound to an auxiliary variable `$fs:last`. The second expression e_2 is the filter predicate. It is evaluated for each item in `$tmp` by iterating over the input sequence with a run variable `$fs:dot` and a position variable `$fs:pos`. The variables `$fs:dot`, `$fs:last`, and `$fs:pos` define the implicit evaluation context for evaluating e_2 (see also Section 3.3).

The `return` clause consists of a two-staged conditional expression which simply returns the item currently bound to `$fs:dot` if the predicate is fulfilled and emits the empty sequence `()` otherwise.

3.2.3. Path Expressions

Path expressions probably attracted the most attention of all expression types, because they are the building block for navigating XML trees. Syntactically, they have the form e_1/e_2 with e_1 and e_2 being individual subexpressions. Their special role in XML processing is owned to the fact that the second expression evaluates in the context of first, i.e., e_2's result is dependent on the outcome of e_1. This is achieved by interpreting path expressions as composites of a FLWOR expression and the two step expressions e_1 and e_2 as shown in Figure 3.5.

$e_1/e_2 \quad \Leftrightarrow$

```
fs:ddo (
 let $tmp := e1
 let $fs:last := fn:count($tmp)
 for $fs:dot at $fs:pos in $tmp
 return e2
)
```

Figure 3.5.: Equivalence of path expressions and FLWOR expressions.

The evaluation of path expressions is similar to the aforementioned filter expressions, except that the sequences obtained by evaluating e_2 are part of the result. The FLWOR is wrapped in a call of the auxiliary function `fs:ddo()`, which enforces some XML-specific ordering and typing properties like distinct-document-order on the result sequence.

XQuery 3.0 introduces the map operator `!` as a generalized variant of the traditional path operator `/`. A path expression $e_1 \,!\, e_2$ is then evaluated as shown before, except that the map operator does not imply the use of the `fs:ddo()` function.

In practice, most step expressions are so-called axis steps, which allow to use path expressions for pattern-based navigation in complex XML structures. As already shown in Section 3.1, axis steps basically consist of an axis specifier and a node test, which evaluate the XDM accessor functions for a given node. The context node is the implicitly bound variable `$fs:dot`.

Optionally, an axis step may also have a trailing list of predicates enclosed in '`[]`' for further filtering the list of qualified result nodes. The predicates in axis steps are treated similarly to filter expressions except for subtle differences in the treatment of positional predicates when the axis specifier navigates in reverse document order.

3.2.4. Quantified Expressions

Quantified expressions yield a truth value for a predicate over a collection of values. A `some` quantified expression iterates with a variable over a sequence and yields true, if the predicate holds for any value in the sequence. In an `every` quantified expression, the predicate must hold for a values in the sequence.

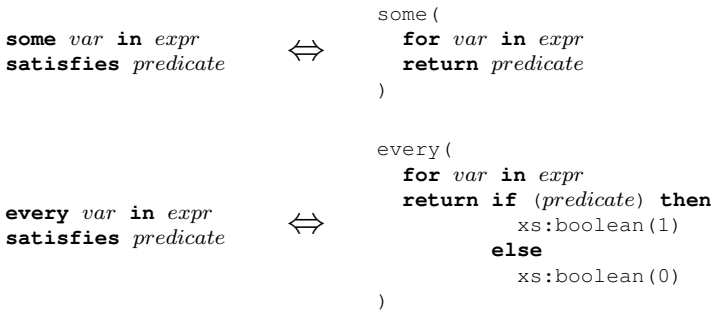


Figure 3.6.: Interpretation of quantified expressions as FLWORs.

The standard considers quantified expressions as independent expression types [W3C10b]. However, they can also be emulated by simple FLWOR expressions as shown in Figure 3.6. A single `for` loop iterates over the input sequence and the `return` expression evaluates the respective predicate. The actual quantification is finally performed by passing the result of the FLWOR as argument to a helper function.

For the `every` quantifier, one must take care that the predicate does not evaluate to the empty sequence, because it would be discarded from the result even though it should cause the whole quantification to evaluate to false. Accordingly, the respective predicate in Figure 3.6 is wrapped in an `if-then-else` as a guard against empty sequences.

A query processor is allowed to evaluate quantified expressions in arbitrary orders and short-circuit the evaluation to stop as soon as the truth value is determined. In the FLWOR representation, this obvious optimization is not present anymore, but for the sake of simplicity, we assume that quantified expressions will always be converted to corresponding FLWORs. With regard to efficiency, we can also safely assume that the compiler understands the semantics of the employed helper functions and introduces appropriate short-circuiting techniques.

3.3. Evaluation Context

Some expressions and FLWOR clauses depend on variables and other external factors, which affect their result or their behavior. This external environment is divided into a static and a dynamic part. The plain variable reference expression `$a`, for example, evaluates to the value currently bound to the respective variable. The FLWOR clause `order by $a` refers to tuple position associated with the variable `$a` to get the sorting key for incoming tuples and it depends on the static declaration `order empty` indicating whether to treat empty sorting keys as greatest or least possible value in comparisons.

3.3.1. Static Context

The static context contains information for the static analysis and compilation of a query like namespace declarations, imported libraries, function signatures, and schema types. Aside this, it contains default options for runtime operations like orderings, URI resolution, etc. Most importantly, the static context contains all global and external variables and it provides default values for the implicit context variables.

The implicit context consists of the artificial context item variable `$fs:dot`, its position in a context sequence (`$fs:pos`), and the size of the latter (`$fs:last`). In principle, they are plain variables in the static context, which are overriden in desugarized filter and path expressions as shown in Section 3.2.1. However, they cannot be accessed like normal variables. They can only be referred to via the context item expression '`.`' and the special functions `fn:position()` and `fn:last()`, respectively.

3.3.2. Dynamic Context

The dynamic context is the environment in which expressions evaluate individual subexpressions. Besides platform- and runtime-specific settings like available resources and the current date, it consists of all dynamically-bound variables in the current scope. Accordingly, simple expressions like arithmetics or comparisons evaluate subexpressions in the same context they are evaluated in, whereas variable-binding expressions or FLWOR clauses modify the dynamic context beforehand.

In a FLWOR expression, the variables currently bound by `for`, `let`, `window`, and `count` clauses can be referenced in all expressions of subsequent clauses. Within the `for` loops in Figure 3.3, e.g., individual subexpressions like the comparison and the summation are evaluated multiple times, but each time within a different iteration, i.e., dynamic context. Similarly, each step in a path expression provides a context node to the next step by iteratively binding its local result to the implicit context variables `$fs:dot`, `$fs:pos`, and `$fs:last`. Non-FLWOR-based expressions which bind variables are, e.g., typeswitch expressions, quantified expressions, and try-catch expressions.

3.4. Scripting

XQuery is not as scripting language in the sense of an operating system shell or even a general-purpose scripting language. It was designed as functional-style query language and therefore lacks an explicit control flow and does not support mutable data structures. Nevertheless, the language accommodates the simpler form of data processing scripts discussed in Chapter 2 very well. Recall, they consist of a simple sequence of statements, which bind the result to a variable for further reuse.

XQuery's equivalent to a statement is a `let` binding. A `let` clause evaluates an expression, i.e., a query, binds the result to a variable, and continues with the next clause in the FLWOR. Accordingly, a script is simply a sequence of `let` bindings in the same FLWOR expression.

As convenient variant to the standard FLWOR syntax, we may enrich XQuery therefore with a special syntax, which translates a sequence of '`;`'-terminated statements into a single FLWOR expression as exemplified in Figure 3.7. A statement of the form `$v :=` e`;` is directly translated to `let $v :=` e. Statements without variable assignment like the function call `write-file(...);` are complemented with an anonymous variable. The result of a script FLWOR is simply the result of the last statement.

Note again that statements are a simple syntax extension to simplify the writing of data processing tasks already covered by regular XQuery. It is neither a subset of nor an equivalent to the XQuery Scripting Extension 1.0 [W3C10d], which defines an explicit control flow and weakens immutability guarantees by introducing explicit side effects.

```
$start := now();
$log := fn:collection("log.txt");
$report := generate-report($log);
write-file('reports.txt', $report);
for $i in search-incident($report)
  send-report('admin@acme.org', $i);
$end := now();
"Duration: " || ($end - $start) || "ms.";
```

⇔

```
let $start := now()
let $log := fn:collection("log.txt")
let $report := generate-report($log)
let $v0 := write-file('reports.txt', $report);
let $v1 := for $i in search-incident($report)
               send-report('admin@acme.org', $i);
let $end := now();
let $v2 := "Duration: " || ($end - $start) || "ms."
return $v2
```

Figure 3.7.: FLWOR-based statement syntax for scripting.

4. Hierarchical Query Plan Representation

The translation of a query, respectively a script, into an executable query plan is a complex process. The initial query string is parsed into an internal representation, which is subject to various transformations and optimizations, and finally translated into an executable form. Therefore, a suitable intermediate representation is essential for the compilation process. In the following, we present the theoretical basement of a compiler, which takes advantage of a hierarchical top-down representation for modeling nested evaluation scopes and variable bindings in a query.

4.1. Requirements

Before starting with the introduction of the top-down query representation, we motivate first a catalog of requirements that should be covered:

Conciseness The query representation should reflect the query intent in a crisp and consistent form to facilitate both query rewriting and standard compilation techniques (simplification, dead-code elimination, etc.).

Portability To support multiple platforms, query representation and rewriting rules should abstract from platform and machine-specific aspects (e.g., CPU registers, caches, memory) to the most possible extent, but at least to the final stages before the platform-specific translation into an executable query plan.

Set-orientation Support for set-oriented optimization and translation is mandatory for data processing. The query representation should reflect set-oriented aspects in a way that allows the optimizer to

perform common and data-specific optimizations. Chances for parallel query execution should be maximized and easy to exploit. However, all this should not interfere with other parts of a query and still permit a decent representation of non-bulk-oriented sections.

Abstract Data Access To maximize the portability of compiler rules, there should be clear-cut separation of query logic and data access primitives. Data access operations should be primarily treated as abstract operations to support the translation to both generic implementations (e.g., adapters) and efficient native alternatives. Furthermore, storage-specific operations and indexes should be easy to integrate, too.

Extensibility An extensible query representation enables not only support for native storage, but also for specialized operations and algorithms like XML twig joins [BKS02]. With a customizable compiler kernel, specific functionality should only require minimal adjustments of the compilation process – ideally, in form of a few rewrite rules and small utility functions.

With these design goals in mind, the first step towards an appropriate query representation requires to look at bulk operators and variable bindings from a more abstract perspective. As foretold in Section 2.3.4, we want to abstract referential dependencies to variable bindings with the help of lambda abstractions. The following section introduces the respective theoretical basement for operator compositions and rewriting rules, which finally leads to a sound and flexible query representation.

4.2. Query Representation

In contrast to most other query languages, our data programming language does not base on an operator algebra, but on a functional specification, where variable bindings and arbitrarily nested scopes play a central role. A typical bottom-up data flow graph of set-oriented operators is therefore not ideal to represent a query. It is too difficult to keep track of variable dependencies, if variables are bound at the bottom

nodes of a query plan, but referenced at higher levels in nested expressions and subqueries, which imply a local top-down view. Instead, a consistent, hierarchical top-down representation is preferable because it naturally reflects nestings and variable scopes, which notably simplifies the analysis of variable dependencies. Furthermore, a proper tree structure simplifies routines for pattern matching and rewriting rules.

4.2.1. Comprehensions

In functional programming, lambda abstractions occur in various syntactical forms. For implementing variables, i.e., dynamic bindings of values to names, many languages support constructs like the `let`-clause bindings in XQuery. A simple `let`, however, binds a variable only once in a scope and cannot reflect dynamic computation. Therefore, some languages use lambda abstractions for realizing list comprehensions, a syntactic structure for representing iterative computations on lists as functional values [ASS85].

In the functional programming language Haskell, for example, the list comprehension `[x|x<-[1..9],x<=3]` represents the list `[1,2,3]`, which is derived by iterating over the integers `1`,`2`,...,`9` and picking all values less or equal `3` to construct the result. In each iteration, the variable `x` serves as symbolic name for the current value in the integer sequence. A closer look at this comprehension reveals that it already incorporates similar functionality as the bulk operators `ForEach` and `Select`: It enumerates and filters data.

Conceptually, a list comprehension has the basic structure $[\,t \mid q\,]$ with a term t and a qualifier q [Wad90]. A qualifier is either the empty qualifier Λ, a generator of the form $x \leftarrow u$, a filter b, or a composition of qualifiers $(p; q)$. A comprehension is interpreted by applying the following rules[1]:

[1] Function applications are written in the classic functional syntax of the original paper without parentheses around the function arguments. The term $foo\,a\,b$, for example, could also be written as $foo(a, b)$ and denotes the application of a function foo on two arguments a and b.

$$
\begin{array}{lllll}
\text{a)} & [\,t \mid \Lambda\,] & = & \mathit{unit}\ t \\
\text{b)} & [\,t \mid x \leftarrow u\,] & = & \mathit{map}\ (\lambda x \rightarrow t)\ u \\
\text{c)} & [\,t \mid b\,] & = & \mathit{if}\ b\ \mathit{then}\ [\,t\,]\ \mathit{else}\ [\,] \\
\text{d)} & [\,t \mid (p; q)\,] & = & \mathit{join}\ [\,[\,t \mid q\,] \mid p\,]
\end{array}
$$

The function *unit* in rule a) is a constructor, i.e., for the empty qualifier Λ, $[\,t \mid \Lambda\,]$ creates a list of the single value t. In a generator $x \leftarrow u$, the x denotes a variable and u denotes a list value. Rule b) says accordingly that the higher-order function *map* creates a list by applying a function on each element x of the input list u defined by the term t. The third rule produces a singleton list $[t]$ and an empty list $[\,]$ for true and false Boolean-valued terms b, respectively. Finally, rule d) normalizes compositions of qualifiers recursively to the three other cases. The function *join* concatenates the resulting lists to a single list. Altogether, these rules provide a framework for composing iterative computations on list values, which use variables as glue between generators and terms.

Monads

A generalization of list comprehensions, so-called monad comprehensions, will allow us to represent all bulk operators and compositions thereof as a clear-cut concept. This great utility of comprehensions for representing queries has been recognized in the literature more than twenty years ago [TW89, Feg98, Gru99].

A monad itself is a type concept around a set of axioms, the so-called monad laws, which allows to encapsulate computation in values. Let *id* be the identity function and let $f \circ g$ denote the composition of two functions f and g, then the three functions *map*, *unit* and *join* form a monad if the following conditions hold [Wad90]:

$$
\begin{array}{lllll}
1) & \mathit{map}\ \mathit{id} & = & \mathit{id} \\
2) & \mathit{map}\ (g \circ f) & = & \mathit{map}\ g \circ \mathit{map}\ f \\
3) & \mathit{map}\ f \circ \mathit{unit} & = & \mathit{unit} \circ f \\
4) & \mathit{map}\ f \circ \mathit{join} & = & \mathit{join} \circ \mathit{map}\ (\mathit{map}\ f) \\
5) & \mathit{join}\ \mathit{unit} & = & \mathit{id} \\
6) & \mathit{join} \circ \mathit{map}\ \mathit{unit} & = & \mathit{id} \\
7) & \mathit{join} \circ \mathit{join} & = & \mathit{join} \circ \mathit{map}\ \mathit{join}
\end{array}
$$

To support filters as qualifiers, a monad needs the additional function *zero*, which creates the empty monad, e.g., the empty list monad []. It must support the following supplementary rules:

$$
\begin{array}{rlcl}
8) & map\ f \circ zero & = & zero \circ g \\
9) & join \circ zero & = & zero \\
10) & join \circ map\ zero & = & zero
\end{array}
$$

Monads play a key role in functional languages because they allow to model I/O, mutable state, nondeterminism, etc. in a functional setting. The theory behind and application of monadic constructs is a wide research field and we refer to the literature for further details.

For our discussion it is sufficient to note that monadic comprehensions are not limited to lists. They can model bags, sets, and any other type τ for which the respective functions $unit^{\tau}$, map^{τ}, $join^{\tau}$, and $zero^{\tau}$ obeying the above rules can be defined [Gru99]. In fact, it is possible cover the functionality of all bulk operators by defining monads which operate on two-dimensional arrays, i.e., lists of lists.

The derivation of monadic representations of `ForEach`, `Project` and `Select` follows straightforward from the design ideas of list comprehensions. In a respective monad $array^2$, solely the four monad functions must be adapted for operating on two-dimensional arrays instead of lists. The constructor function $unit^{array^2}$ takes a tuple, i.e., an array and produces a relation of this tuple. The map function map^{array^2} applies a transformation function to each input tuple and produces the cross product of the resulting value and the input tuple. The function $join^{array^2}$ concatenates relations. The neutral element in this monad is the empty relation [[]] created by $zero^{array^2}$. For simplicity, we assume here that tuples in a relation must not be of the same size. A tuple with less elements is considered equivalent to a tuple in which all further elements are padded with null values. In case of `Project`, the actual elimination of columns happens in the final term t.

Bulk operators that operate on a per-relation basis, i.e., `OrderBy`, `GroupBy`, and `Count`, are a bit more difficult, because they do not incorporate the simple nested-loops-like behavior of the others. They must be defined as individual monads, which also operate on two-dimensional array values.

The key idea of comprehensions, which compute values over whole relations, e.g., aggregates, bases on a suitable definition of the map^{τ} function. To give an idea of how an appropriate function definition may look like, assume a simple monad *sum* for summing up the values in a 2-dimensional array of the form $1 \times n$. It can be specified as

$$\begin{array}{lcl} zero^{sum}\ x & = & [[0]], \\ unit^{sum}\ x & = & [x], \\ map^{sum}\ f\ g & = & foldr^{sum}\ (\lambda x\ xs \rightarrow (f\ x)\ +^{sum}\ xs)\ [[0]]\ g, \\ join^{sum}\ x & = & foldr^{sum}\ (+^{sum})\ [[0]]\ x. \end{array}$$

The *zero* element in this monad is the 2-dimensional array $[[0]]$. Intuitively, it represents the sum aggregation of the values of zero 1-dimensional arrays. The constructor function $unit^{sum}$ takes a value, which is in this monad a 1-dimensional array, and wraps it to create a 1×1-dimensional array. The functions $unit^{sum}$ and $join^{sum}$ employ the higher-order function $foldr$, a standard operator for realizing structural recursion on list types [Hut99], to enumerate, transform, and combine the fields of a given array. For the binary infix operator $\oplus$, the term $foldr \oplus [0]\ [[1], [2], [3]]$, for example, evaluates to $[1] \oplus ([2] \oplus ([3] \oplus ([0])))$.

The actual aggregation is performed by the summation function[2] $+^{sum}$, which adds two 1×1-dimensional arrays:

$$[[x]]\ +^{sum}\ [[y]] \quad = \quad [[x + y]],$$

The *sum* monad is now ready to be used. For example, the sum of all row values greater than 2 in the table $[[1], [2], [3], [4]]$ can be written as the comprehension

$$[\ x \mid x \leftarrow [[1], [2], [3], [4]];\ x > [2]\ ]^{sum}.$$

The evaluation of the comprehension follows directly from the application of the general comprehension rules and the respective functions defined for this monad:

[2] Note that we use the more intuitive infix notation for the binary function $+^{sum}$.

$$
\begin{aligned}
& [\, x \mid x \leftarrow [[1],[2],[3],[4]];\ x > [2] \,]^{sum} \\
= \quad & join^{sum}\ [\, [x \mid x > [2]\,] \mid x \leftarrow [[1],[2],[3],[4]]\,]^{sum} \\
= \quad & join^{sum}\ [\, if\ x > [2]\ then\ [[x]]\ else\ [[0]] \mid x \leftarrow [[1],[2],[3],[4]]\,]^{sum} \\
= \quad & join^{sum}\ [\\
& \quad foldr\ (\lambda x\ xs \rightarrow (if\ x > [2]\ then\ [[x]]\ else\ [[0]])\ +^{sum}\ xs) \\
& \quad [[0]] \\
& \quad [[1],[2],[3],[4]] \\
&]^{sum} \\
= \quad & join^{sum}\ [\\
& \quad if\ [1] > [2]\ then\ [[1]]\ else\ [[0]] +^{sum} \\
& \quad (if\ [2] > [2]\ then\ [[3]]\ else\ [[0]] +^{sum} \\
& \quad (if\ [3] > [2]\ then\ [[3]]\ else\ [[0]] +^{sum} \\
& \quad (if\ [4] > [2]\ then\ [[4]]\ else\ [[0]] +^{sum} \\
& \quad ([[0]])))) \\
&]^{sum} \\
= \quad & join^{sum}\ [[[0]],[[0]],[[3]],[[4]],[[0]]] \\
= \quad & foldr^{sum}\ (++^{sum})\ [[0]]\ [[[0]],[[0]],[[3]],[[4]],[[0]]] \\
= \quad & [[7]]
\end{aligned}
$$

The use of the structural recursive function `foldr` allows to model any kind of monad that aggregates values. Because it is parameterized to take any suitable aggregation function, it is independent of the actual kind of values, i.e., it also nicely operates on tuples of semi-structured items and sequences. With `foldr` it is also possible to sort and enumerate a relation as required for `OrderBy` and `Count`, respectively. An alternative solution, which does not need nested comprehensions was presented in [JW07].

The monadic representation of the third group of operators (i.e., `Join`, `Concat`, `Union`) follows from their nature as combinations of the other operators. For example, a simple join can be replaced by a cross product and a selection, which corresponds to a monad with two consecutive generators and a filter.

Composing Monads

Because all bulk monads operate on relational-structured data, they can be composed like bulk operators. For example, the composite function `sum-greater-3-bulk` of Figure 2.12 can be represented by the monad expression:

$$
\begin{array}{l}
[\\
\quad [\,\$a \mid \$a \leftarrow \$val;\ \$a > 3;\ \$g' \leftarrow [[0]];\ \$g' = \$g\,]^{sum_{\$a}} \\
\qquad\qquad \mid \$g \leftarrow [[0], [1], \ldots] \\
]^{array^2}
\end{array}
$$

In the $array^2$ comprehension, the generator $\$g \leftarrow [[0], [1], \ldots]$ produces tuples of dummy partition identifiers for the grouping step and evaluates for each the nested comprehension. The generator $\$a \leftarrow \$val$ in the $sum_{\$a}$ comprehension enumerates input tuples from $\$val$ and binds them successively to the variable $\$a$. Then, the filter $\$a > 3$ eliminates all tuples where $\$a$ is less or equal three. Finally, the generator $\$g' \leftarrow [[0]]$ attaches 0 as grouping column $\$g'$ to all tuples. The filter comparison $\$g' = \$g\$$ eliminates all tuples except those in the current grouping partition identified by the value in column g. Finally, the $sum_{\$a}$ comprehension sums up the $\$a$ column values of the remaining tuples.

Note, because the bulk function aggregates the input relation as a whole and not partitions of it, the grouping column $\$g'$ is static and, hence, the comparison filter will eliminate all tuples except those of the partition group $\$g = 0$. Accordingly, we could omit the outer $array^2$ monad completely. However, for the purpose of demonstration, we included the dummy partitioning step to exemplify a basic pattern for data partitioning and grouping in comprehensions.

To summarize, we can state that the working principles of operators and variable bindings can be explained with the theoretical construct of monads. For the compilation process, however, a monadic specification of each operator is not practical. Instead, we must find a representation, which is easy to handle during the compilation processes, but can be mapped anytime to the theoretical comprehension model, e.g., to justify rewriting rules.

Compiling Monads

As shown in the previous section, monads are perfectly suitable for representing bulk operators and variable bindings in an abstract way. In addition to that, monads also provide an algebraic kernel for our query compiler. With the monad laws, one can derive transformation rules to reorganize and simplify comprehensions in various ways, which in turn serve as solid basement for optimizing operator compositions. For example, [Wad90] states two useful equivalences for implementing predicate pushdown. For two Boolean-valued terms a and b holds

$$[\,t \,|\, b;\, c\,] \quad = \quad [\,t \,|\, (\,b \wedge c\,)\,],$$

and if the Boolean qualifier b is independent of the variables bound by qualifier q, the following equation holds:

$$[\,t \,|\, q;\, b\,] \quad = \quad [\,t \,|\, b;\, q\,].$$

With these rules, conjunctive predicates in a query can be splitted into separate filters, which can then be separately pushed close to the binding qualifiers they refer to, i.e., the inputs of the query.

Further, sophisticated rewrite rules can be formulated inductively by applying the equivalence rules of the monad axioms or by showing the output equivalence of two comprehensions with respect to the respective monad functions. For the systematic application of such rewrite rules, the query optimizer can employ any of the well-known rule-based, heuristics-based, or statistics-based search strategies.

Besides performing structural optimizations, it is crucial to derive efficient execution plans from an abstract comprehension model. Conceptually, comprehensions reflect nested loops over relations, because they are defined on the higher-order functions *map* and $foldr$, which are specified as iterative and recursive operations, respectively. The key to efficiency is the implicit independence of individual loops. Under the premise that side effects do not occur, the order in which loops are processed does not affect the result of most comprehensions, because each variable binding is independent and only valid in the in scope of nested qualifiers. Practically, this allows the compiler to transform and unnest loops to align the nested, inherently sequential evaluation process to set-oriented algorithms and physically advantageous patterns.

4.2.2. AST-based Query Representation

The comprehension model is a powerful foundation for the query compiler, but it does not directly lead to a concrete representation for queries. To obtain a query representation, which reflects the hierarchical nesting of variable-binding expressions and variable-resolving expressions on the one hand and which profits from set-oriented algorithms on the other hand, bulk operations must be modeled appropriately. The goal is to incorporate the algebraic power of comprehensions in a composable, hierarchical representation.

The qualifier lists of comprehensions, which are evaluated from left to right and normalized to nested comprehensions, already yield a top-down view. However, bulk operators, as we defined them, can only be composed in nestings, which effectively form a tree with implied bottom-up data flow. The solution is a redesign of operators to continuation-passing style (CPS) [Rey72, SJ75]. Operators get a continuation function as additional parameter and, instead of returning the result directly, they return now the result obtained by applying the given function on the former result.

The conversion of the sample function `sum-greater-3-bulk` to CPS-style operators is shown in Figure 4.1. In the CPS version, the previously nested `ForEach` has become the top-most function, which returns as result the application of `Select` on the own output relation. `Select`, in turn, filters the input as before, but applies `GroupBy` on the filtered relation and returns the result. The former top-most `GroupBy` is now at the inner-most position and applies its result to a function `End`, which terminates the continuation passing and simply returns the given input as result.

Note that the conversion to CPS-style operators is solely of structural nature. The change preserves the general functionality of each operator, however, it inverts the composition logic. Logically, everything still fits the abstract comprehension model. In fact, monad types have been shown to be equivalent to CPS [Wad90].

The CPS representation delivers the desired top-down style for bulk-oriented operators so that they can be easily integrated into the standard expression tree. Everything is represented in its most natural form – as a tree of scoped expressions. Hence, the compiler operates practically on a hierarchical, AST-like query representation, just like

```
declare function sum-greater-3-bulk($val) {
  GroupBy[$a'](
     Select[$a](
       ForEach[$a](
          [],
          transpose($val)
       ),
       greater-than($a,3)
     ),
   zero,
   sum($a))
}
```

```
declare function sum-greater-3-bulk($val) {
  ForEach[$a](
    [],
    transpose($val),
    Select[$a](
      greater-than($a,3),
      GroupBy[$a'](
        zero,
        sum($a),
        End
      )
    )
  )
}
```

Figure 4.1.: Conversion of bulk operators to CPS.

compilers for conventional programming languages. The initial expression tree, i.e., the AST of the parsed query, must solely be brought to a form where all variable-binding expressions form trees of properly nested of variable scopes.

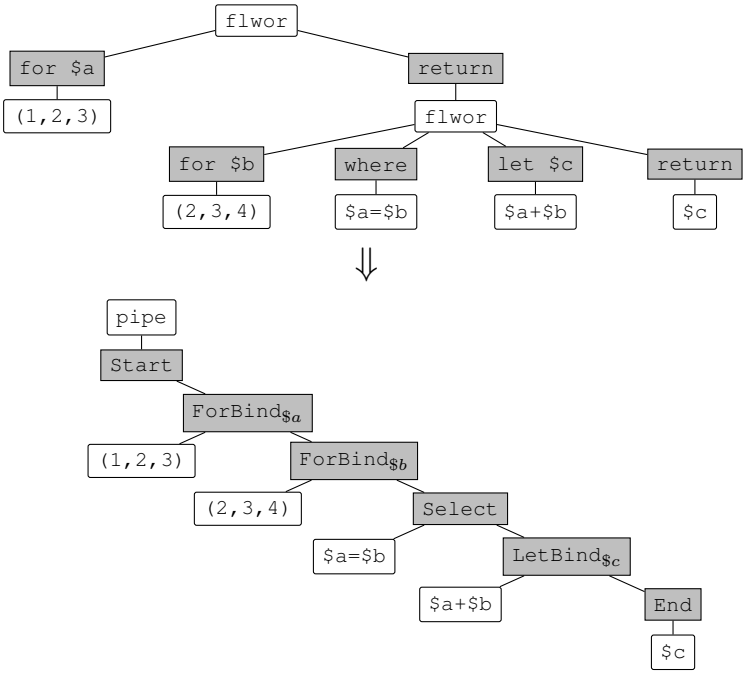


Figure 4.2.: Conversion of a FLWOR expression to an operator pipeline.

4.2.3. FLWOR Pipelines

To compile XQuery, the initial AST must be reshaped to a form where all variable bindings in the query reflect nested variable scopes. Target to the rewriting process are all individual FLWORs as well as nestings of them. The rest of the query is left intact[3]. Figure 4.2 illustrates rewriting of the expression tree[4]. Listing 1 shows the core loop of the rewriting process, which successively replaces all FLWOR expressions.

[3]For brevity, we focus the discussion on FLWORs only. The derivation of correctly nested AST representations for the few other variable-binding expression types like `typeswitch` and `try-catch` is trivial because they do not imply iterative behavior.

[4]In the remainder of this thesis, illustrations of trivial subtrees like arithmetics and comparisons will be collapsed to a single expression node to save space.

Listing 1 Introduction of FLWOR pipelines in the AST.

```
1: function INTRODUCE_PIPELINES(ast)
2:     while ast contains FLWOR do
3:         flwor ← find_FLWOR(ast)
4:         pipe ← create_pipeline(flwor)
5:         ast ← replace_node(ast, flwor, pipe)
6:     end while
7:     return ast
8: end function
```

Every `flwor` expression in the AST is replaced by a `pipe` expression with a right-deep pipeline of operators. The actual pipeline begins with an artificial `Start` operator, which provides the initial input. In a loop over all clauses, an operator is introduced for each clause, which provides its output to the operator created for the following clause. At the end, a special `End` operator terminates the pipeline and binds the result of the `return`-clause expression for each input tuple to an anonymous variable. With exception of the terminating `End`, every operator node feeds its output always to the right-most child, which is the next operator in the pipeline. Accordingly, the subtree of the right-most child virtually represents the continuation function.

During this rewriting process, a first and very small simplification can be performed on the fly. If the `return` clause evaluates itself a FLWOR, it can be directly integrated into the current pipeline. Respective pseudocode for the pipeline construction is given in Listing 2. The mapping function *map_to_operator* in line 19 instantiates the corresponding operator AST nodes for non-`return` clauses.

To simplify the AST, the variable-binding clauses `for`, `let` and `window`, which actually translate to `ForEach` operators with respective `bind` functions, are rendered as the operators `ForBind`, `LetBind`, and `WindowBind`, respectively. In case of `ForBind`, the implicit `bind` function produces a tuple for each item in the sequence obtained by evaluating the attached binding expression. If the `for` clause specifies a position variable, the `bind` function produces an array of 2-tuples with the current item in the first position and its index in the second position. Consequently, the `ForBind` binds in this case two variables.

Listing 2 Creation of FLWOR pipelines.

```
1: function CREATE_PIPELINE(flwor)
2:     pipe ← create_node(PIPE)
3:     prev ← create_node(START)
4:     append_child(pipe, prev)
5:     clauses ← children(flwor)
6:     while clauses is not empty do
7:         clause ← remove_first(clauses)
8:         if clause is RETURN then
9:             expr ← first_child(clause)
10:            if expr is FLWOR then
11:                // continue pipeline with nested FLWOR
12:                clauses ← children(expr)
13:            else
14:                end ← create_node(END)
15:                append_child(end, expr)
16:                append_child(prev, end)
17:            end if
18:        else
19:            op ← map_to_operator(clause)
20:            append_child(prev, op)
21:            prev ← op
22:        end if
23:    end while
24:    return pipe
25: end function
```

The `bind` function of a `LetBind` produces a single tuple for the whole binding sequence.

`WindowBind` operators use the most complex `bind` functions. They produce a relation, which reflects a sliding or tumbling window over the binding sequence. Besides the sequence of items in the current window, the tuples can consist of several other variable bindings, e.g., the first and last items in the current window.

4.2.4. Runtime View

At runtime, the static structure of the expression tree is augmented with the notion of intermediate state, which reflects the values currently bound to variables. With respect to performance, the realization of variable bindings, i.e., the dynamic context, is certainly the most crucial aspect. One option is to treat the dynamic context literally as a set of variables, which is accessible to each expression and updated whenever necessary, i.e., when a new value is bound to a variable. The second option is to represent each specific state during processing as a separate and immutable dynamic context. Each binding of a value to a variable then creates a new copy of the current context.

A set-oriented compiler like the one developed in this thesis, requires the explicit representation of each individual state to compute the result of a FLWOR expression interleaved for multiple or all iterations. Although this implies some overhead to materialize the dynamic context, the performance and scalability advantages usually outweigh. Furthermore, a mutable dynamic context inhibits parallelism, which is another reason, why we build exclusively on explicit but immutable context tuples for each context change.

In the default case, the expression tree is evaluated – like in most XQuery processors – in a sequential fashion. The result is computed in a bottom-up fashion, i.e., subexpressions are evaluated before the actual result sequence is built. Intermediate results are usually not materialized to save main memory. Instead, most expressions evaluate to a lazy sequence, which employs a pull-based iterator concept to compute individual result items on demand [BBB+09]. In general, this makes even processing of very large results very space efficient.

The dynamic context is stored in context tuples which are passed between expressions and operators. A tuple `[1,2]`, for example, can represent a dynamic context with the variable bindings `$a=1` and `$b=2`. Variable reference expressions like the operands of the summation expression `$a+$b` evaluate then to respective variable values in the given context tuple. In analogy to procedural languages, a context tuple reflects the stack at a specific point in time.

ForBind$_{\$a}$

$a
1
2
3

ForBind$_{\$b}$

$a	$b
1	2
1	3
1	4
2	2
2	3
2	4
3	2
3	3
3	4

Select

$a	$b
2	2
3	3

LetBind$_{\$c}$

$a	$b	$c
2	2	4
3	3	6

End

$a	$b	$c	
2	2	4	4
3	3	6	6

Figure 4.3.: Output of the pipeline operators of Figure 4.2.

The fundamental difference between normal expressions and bulk-oriented operators is that the former are evaluated for a single context tuple at a time and produce a result value, whereas the latter consume and produce streams or arrays of context tuples. The `pipe` expression at the top of operator pipelines mediates between the two processing modes. It passes the single context tuple on to the `Start` operator, which feeds it as a single-tuple relation to the pipeline, and builds the final result sequence by merging the anonymous `return` variable values in the output tuples of the final `End`. The output relations of the pipeline operators of Figure 4.2 are shown in Figure 4.3.

Last but not least, function calls are realized with the very same mechanisms. Within a function, parameters appear like references to variables that are bound by the caller. The only difference between function calls and normal expressions is that functions only have access to their parameters and not to the variables bound in the scope where the function is called.

To call a function, the argument expressions are evaluated and the resulting arguments are passed on to the function as a normal context tuple. The result value is returned to the caller. If necessary, dynamic type checking is performed on both the arguments and the result.

4.3. Compiler Architecture

The flexibility and portability ambitions of a data processing platform come at a certain cost. Monolithic solutions, which tightly integrate physical aspects of the storage in the compilation process can per-

form optimizations, which are out of reach of a more general approach. Nevertheless, a modular compiler with a set of solid core algorithms and optimizations that are complemented with storage-specific extensions is able to achieve competitive performance, too. For example, recent successes of the language-independent compiler infrastructure LLVM demonstrate well that modular architectures can achieve top results, when they are equipped with strong platform-specific code generators [LA04]. Besides, such infrastructures excel in categories like portability, stability, and time-to-market, because a large fraction of core optimizations is required in every specialized compiler, too.

The main objective of our compiler architecture is a clear-cut separation of set-oriented aspects from physical aspects. The optimization of the former is given priority in the compilation process, because wrong decisions here easily result in query plans which are orders of magnitude slower. Despite, physical aspects increasingly come into focus on modern hardware platforms, too. For example, storage subsystems and even whole DBMSs are today already tuned for data cache and instruction cache locality.

4.3.1. Compilation Pipeline

The compilation process consists of several rewriting stages and the final assembly of the executable plan. An overview of the rewriting process is given in Figure 4.4. The initial expression tree is created directly by the parser and processed by individual rewriting stages, which annotate and transform it for the final translation to an executable plan. In the following, we will make a brief walk-through of the whole process. The highlighted stages are in the focus of this work and are covered in detail in Section 4.2.3 and the following chapters, respectively.

In the first step, syntactic sugar and functional redundancy in the initial expression tree is normalized. For example, quantified expressions are converted to FLWOR expressions as shown in Section 3.2.4. However, in contrast to other compilers, not everything is strictly normalized to basic language constructs of the XQuery Core model [W3C10b]. Among others, this would break up all path expressions into sequences of FLWOR expressions, and, as we will see in Chapter 6, it is beneficial to preserve data access operations like path expressions and filter expressions in the first place.

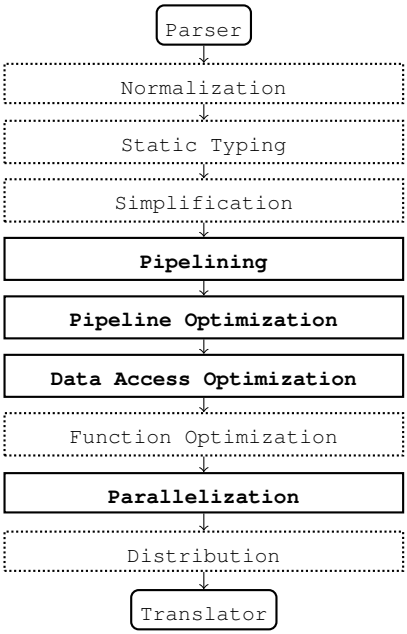


Figure 4.4.: The query rewriting pipeline.

The next stage performs static typing. The query is checked for type errors and all expressions are annotated with type information for data-type-specific optimizations in subsequent steps.

The simplification stage is intended for standard pruning operations like the removal of unused variables, constant folding and dead-code elimination. Furthermore, type information derived in the previous stage is used to simplify expressions, which can be confined to certain types, and to eliminate otherwise mandatory runtime checks, e.g., for function arguments and return values.

In following stage, FLWOR expressions are transformed into operator pipelines as explained in Section 4.2.3. The rewriting stage itself is rather simple, because it simply brings the expression tree into a different form.

After bringing all FLWORs into the nested pipeline form, various pipeline-local optimizations like early filtering, join rewriting, sort elimination, etc. are performed. Furthermore, nestings of pipelines are fusioned to maximize the potential of set-oriented optimizations. Details on the central rewriting rules are given in Chapter 5.

The following stage takes care of physical data accesses, i.e., path expressions and related operations. Naturally, the rewritings performed in this stage heavily depend on the target platform, but in essence, this stage introduces platform-specific "expressions", which are later compiled into native operations instead of generic storage adapter calls. As the selection of DB-specific optimizations presented in Chapter 6 illustrates, data access optimization takes places in this stage at multiple granules and in multiple dimensions.

The next stage optimizes function calls. It is primarily important for user-defined functions, because each declared function is itself a query. Accordingly, opportunities for optimization are high. Non-recursive functions, for example, can be inlined to avoid the overhead of function invocation and to increase chances for further optimization. The function call is replaced in the expression tree by the function body and references to function arguments are replaced by the corresponding argument expressions. However, if the statically available typing information is not sufficient to guarantee correct types of arguments and results, additional code must be generated to discover typing errors at runtime. In the normal case, respective checks are integrated into the dynamic function call mechanism. Recursive functions can benefit from other well-known compiler techniques like tail-call optimization [ASS85]. Higher-order functions and partial function application can draw from the compilation of functional languages [PL92].

To improve parallel processing, the next stage allows to enhance the expression tree with additional information like the maximum degree of parallelism for operations or suitable buffer sizes. Note, our parallel execution framework presented in Chapter 7 exploits query-inherent data parallelism automatically at runtime and does not need a structural reorganization of the expression tree itself.

Finally, we envision a stage for taking care of the special requirements in distributed environments. These, however, have not been studied in the context of this thesis, but the field of MapReduce-based languages [BEG+11, ORS+08] suggests itself as starting point for future work.

4.3.2. Plan Generation

The final expression tree is compiled into an executable query plan in a single top-down pass. The compilation is straightforward as the pseudo code in Listing 3 shows. For each expression type, there is a concrete implementation, which is initialized with the compiled subexpressions and, if present, annotated information in the AST. The translation process is that simple because each subexpression, i.e., each subtree in the AST, is self-contained and does not have to obey other dependencies than plain variable references.

Listing 3 Translation of expressions.

```
1: function COMPILE_EXPRESSION(ast)
2:     if ast is LITERAL then
3:         expr ← compile_literal(ast)
4:     else if ast is AND then
5:         expr ← compile_and_expression(ast)
6:     else if ast is OR then
7:         expr ← compile_or_expression(ast)
8:     else if ... then
9:         ...
10:    end if
11:    return expr
12: end function

13: function COMPILE_AND_EXPRESSION(ast)
14:    f ← child(0, ast)
15:    first ← compile_expression(f)
16:    s ← child(1, ast)
17:    second ← compile_expression(s)
18:    expr ← create_and_expression(first, second)
19:    return expr
20: end function
```

Variable bindings are handled completely by the respective binding expression or operator. A variable must be declared before subexpressions in the visibility scope of the variable are compiled, and it must be undeclared again when the variable goes out of scope, i.e., after the

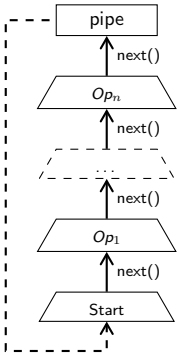
respective subexpressions have been compiled. A variable table keeps track of all variables declared and assigns them a binding position in the corresponding scope. In subexpressions with a variable dependency, a simple lookup in the variable table is sufficient to resolve the corresponding position in context tuples. Accordingly, a variable reference translates at runtime into a simple array access in a context tuple.

The special scope variables `$fs:dot`, `$fs:last`, and `$fs:pos` are treated like normal variable declarations. Expressions modifying this part of the context (e.g., filter expressions) declare them as special variables, which are resolved during compilation of expressions referring to these elements. Listing 4 illustrates the variable binding and resolution process for filter expressions and the context size expression `last()`.

Listing 4 Binding and resolution of variables.

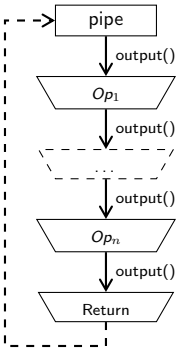
```
1: function COMPILE_FILTER_EXPRESSION(ast)
2:     input ← child(0, ast)
3:     e ← compile_expression(input)
4:     predicate ← child(1, ast)
5:     bind(table,fs:dot)
6:     bind(table,fs:last)
7:     bind(table,fs:position)
8:     p ← compile_expression(predicate)
9:     bind_i ← unbind(table,fs:position)
10:    bind_l ← unbind(table,fs:last)
11:    bind_p ← unbind(table,fs:dot)
12:    expr ← create_filter_expression(e, p, bind_i, bind_l, bind_p)
13:    return expr
14: end function

15: function COMPILE_CONTEXT_SIZE_EXPRESSION(ast)
16:    name ← variable_name(fs:last)
17:    position ← lookup(table, name)
18:    expr ← create_tuple_access(position)
19:    return expr
20: end function
```




```
type Cursor {
  void open()
  Tuple next()
  void close()
}
```

(a) Pull-based pipeline




```
type Sink {
  void begin()
  void output(Tuple t)
  void end()
}
```

(b) Push-based pipeline

Figure 4.5.: Comparison of pull-based and push-based pipelines.

The compilation of operator pipelines is almost identical to the compilation of expressions. Beginning at the `Start` operator of the corresponding `pipe` expression, the pipeline is compiled in a recursive top-down pass along the output edges. The binding and resolution mechanism for variables is also identical. Pseudocode for the compilation step is found in Listing 17 and Listing 18 in Appendix A.

Note that the actual realization of an operator pipeline is independent of the compilation process. The system can either employ a pull-based or push-based operator model as depicted in Figure 4.5(a) and Figure 4.5(b), respectively. The pull-based model is widely used in all kinds of database systems, because it matches the bottom-up perspective of conventional query plans. The push-based model is almost identical, but better suits to the top-down perspective.

In the pull-based model, each operator is realized as a cursor, which implements a simple open-next-close interface [Gra94]. The client, in our case the compiled pipe expression, initializes the cursor pipeline

by passing on the start tuple the Start operator. Then, it iteratively retrieves the result by calling *next()* on the cursor of the last operator in the pipeline, which propagates the call through the individual pipeline stages. The push-based model works in the exact opposite direction. Each operator is represented as a sink, which receives a tuple stream as input and emits a tuple stream to another sink. The pipe expression emits the start tuple to the first operator in the pipeline by calling *output()* and collects the results, which are emitted by the final operator in the pipeline.

5. Pipeline Optimization

The conversion of FLWORs to operator pipelines is the enabling step for improving the scalability aspects of a query, because it opens the door for various set-oriented rewritings of the operator tree. Generally, the compiler can employ any prevalent optimization from the relational world like projection, predicate merge, sort elimination, etc. As long as the final stream of output tuples is not affected, operators can be reordered, merged, or replaced to obtain a better performing pipeline.

Besides the fact that a pipeline reflects in principle an upside down version of a relational query plan, the rewriter must additionally take care of the data-model-inherent order sensitivity. However, in many data processing scenarios, users may safely override strict order preservation to increase query performance.

The following discussion concentrates on optimization rules for join processing and aggregation, which are special to or particularly effective in the context of nested top-down pipelines. But beforehand, some general aspects of rewriting rules and binding operators will be introduced with the most obvious pipeline optimization: predicate pullup – the equivalent to predicate pushdown in a bottom-up query plan.

5.1. Generalized Bind Operator

Due to the hierarchical nesting of variable binding operators and dependent expressions, the realization of predicate pullup is quite simple. A simple look at the ancestors of a `Select` operator in the AST reveals the operators, which bind the variables on which the predicate expression depends. This information is sufficient to decide whether and to where the filter can be pulled up in the pipeline to reduce the number of intermediate tuples. In Figure 5.1, for example, the predicate `$a>1` depends only on variable `$a` and, thus, the `Select` can be placed directly under the `ForBind` that makes `$a` available in the local scope.

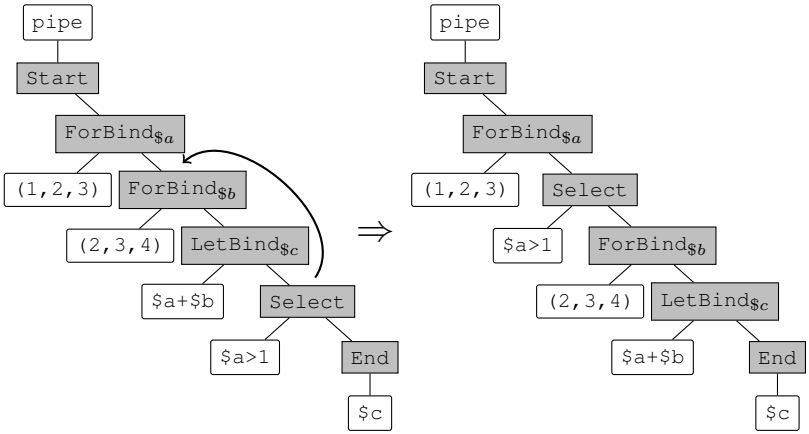


Figure 5.1.: Predicate pullup in the operator pipeline.

Consideration of such variable dependencies is crucial for almost every rewriting rule. As the example shows, the proper nesting of scopes in the AST makes it easy to keep track of variable dependencies. The source of a binding can be resolved by looking at ancestor nodes; uses of a binding can be found by traversing the subtree of the binding node. The notation of rewrite rules, however, gets cumbersome, because many rules will refer to structural relationships with wildcard patterns like "the ancestor that binds variable `$a`".

To allow for a crisp notation of rewrite rules, we can make use of the fact that most operators in the AST reduce to specializations of `ForEach`. Except `Project`, `OrderBy`, `GroupBy`, and `Count`, all operators virtually bind zero or more variables and emit for each input tuple the Cartesian product with an input-dependent relation, thereby preserving the relative order of input tuples. Commonly, these operators can be referred to with a generalized operator $\texttt{Bind}_{\$X:\$V}$, characterized as a `ForEach`-based operator, which binds additional variables $\$X = \$x_1, \ldots, \$x_n$ and depends on variables $\$V = \$v_1, \ldots, \$v_n$ to compute for each input tuple a binding relation for emitting the Cartesian product.

Because the Cartesian product on relations is associative, a sequence of two operators $\texttt{Bind}_{\$X:\$V}$ and $\texttt{Bind}_{\$Y:\$W}$ exposes, when considered

as a whole, the same properties as a single one. Hence, it is valid to merge them to a single operator $\texttt{Bind}_{\$X,\$Y:\$V\cup\$W}$. In rewriting rules, we can use this property to collapse sequences of `ForEach`-based operators as a match for a single `Bind` operator. Accordingly, the rewriting rule for pulling up `Select` predicates can be formulated as shown in Rule 1. Note, the rule is applicable in Figure 5.4 because the two consecutive operators $\texttt{ForBind}_{\$b}$ and $\texttt{LetBind}_{\$c}$ can be matched together as $\texttt{Bind}_{\$b,\$c:\emptyset}$.

Rule 1 Pullup of `Select` predicate.

$$\frac{\begin{array}{cc}\$x_1,\ldots,\$x_n \Rightarrow \$X & \$v_1,\ldots,\$v_m \Rightarrow \$V \\ \texttt{Bind}_{\$X:\$V} \Rightarrow Op_x & \textit{pred} \text{ independent of } \$X\end{array}}{Op_x(\texttt{Select}(\textit{pred}, \textit{out})) \quad \Rightarrow \texttt{Select}(\textit{pred}, Op_x(\textit{out}))}$$

The reordering of binding operators within pipelines – remember `Select` is also a `Bind` operator – is advantageous in many situations, e.g., for bringing a pipeline into an appropriate form for join rewriting or to extract expensive, loop-invariant `let`-bound variables out of a `for` loop. However, two consecutive `Bind` operators may only be swapped under the following conditions:

1. The nested `Bind` is independent of the variables bound by the parent operator.
2. The nested `Bind` will not shadow variables on which the parent depends, i.e., it does not re-bind values to variables on which the parent depends[1].
3. The swapping does not influence the ordering of output tuples. This is the case when at least one of the two `Bind` operators emits at most one tuple per input tuple. This suggests particularly `Select` and `LetBind` operators as potential candidates for reordering. Alternatively, it is allowed to swap to `Bind` operators if the user specified that the (partial) order of tuples must not be preserved or if the reordering does not affect the final outcome

[1] Implementations may circumvent this problem by internally assigning unique variable names.

of the pipeline. The latter property, however, is very difficult to prove in the general case.

4. Neither of both operators causes side effects or produces otherwise indeterministic results.

The correctness of pipeline rewritings can be shown with equivalences in the monad comprehension model. For example, correctness of Rule 1 is given by the respective equivalence rule mentioned in Section 4.2.1. Usually, common sense behind a rewriting rule is evident and, because this thesis puts emphasis on the realization aspects of a data programming language, we abstain from formal proofs for rewriting rules presented.

5.2. Join Processing

The looping nature of `ForEach`-based operators frequently results in pipelines which compute expensive Cartesian products. Efficient join support is therefore crucial to fight the data explosion of nested loops.

To get familiar with the representation of joins in top-down pipelines, consider again the pipeline of Figure 4.2. Two consecutive `ForBind` operators compute the cross product of the sequences `(1,2,3)` and `(2,3,4)`, which is filtered afterwards with a join-like equality predicate. Obviously, the direct nesting of the two `ForBind`s can be interleaved with a `Join` as shown in Figure 5.2.

The `Join` node has three children. The right-most child is, as for all operators, the pipeline to which the output is fed. The first and the second child produce the left and right join input relations, respectively, i.e., they reflect the join parameter functions `left` and `right`. The equality predicate of the `Select` has been splitted. The operand expressions have become the join keys, which are located under the `End` terminators of the left and right input pipelines for correct scoping. The comparison type is a property of the `Join` node, which is indicated by the = subscript in the example.

The join is computed exactly as described in Section 2.3.3. The tuples of the input relation `[[1],[2],[3]]`, are successively fed to the join input pipelines, and their respective outputs are pairwise matched for the join predicate. Notably, the left join input is a trivial pass-through,

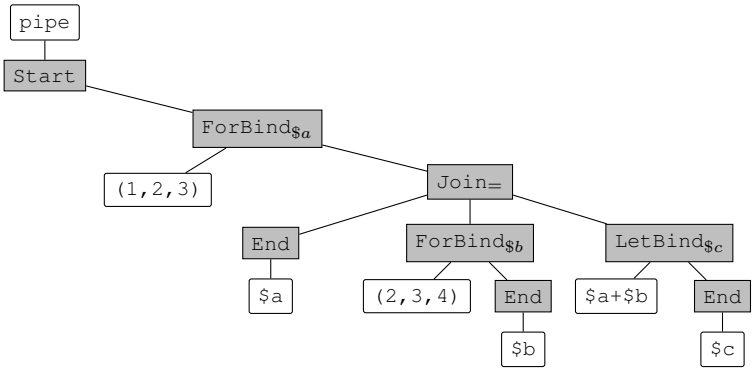


Figure 5.2.: Introduction of a basic join in the pipeline of Figure 4.2.

which only provides the left join key for the input tuple. In turn, the right join input binds the sequence `(2,3,4)` and emits three tuples per input tuple including their join keys.

Without further assistance, the computation of the join will result in the same nested-loops behavior as the nested `ForBind` operator, because the `Join` operator actually computes and concatenates three results of three separate joins – one join per input tuple. Hence, even if the equi join was implemented as a hash join, the hash table for the inner table, i.e., the right input, had to be rebuild for each input tuple, which effectively would make a hash join even more costly than simple nested loops. However, the right input is independent of the variable `$a`, which allows to push the outer `ForBind` down to the left join input branch as shown in Figure 5.3. Now only one join is computed – one for the single input tuple emitted by `Start`. If the equi join is then implemented as a hash join, the initially nested binding sequence `(2,3,4)` has to be processed only once to build the hash table.

5.2.1. Join Recognition

In contrast to SQL, the front-end language XQuery is not able to express joins explicitly. Accordingly, the optimizer must detect join semantics in a pipeline, i.e., the filtering of a Cartesian product, by itself. Without loss of generality, a Cartesian product can be confined to a nesting of two

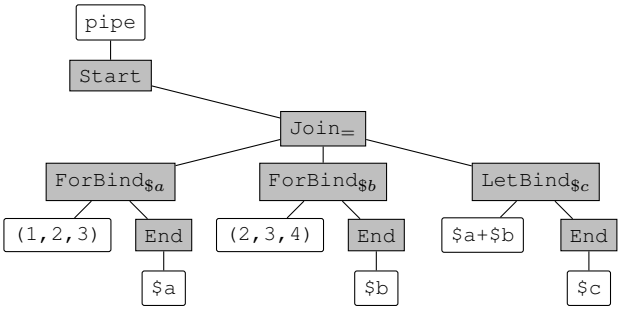


Figure 5.3.: Join with two independent inputs.

arbitrary but independent `Bind` operators. Independent means that the inner `Bind`, is independent of the variables bound by the outer `Bind`. If the Cartesian product is immediately followed by a `Select` with a comparison predicate $\Theta \in \{=, <, >, <=, \ldots\}$ and if the two operand expressions disjointly depend on variables bound by either the inner or by the outer `Bind`, then this operator sequence has join semantics. Rule 2 and Rule 3 show the respective rewriting patterns, which are illustrated in Figure 5.4.

Rule 2 Introduction of `Join`.

$$\frac{\begin{array}{c}\$x_1, \ldots, \$x_n \Rightarrow \$X \qquad \$v_1, \ldots, \$v_m \Rightarrow \$V \\ \$y_1, \ldots, \$y_l \Rightarrow \$Y \qquad \$w_1, \ldots, \$w_k \Rightarrow \$W \\ \texttt{Bind}_{\$X:\$V} \Rightarrow Op_x \qquad \texttt{Bind}_{\$Y:\$W} \Rightarrow Op_y \\ Op_y \text{ independent of } \$X \\ k_1 \text{ independent of } \$Y \qquad k_2 \text{ independent of } \$X \\ \Theta \in \{=, <, >, <=, \ldots\}\end{array}}{\begin{array}{c}Op_x(Op_y(\texttt{Select}(k_1 \Theta k_2, out)) \\ \Rightarrow Op_x(\texttt{Join}_\Theta(\texttt{End}(k_1), Op_y\texttt{End}(k_2), out)))\end{array}}$$

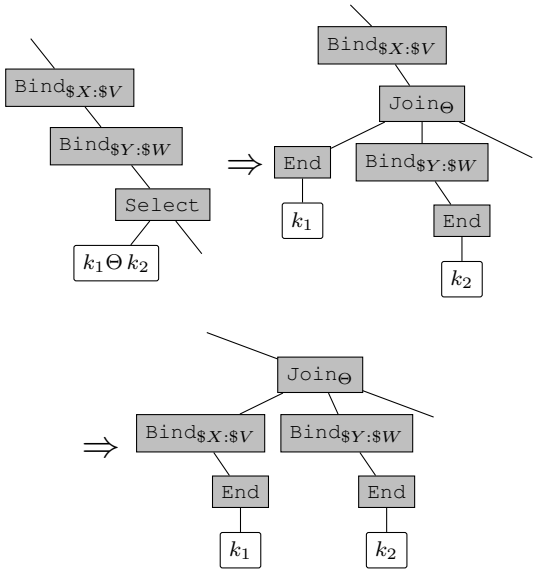


Figure 5.4.: Application of join rewriting rules 2 and 3.

Rule 3 Push-down of independent bindings to left join input.

$$\frac{\begin{array}{c}\$x_1,\ldots,\$x_n \Rightarrow \$X \qquad \$v_1,\ldots,\$v_m \Rightarrow \$V \\ \texttt{Bind}_{\$X:\$V} \Rightarrow Op_x \qquad \texttt{Join}_{\Theta} \Rightarrow Op_j \\ \textit{Right} \text{ independent of } \$X \qquad k_2 \text{ independent of } \$X \\ \Theta \in \{=, <, >, <=, \ldots\}\end{array}}{\begin{array}{c}Op_x(Op_j(Left(\texttt{End}(k_1)), Right(\texttt{End}(k_2))), out) \\ \Rightarrow Op_j(Op_x(Left(\texttt{End}(k_1)), Right(\texttt{End}(k_2))), out)\end{array}}$$

The sample query in Figure 4.2 contains the simplest pipeline constellation that fits the join rule pattern. In the `Select` predicate, the left-hand operand, the variable reference `$a`, depends solely on the outer $\texttt{ForBind}_{\$a}$, whereas the right-hand operand `$b` depends solely on the inner $\texttt{ForBind}_{\$b}$. Furthermore, the binding sequence of the inner $\texttt{ForBind}_{\$b}$ is independent of `$a`, too.

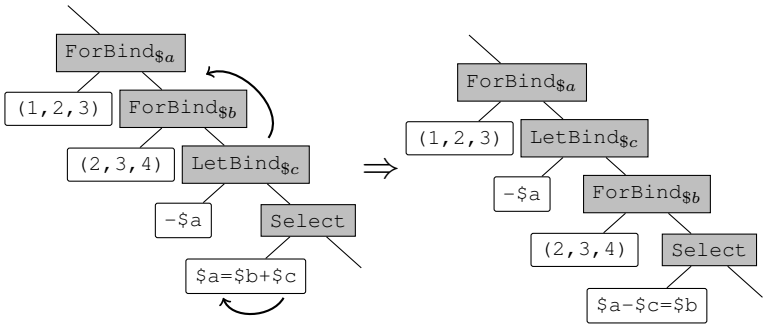


Figure 5.5.: Pipeline reshaping for join processing.

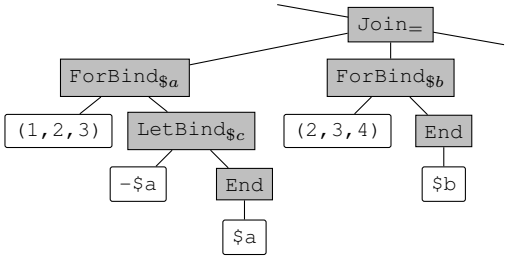


Figure 5.6.: Join in reshaped operator pipeline.

5.2.2. Pipeline Reshaping

Figure 5.5 shows a more difficult example of join semantics. In the left expression tree, both variable references in the right predicate operand are bound after `$a`, but `$c` directly depends on variable `$a`. Thus, a join rewriting cannot be performed. However, there is the chance to rewrite the query to the expression tree shown at the right-hand side of Figure 5.5. Now, there is a proper separation of bindings and references, which can be assigned to the left and to the right input of the join in Figure 5.6, respectively.

Obviously, reshaping of a pipeline to match the join pattern can become arbitrarily complex. Especially, if an arbitrary join predicate has

to be adjusted as in the example above. Note also that it was here only possible to move the critical variable binding `$c` because it was `let`-bound and because the binding expression is side-effect-free.

Of course, join semantics may also be embodied in all FLWOR-based or FLWOR-like expressions, e.g., in filter expressions of the general form $s_2[e_2 \Theta e_1]$, where e_2 is a predicate over the inner context provided by s_2 and e_1 is a predicate over the context of a surrounding loop over an external sequence s_1. In these cases, we may rewrite the respective expression to their equivalent FLWOR representation and compute the join in the pipeline.

In summary, we can state that the presented join rewriting rule is capable to cover a wide range of typical join scenarios, but there is no guarantee that hidden join semantics will always be detected.

5.2.3. Join Groups

Some aspects of XQuery make join processing conceptually more complex than in SQL. First, there is the strong emphasis on tuple order, which is not preserved automatically in many standard algorithms. Second, comparisons in XQuery follow subtle typing and type conversion rules – a concession to work smoothly with untyped and semi-structured data. In consequence, a join operator has to cope with untyped data and mixed-type sequences if the types of the operands cannot be statically determined.

As result of the above restrictions, a sort-merge join is only useful if the operands can be determined to be of a single type beforehand and if output order can be ignored or is compensated by an explicit sort afterwards. More typically, join algorithms will keep one input (the inner) in a lookup table, e.g., a hash table or a sorted index and probe it with the other input (the outer). Further implementation details can be found in [RSF06].

An important aspect of lookup-table-based algorithms is the possibility to reuse a lookup table. Reconsider the initial join rewriting in Figure 5.2. The empty left input pipeline implied a rebuilding of the lookup table for each incoming tuple – although the right input and, thus, the lookup table was not affected. In Figure 5.3, we fixed the issue by pushing the ancestor operators from which the right input is independent of to the left input.

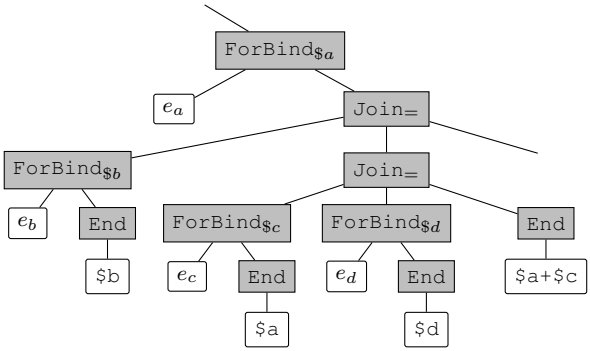


Figure 5.7.: Potential inefficiency of nested joins.

In more complex situations or in case of deep nestings of both FLWOR and non-FLWOR expressions, it is not possible to push all unrelated binding operators to the left input. Then again, the lookup table may have to be rebuilt more often than necessary.

Consider the example in Figure 5.7. Let all binding expressions e_a, e_b, e_c, and e_d be free of variable dependencies. The top-most operator `ForBind`$_{\$a}$ potentially binds multiple values to `$a`, which leads to multiple evaluations of the top-most join. Because the third branch of the nested join is dependent on `$a`, `ForBind`$_{\$a}$ cannot be pushed down to the left input branch. Hence, the nested join, i.e., the right input of the parent join, must be recomputed of for each iteration of `$a`, which, in turn, triggers a re-build of the lookup table, although the right input of the nested join is not affected by the changed binding of `$a`.

Clearly, the above situation is unsatisfactory. Therefore, the prototype developed in this thesis keeps a loaded lookup table until a recomputation is actively triggered or until the query has finished. The trigger mechanism is transparently embedded in the tuple flow. For each join, a `Count` operator is introduced directly after the lowest ancestor that binds a variable upon which the right join input depends. Every change of the counter variable then indicates a new binding situation. Accordingly, the join must simply check for each input tuple whether or not the counter variable changed, i.e., whether or not the lookup table must be recomputed.

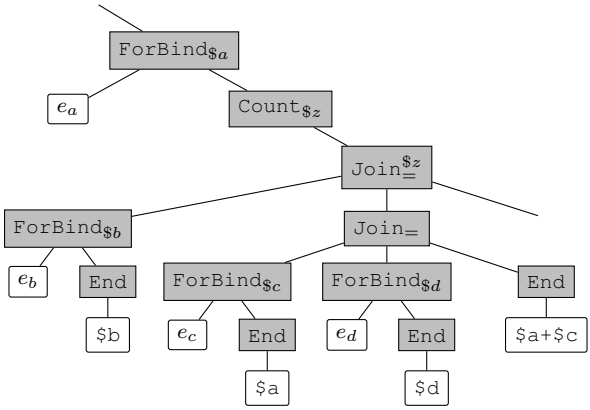


Figure 5.8.: Use of trigger variable for join table rebuilding.

Figure 5.8 renders the situation for the example. An additional operator $\texttt{Count}_{\$z}$ was introduced directly after $\texttt{ForBind}_{\$a}$ and the top-most join was annotated accordingly to trigger a rebuild of the lookup table on each change of `$z`. For the nested join, such a trigger is not necessary because its right join input is independent of foreign variables. Hence, the lookup table of the nested table is computed only once and reused when the parent join re-builds its lookup table.

From an abstract view point, the trigger mechanism allows to share intermediate results between a sequential group of individual nested iterations. We call them *join groups* and refer to the respective rewriting rule accordingly as join group demarcation:

Rule 4 Join group demarcation.

$$\frac{\begin{array}{c}\texttt{Join}_{\Theta}\,(left, right, out) \Rightarrow Op_1 \qquad \texttt{Bind}_{\$a} \Rightarrow Op_2 \\ \$a \Rightarrow \text{latest-bound variable on which } right \text{ depends} \\ \Theta \in \{=, <, >, <=, \ldots\}\end{array}}{\begin{array}{c}Op_1 \Rightarrow Op_1^{\$join-grp} \\ Op_2(out) \Rightarrow Op_2(\texttt{Count}_{\$join-grp}(out))\end{array}}$$

Note that this optimization principle directly applies to the similarly-structured operators `Concat`, `Union`, and `Intersect`, too. If either the left, the right, or both inputs are stable throughout multiple iterations (i.e., groups of consecutive input tuples), it can be memoized to reduce compuation overhead.

5.3. Pipeline Lifting

So far, we only considered `for`-bound variables, which hold the single items of a binding sequence. However, XQuery also allows to bind whole item sequences to variables, e.g., with `let` bindings as exemplified by the query in Figure 5.9. After rewriting the corresponding expression tree, we obtain a nesting of two pipelines as depicted in Figure 5.10. The operator $\texttt{LetBind}_{\$c}$ evaluates for each incoming context tuple a `pipe` expression for the rewritten FLWOR expression of the initial binding and binds the entire result to variable `$c`. Accordingly, it produces the `[$a,$c]` tuples `[1,(2,3,4)]`, `[2,(3,4)]`, `[3,4]`, and `[4,()]`.

```
for $a in 1 to 4
let $c:= for $b in 2 to 4
         where $a<$b
         return $b
return ($a,$c)
```

Figure 5.9.: Nested FLWOR in `let` binding.

Technically, tuples of sequences instead of single items are sometimes disruptive but rarely a problem. Small to medium-sized sequences can be stored inline in a tuple; bindings of large sequences may be realized as pointers to the respective sequences, which may reside in memory or on external storage. Remember, it is often also possible to bind just a lazy sequence, i.e., a closure, that will compute its items on demand.

In many cases, it is reasonable to compute `let`-bound sequences directly within the pipeline, e.g., because it offers more options for optimizations, or because cheap lazy sequences that simply flow through the pipeline can cause significant load skew during parallel processing – the single process at the pipeline end does all the real work.

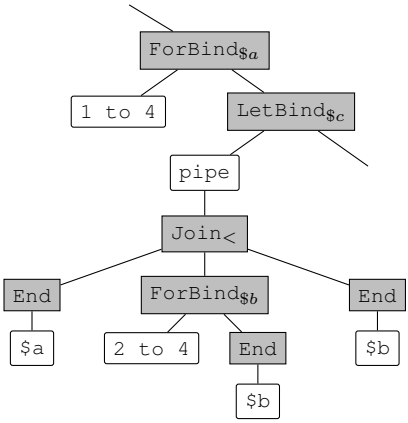


Figure 5.10.: Nested FLWOR pipelines.

The nested join in Figure 5.10 reveals a fairly common pattern in XQuery. Outer join semantics is expressed with a nested FLWOR, because the language does not provide an explicit join primitive. Unfortunately, initialized pipelines and, thus, also the lookup tables built for joins are lost between evaluations of a nested FLWOR, because an expression is, in contrast to an operator, evaluated only for a single context tuple at a time. In general, pipeline sharing across multiple evaluations is viable, but violates the principle that expression evaluations are independent and evaluation order is only partially specified, i.e., expressions can be evaluated in parallel. Hence, the problem of multiple join evaluations must be solved here differently.

5.3.1. 4-way Left Join

To overcome potential inefficiency of FLWOR nestings, we perform *pipe lifting*: In several rewriting steps, we "lift" pipelines nested in `LetBind` operators and integrate them into the higher-level pipeline. In the first step, we apply Rule 5, which converts the `LetBind` with the nested `pipe` expression to a left join as shown in Figure 5.11.

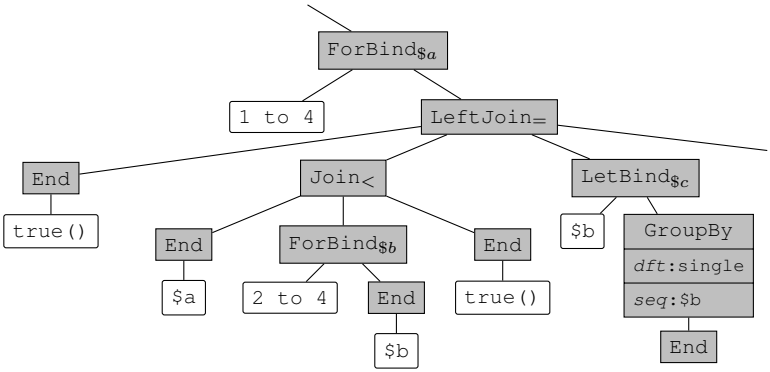


Figure 5.11.: Conversion of `let`-binding to 4-way `LeftJoin` with post-join pipeline.

Rule 5 `LetBind` to `LeftJoin` conversion.

$$\frac{\texttt{End}(\,(\mathit{true}())) \Rightarrow \mathit{end}_1 \qquad \texttt{End}(\,(\mathit{true}())) \Rightarrow \mathit{end}_2}{\begin{array}{c}\texttt{LetBind}_{\mathit{var}}(\texttt{pipe}(\mathit{Op}(\texttt{End}(e))), \mathit{out}) \Rightarrow \\ \texttt{LeftJoin}_{=}(\mathit{end}_1, \mathit{Op}(\mathit{end}_2), \texttt{LetBind}_{\mathit{var}}(e, \texttt{GroupBy}(\texttt{End})), \mathit{out})\end{array}}$$

Note, the rendered `LeftJoin`$_=$ operator has four children. The two first children are the two join input pipelines, the third child is a special *post-join pipeline*, and the last child is the normal output pipeline.

The post-join pipeline implements a post-processing step, which restores the grouping semantics of the rewritten `LetBind`. The principle is as follows: Each group of left-join matches, i.e., the join result constructed from one tuple of the left input and the matching tuples from the right input, is fed to the post-join pipeline before the result is passed on to the output. For `$a=1`, these are the tuples `[1,2]`, `[1,3]`, and `[1,4]`; for `$a=2` these are the tuples `[2,3]` and `[2,4]`; and for `$a=3`, it is the tuple `[3,4]`. For `$a=4` there is no join partner in the right input, thus the left-join tuple `[4,()]` does not pass through the post-join pipeline.

Conceptually, the 4-way `LeftJoin` is a macro, which maps to the original 3-way left-join operator as depicted in Figure 5.12. In the expanded version, a counter variable `$grp` is introduced to define the virtual post-join partitions described above. The optional nature of the post-join pipeline is realized by a `Concat` operator, which, dependent on whether or not a left input tuple found a join partner, processes or skips the *post* pipeline fragment, respectively. We introduce the 4-way representation not only for simplifying the AST, but also because it is easier translated into a more efficient query plan.

The most interesting part of the lifted section is the `GroupBy` step at the end of the post-join pipeline. In the AST, the grouping function for `GroupBy` is expressed as a sequence of child expressions, which evaluate to primitive, SQL-like grouping keys. In the 4-way left join, the `GroupBy` does not specify any grouping keys, which means that all incoming tuples will belong to the same partition. In the normalized the 3-way left join, the `count` variable `$grp` serves as grouping key.

The aggregation of non-grouping columns in a partition is specified by additional operator properties. The default aggregation type for all columns is specified by the property *dft*. In the 4-way case, it is `single`, which means that one column value is picked as aggregate for all column values. The default aggregation type is overridden for the `$b` column, which is aggregated by combining all values to a single sequence as indicated by the property *seq*`:$b`. Further details about the specification of aggregates will be given in Section 5.5.

Putting it all together, the left join in Figure 5.11 emits tuples of the form `[$a,$b,$c]`[2]. Assuming, that aggregation type `single` picks the column value of the first tuple in a partition, the respective output tuples are `[2,2,(3,4)]`, `[3,3,4]`, and `[4,4,()]`. Except the additional binding column `$b`, this output is equivalent to the original `[$a,$c]` output tuples of $\texttt{LetBind}_{\$c}$ in Figure 5.10. Logically, variable `$b` is now out of scope and therefore does not influence subsequent operations. Physically, the additional column is harmless ballast, but it can be dropped for efficiency.

[2] XQuery's grouping clause simply re-binds aggregated columns to the same variable names.

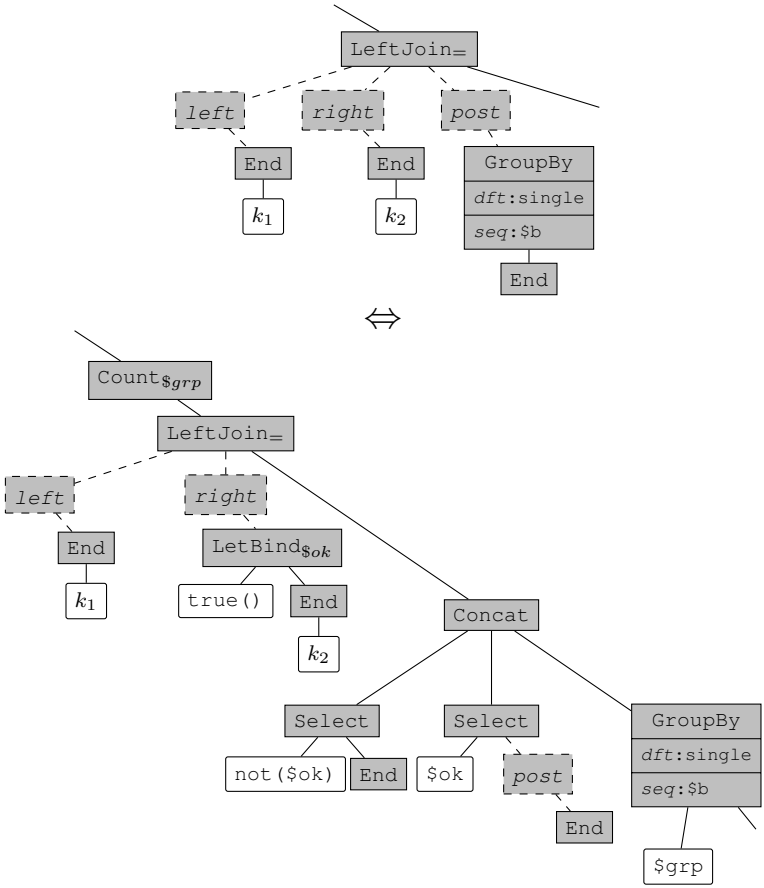


Figure 5.12.: Expansion of 4-way `LeftJoin` macro.

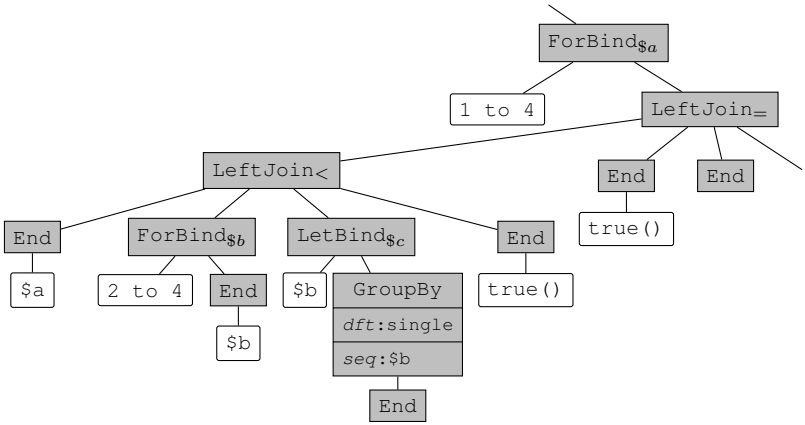


Figure 5.13.: Trivial join nesting after lifting.

5.3.2. Lifting Nested Joins

After converting the nested pipeline to the right input of a left join, we can employ standard join group demarcation to effectively prevent continuous rebuilding of the lookup table in the nested join. However, considering the trivial shape of the introduced left join allows further simplification.

The left join input is empty; it solely echoes the respective input tuple. Hence, the two join branches can be swapped without tampering output order. However, swapping the inputs of a left join requires us to convert the topmost join of the right input into a left join. Of course, the input grouping for the post-join pipeline must also be observed. The post-join pipeline is therefore moved to the left input side, turning the converted 3-way left join into a 4-way left join as shown in Figure 5.13.

If we consider a join as just another representation form of nested loops, the performed input swapping effectively "lifts" the nested iteration, i.e., the right join input, to the outer iteration. Accordingly, Rule 6 is called *left-join lifting*.

Rule 6 Left-join lifting.

$$\frac{\begin{array}{c}\texttt{LeftJoin}_{=} \Rightarrow Op_j \qquad \texttt{Join}_{\Theta} \Rightarrow Op_{j2} \\ \texttt{End}(\,(true())) \Rightarrow end_1 \qquad \texttt{End}(\,(true())) \Rightarrow end_2 \\ \Theta \in \{=, <, >, <=, \ldots\}\end{array}}{\begin{array}{c}Op_j(end_1, \texttt{Join}(left, right, end_2), post, out) \Rightarrow \\ Op_j(\texttt{LeftJoin}(left, right, post, end_1), end_2, \texttt{End}, out)\end{array}}$$

Cleaning Up

Left-join lifting leaves the AST with a nesting of two left joins, where the parent is a trivial left join against an empty right input. As last step, this trivial left join is removed from the pipeline with Rule 7.

Rule 7 Trivial left-join removal.

$$\frac{\begin{array}{c}\texttt{LeftJoin}_{=} \Rightarrow Op_j \\ \texttt{End}(\,(true())) \Rightarrow end_1 \qquad \texttt{End}(\,(true())) \Rightarrow end_2\end{array}}{Op_j(Right(end_1), end_2, \texttt{End}, out) \Rightarrow Right(out)}$$

The final AST for the sample query is shown in Figure 5.14. The major difference to the unlifted version is that the result items of the `let`-bound FLWOR expression are now directly computed within the upper-level operator pipeline and not in a separate `pipe` expression. Furthermore, the final AST exploits the join semantics present in the original nested FLWOR efficiently.

5.4. Join Trees

Queries combining data from more than two sources are very common in data management, especially in the context of relational DBMS where frequently several tables are joined via foreign-key references. In the standard bottom-up representation of relational systems, joins over multiple inputs are, as shown in Figure 5.15, usually modeled as binary join trees distinguished as either left-deep, right-deep or irregular, i.e., bushy.

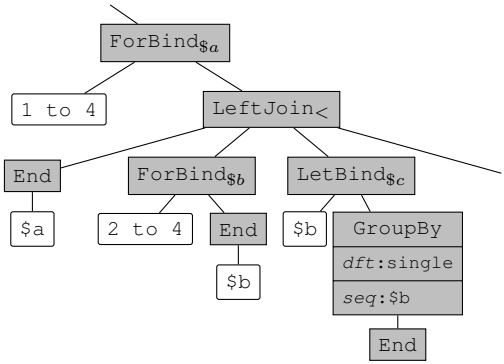


Figure 5.14.: Final pipeline after removal of trivial left join.

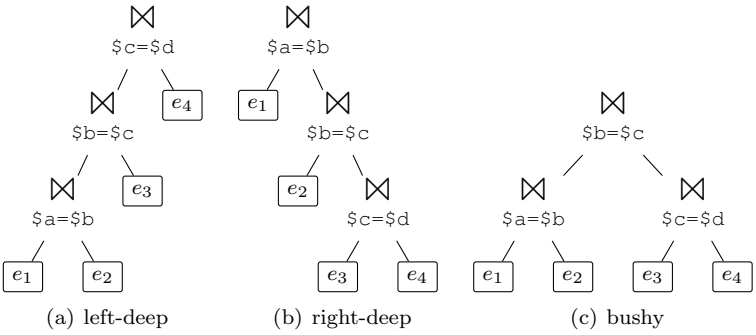


Figure 5.15.: Bottom-up style join trees.

Finding the optimal arrangement of joins is a multi-faceted challenge since the late 1970's [SAC+79]. It is dependent on various factors such as input cardinalities, join selectivity, I/O, parallelism, etc. In the following, we present basic rewriting rules for transforming linear join sequences into nested forms, which reflect bushy bottom-up join trees. With these rules, advanced join ordering algorithms can be transfered to the top-down representation.

To get started, consider the XQuery given in Figure 5.16. It binds 4 inputs successively to `for`-bound variables `$a`, `$b`, `$c`, and `$d`, respectively and filters the resulting tuple stream with join-like equality predicates. Clearly, the $\mathcal{O}(n^4)$ complexity of the näive nested loops cry for efficient join support. For the purpose of demonstration, we assume again that the expressions e_1, e_2, e_3, and e_4 are independent of variable bindings. Note that we also ignore in the following that this particular query actually searches for matches where `$a=$b=$c=$d`.

```
for $a in e1
for $b in e2
for $c in e3
for $d in e4
where $a=$b
  and $b=$c
  and $c=$d
return e5
```

Figure 5.16.: Join cascade of `for`-bound inputs.

After splitting the predicate into multiple `where` clauses and applying predicate pullup and join rewriting, we obtain the typical right-deep operator pipeline shown in Figure 5.17. Initially, the left join input branch is always the empty pipeline, i.e., each join actually joins the right join input with the (shared) common input. In the bottom-up view, this effectively reflects the join order of a left-deep join tree.

For conversion into a bushy join tree, the `for`-bound input sources must be moved to the independent left and right join input branches of joins. This is achieved in two steps. The first step is similar to the join input pushdown Rule 3. For two consecutive joins, Rule 8 pushes the upper join down to the left join input branch of the lower join as shown

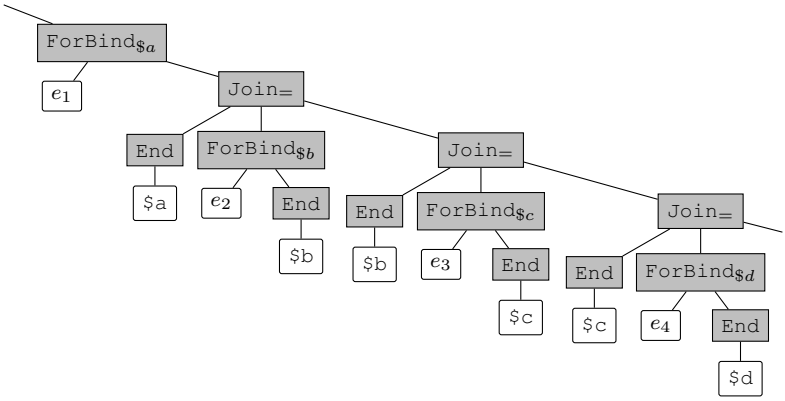


Figure 5.17.: Pipeline with cascade of join operators.

in Figure 5.18. Of course, the right join input must not be dependent on variables bound by the parent. Like the general join input pushdown rule, this rewriting does actually only change the size of the left join input and the number of evaluations of the second join. Accordingly, the join order remains the same as before.

Rule 8 Input join pushdown.

$$\frac{\begin{array}{c}\$x_1,\ldots,\$x_n \Rightarrow \$X \qquad \$v_1,\ldots,\$v_m \Rightarrow \$V \\ \$y_1,\ldots,\$y_l \Rightarrow \$Y \qquad \$w_1,\ldots,\$w_k \Rightarrow \$W \\ \texttt{Join}_{\Theta\$X:\$V} \Rightarrow Op_{j1} \qquad \texttt{Join}_{\Theta\$Y:\$W} \Rightarrow Op_{j2} \\ right_2 \text{ independent of } \$X \\ \Theta \in \{=,<,>,<=,\ldots\}\end{array}}{\begin{array}{c}Op_{j1}(left, right, Op_{j2}(left_1, right_2, out)) \Rightarrow \\ Op_{j2}(Op_{j1}(left, right, left_1), right_2, out)\end{array}}$$

Rule 9 applies the symmetric idea to the output of two consecutive joins and pulls a join into the right join input of its parent. It is illustrated in Figure 5.19. Note, in contrast to Rule 8, join order now has changed because the inputs e_3 and e_4 are joined first.

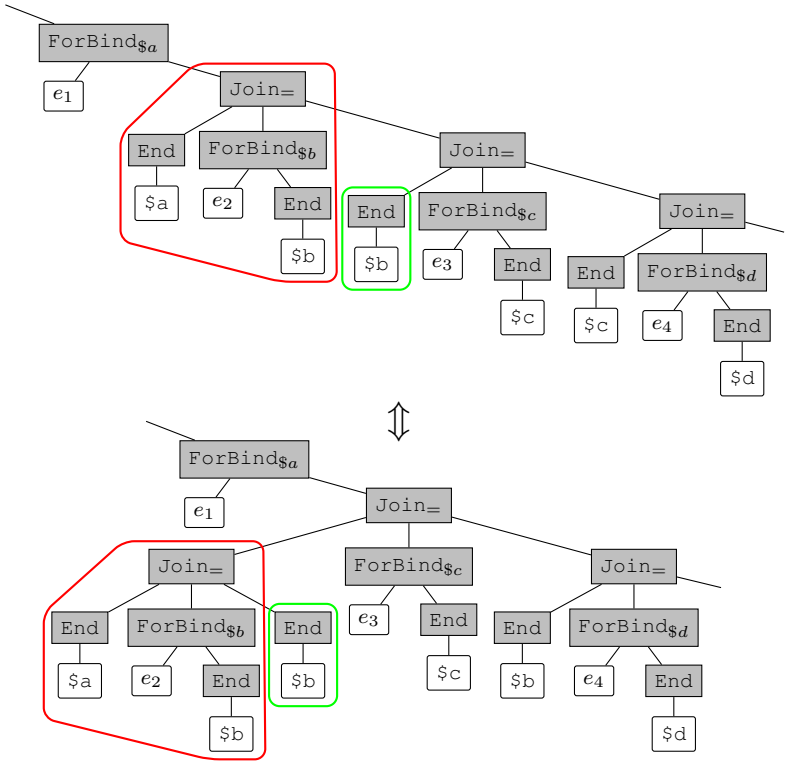


Figure 5.18.: First step of join tree rewriting.

Rule 9 Output join pullup.

$$\frac{\begin{array}{c}\$x_1,\ldots,\$x_n \Rightarrow \$X \qquad \$v_1,\ldots,\$v_m \Rightarrow \$V \\ \$y_1,\ldots,\$y_l \Rightarrow \$Y \qquad \$w_1,\ldots,\$w_k \Rightarrow \$W \\ \texttt{Join}_{\Theta\$X:\$V} \Rightarrow Op_{j1} \qquad \texttt{Join}_{\Theta\$Y:\$W} \Rightarrow Op_{j2} \qquad \texttt{End}(k) \Rightarrow end \\ \$x_1,\ldots,\$x_i \text{ bound by } left,\ \$x_{i+1},\ldots,\$x_n \text{ bound by } right \\ left_2 \text{ independent of } \$x_1,\ldots,\$x_i \\ right_2 \text{ independent of } \$x_1,\ldots,\$x_n \\ \Theta \in \{=,<,>,<=,\ldots\}\end{array}}{\begin{array}{c}Op_{j1}(left, Right(end), Op_{j2}(left_2, right_2, out)) \Rightarrow \\ Op_{j1}(left, Op_{j2}(Right(left_2), right_2, end), out)\end{array}}$$

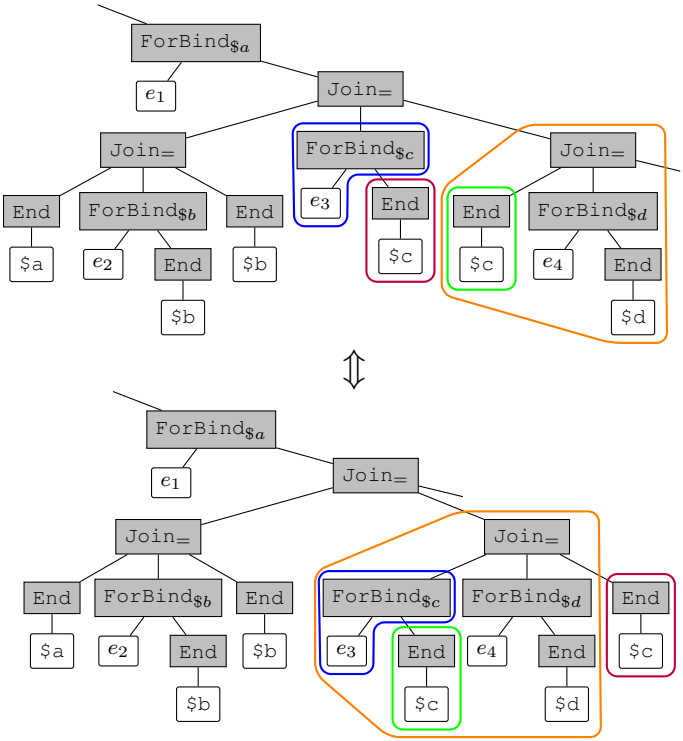


Figure 5.19.: Second step of join tree rewriting.

Finally, by applying Rule 3 twice, the AST reflects a fully balanced join tree as shown in Figure 5.20.

In summary, the rules 8 and 9 provide basic mechanisms for rearranging the order precedence of multiple joins. But again, their benefit is highly dependent on data characteristics and system properties and requires statistics, which are not considered in this work.

Join input swapping can also greatly improve query performance. However, it must be applied only if the output order may be changed, too. If this is the case, the rewriting is trivial because two join input branches are guaranteed to be independent.

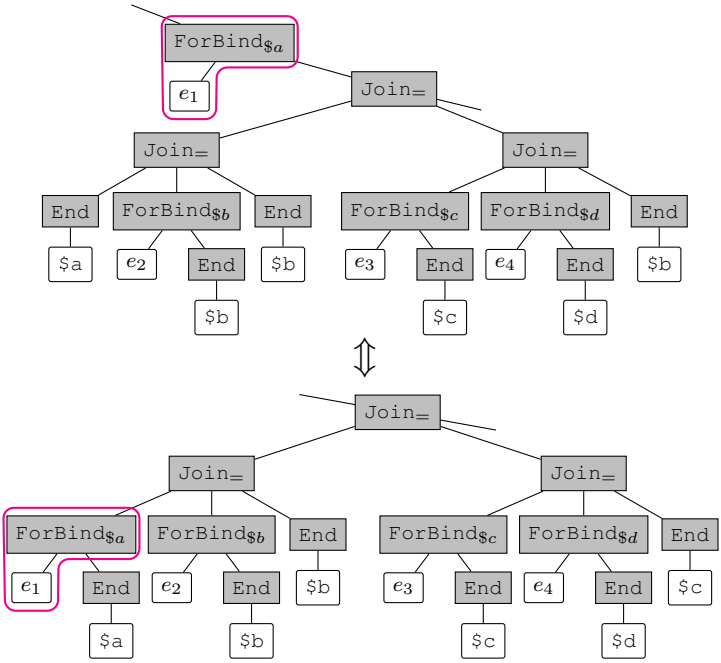


Figure 5.20.: Third step of join tree rewriting.

5.5. Aggregation

The default semantics of grouping clauses is regularly a source of severe performance problems. Already queries over moderate data volumes are likely to exhaust available main memory, because the values of non-grouping variables are simply concatenated to sequences. Furthermore, analytical workloads frequently compute standard aggregates on these sequences manually afterwards, which results in additional overhead. Direct integration of tailored aggregation schemes in `GroupBy` operators is therefore an important aspect.

Figure 5.21 exemplifies a typical situation. A tuple stream with several variable bindings is grouped by one variable and the aggregated sequences of non-grouping variables are reduced afterwards by the func-

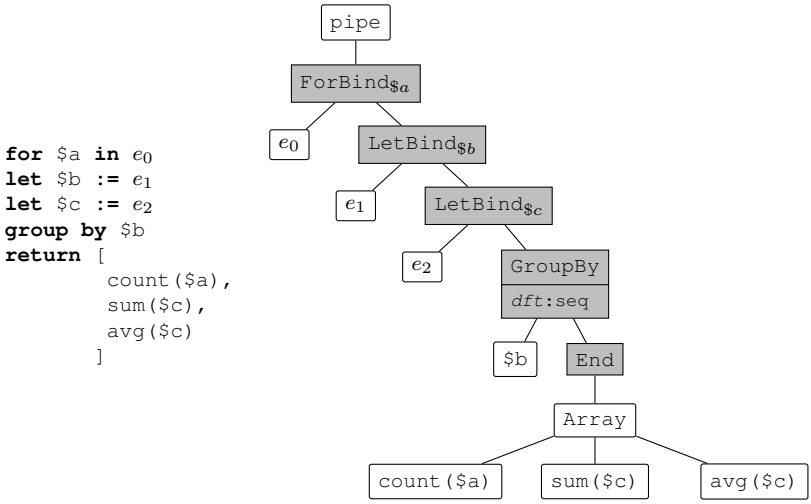


Figure 5.21.: Default aggregation of `GroupBy`.

tions `count()`, `sum()`, and `avg()`. Remarkably, `$c` is reduced twice, i.e., the potentially huge aggregated sequence must be processed by both `sum()` and `avg()`.

The optimizer takes here advantage of the flexible specification of column aggregate types introduced in Section 5.3.1. Instead of aggregating all non-grouping variables to sequences, it changes the default aggregation type from `seq` to the cheaper `single` and introduces manual overrides for those columns, which are actually used after the grouping step. Thanks to the hierarchical AST, all usages of non-grouping variables can be found with a simple scan of the operator output subtree.

Note that changing the default aggregation type from `seq` to `single` is not essential for this optimization. Alternatively, unnecessary grouping overhead can be completely avoided by dropping all non-grouping variables which are not referenced after the grouping.

If a variable like `$a` in the example is referenced as argument of a supported aggregation function (here: `count()`), a respective *aggregate variable* (e.g., $\texttt{\$a}_{cnt}$) is introduced as additional output binding to the `GroupBy` operator, which is computed space and time efficient during

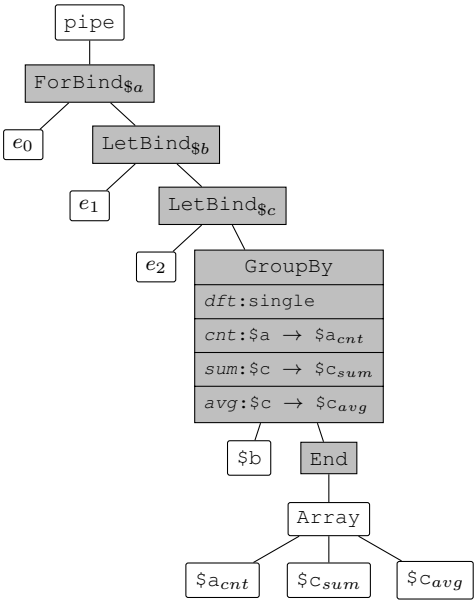


Figure 5.22.: Optimized aggregation in `GroupBy`.

the grouping phase. The referring aggregate function is replaced by a reference to the new variable. If a variable is not used in an aggregation function, but in an arbitrary expression, the default grouping behavior is restored with a respective binding for the aggregation type `seq`.

The demand-driven binding of variable aggregates can be applied to a single non-grouping variable more than once. In the optimized `GroupBy` of Figure 5.22, two aggregate variables, $\$c_{sum}$ and $\$c_{avg}$, were introduced for variable `$c`.

6. Data Access Optimization

Despite careful data flow optimization within operator pipelines, query performance will not be competitive if the compiler cannot exploit the physical capabilities of the target platform. Hence, storage-specific compilation has been a major concern for the design of the compiler framework. In fact, it is one of its biggest strengths. Almost every aspect is open for platform-optimized code, because the compiler does not presume a specific data layout.

The effectiveness of optimizations is necessarily tightly coupled with the concrete system. Some platforms will profit more from tailored data access operations than others, but typical use cases, query patterns, and shapes of semi-structured data suggest storage designs with comparable capabilities and predictable performance profiles. The presented strategies for exploiting platform-specific capabilities are therefore justified by referring to properties and performance characteristics of relational DBMS, native XML-DBMS, and common implementation strategies.

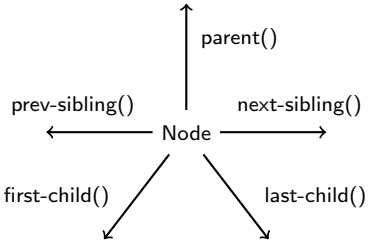
6.1. Generic Data Access

Like any portable architecture, our compiler depends on generic default implementations for all data access operations. As explained in Sections 2.2 and 2.1.2, the basic composition types array and map serve as building blocks upon which a concrete front-end language can define its own abstractions of data including respective types, constructors and access operations. Especially with regard to the latter, data abstractions can and should be picked up by the compiler to support efficient operations. Therefore, we must employ wrapper and adapter techniques to abstract from the physical data organization and allow to plug-in virtually any kind of storage[1] and data source into the system. In the

[1] The term *storage* refers in this context to any kind of disk-resident or memory-resident data structure without any implications regarding database-specific functionality like persistence or transactions.

```
type Node {
  QName name()
  Atomic value()
  Type type()
  Node parent()
  Node first-child()
  Node last-child()
  Node prev-sibling()
  Node next-sibling()
  ...
}
```

(a) Node interface



(b) Elementary navigation

Figure 6.1.: Operations of XML node interface.

concrete case of XQuery, the compiler will rely on an interface-based design for abstracting from XDM data types such as sequence, item, and node.

Adapter interfaces must provide all basic operations for implementing data-specific query logic in a storage-agnostic manner. XPath expressions, for example, are typically processed as iterative traversals of the XML tree. Accordingly, the interface of an XML node will at least need to expose DOM-like operations as depicted in Figure 6.1 for navigating the tree structure. Note that without considering efficiency at this point, every storage will be able to support an adapter for these simple operations. Accordingly, we can safely assume that primitive navigation will work as default option in every system and, therefore, do not go into further details [BBB+09]. More interesting is how we can cope with the drawback of hiding all opportunities for use of efficient native operations behind an interface.

The natural performance penalty of a wrapper-based approach depends on three factors, which will vary between different storages. The main source of inefficiency is the overhead of translating operations between different abstractions. For example, consider the interface operation *next-sibling()*, which returns the right sibling of an XML node in the tree. It will translate to a cheap pointer dereference operation

for a linked tree structure in main memory, but may translate to an expensive scan if the XML tree is stored externally in a relational table.

The second source of inefficiency is indirectly caused by a mismatch in the granularity of operations exposed by the wrapper and supported by the storage layer. In the example above, the navigation step matched the lightweight pointer structure of the in-memory tree, but was too fine-grained to justify a bulk-oriented scan operation. For a path expression `//a/b/c`, however, a scan operation might be adequate to retrieve all matching nodes at once. That aspect is not necessarily symmetric. While node-wise tree traversal is usually prohibitively expensive on external storage, navigation in main memory is still quite efficient.

The third source of inefficiency results from poor consideration of locality effects. The order and the volumes in which data is accessed by a query will considerably affect on performance. Again, this is particularly true for data on external storage, but even operations on in-memory data will be penalized by poor locality.

Data locality is especially difficult to address in a portable design because it is a physical property of the system. However, the assumption that logically related data, e.g., all nodes in an XML fragment, is also physically clustered, is generally a good heuristics. However, the nodes of a linked XML tree, e.g., might also be randomly scattered in memory.

To summarize, we can identify the following key challenges:

Access Granularity The granularity of data accesses performed by a query should be aligned to the capabilities and performance characteristics of the underlying storage.

Access Economy The total number of data accesses should be minimized because every operation will incur some overhead.

Access Locality The locality of operations should be optimized with respect to the physical layout of the data.

In the following, we will investigate common query situations and show how physical data access can be optimized with respect to the above challenges. The optimization strategies pursued will rely on the following principles:

Simplify Complex data access sequences (e.g., path expressions) should be simplified to equivalent but cheaper operations.

Batch Groups and sequences of related data accesses should always be mapped to coarse-grained alternatives.

Short-circuit Superfluous checks and additional operations for special cases should be short-circuited whenever possible.

Choose The cheapest alternative should be chosen if there are several ways to realize an operation or a sequence of operations (e.g., by the use of an index).

6.2. Storage-specific Data Access

The largest fraction of data access operations belongs to navigation routines, which match structural patterns against complex values. The pattern matching itself is a complex process that makes heavy use of relatively fine-grained operations. Accordingly, there is a huge saving potential if navigation routines are shortened.

Consider the evaluation of the single-step path expression `./item` through the basic adapter interface. To find the target elements, the naïve navigation algorithm will iterate over the children of the context node by repeatedly calling *next-sibling()* and perform the matching by calling *name()*, and *kind()* for each node. Executing the same logic directly on the storage is likely to be much faster. This optimization can be seen as some sort of predicate pushdown.

6.2.1. Native Operations

In the first place, native navigation routines dramatically reduce the code path for the matching loop. Additionally, most storage implementations will be able to leverage additional knowledge about the data, which is not available to the generic runtime. For example, a storage might know the location of child nodes with a particular name or that only one child node will qualify at most. This information can be used by the storage to guide the search or to stop as soon as the final result is fixed.

```
type Node {
  ...
  Sequence child-elements(QName name)
  Sequence desc-elements(QName name)
  ...
}
```

Figure 6.2.: Supplementary operations of XML node interface.

The performance gain achieved by offloading query logic to the storage varies not only between different systems but also between different data sets. For example, if the navigated XML subtree is very small, the improvement will also be rather small, but large instances or huge document collections can profit a lot.

Enriched Adapter Interfaces

For frequent query patterns, it is advisable to extend the adapter interface with operations, which map the query logic directly to operations on the storage. XPath-based navigation of XML trees, for example, suggests dedicated navigation operations for the important `child` and `descendant` axes as shown in Figure 6.2. Compiling step expressions to make use of these operations is trivial.

The disadvantage is that storage interfaces are bloated up and that similar logic must be implemented in each adapter. However, it is also feasible to treat the implementation of supplementary operations as optional and automatically fall back on standard navigation routines.

At this point, it should be emphasized that respective optimizations are not restricted to XML data. In XQuery, the sequence interface will, for example, profit from operations for positional access to support filter operations like `$myseq[last()]`. Front-end languages for other data models will similarly suggest useful operations for supporting query logic.

Direct Embedding

The alternative to the extension of adapter interfaces is a mapping of operations and sequences of operations to custom expressions that compile

directly to native operations of the respective storage. The corresponding AST rewriting logic, the implementations of the new expression types, and everything else necessary can be conveniently packaged in a storage-specific compiler module. The efficiency of native processing can so be leveraged without tampering other storage modules.

The rewriting of expressions to native operations requires the ability to confine data access operations at compile time to a particular storage. In case of a path expression `$a/b/c`, for example, the compiler must be able to trace back the variable reference `$a` to the respective data source, which is typically a function call like `fn:doc('data.xml')`. Due to the hierarchical scopes in the AST, the analysis is usually simple, but most queries will access data from a single storage anyway.

However, if a query accesses multiple data sources, there may arise constellations where it is not possible to statically confine operations to a particular storage. In this case, one has solely the chance to dynamically make the distinction and dispatch the operation to the respective storage. But depending on the complexity and the frequency of the operation, the additional runtime overhead can dominate the gains.

Data-specific Simplification

If data source analysis is able to deliver additional information about the data queried, even more powerful rewritings can be performed. Especially the processing of structured data will benefit from the availability of schema information, because navigation steps reduce to structured field accesses rather than structural pattern matching.

As example, consider a database of customer records. The logical and the physical representation of a `Customer` record may be completely different. Depending on the abstraction principles of the front-end language, it can be logically represented, e.g., as a JSON object or an XML fragment, but physically stored in a relational table, a text file or a structured value in main memory. For accessing a component of a record, the storage logic will need to perform a lookup, e.g. in a hash table, to map the field name to the corresponding array field or offset. For large data sets this overhead accumulates and costs performance. With schema-information, however, lookups can be performed once at compile time or at the beginning of the query, so that native operations can access the fields directly.

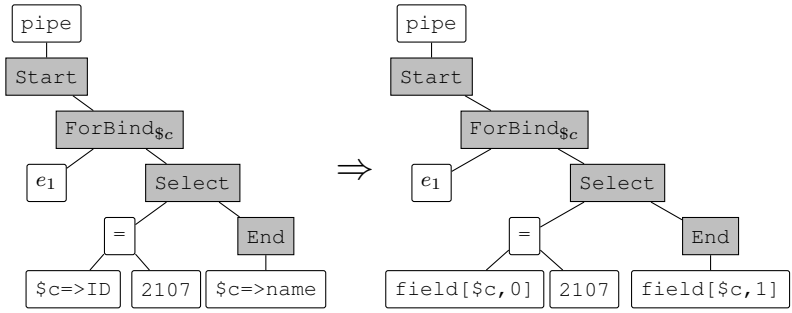


Figure 6.3.: Pre-compilation of navigational access for structured data.

```
$person/age > 21
```
(a) Comparisons

```
<user name="{$u/name}"/>
```
(b) Constructors

```
fn:max($p/price)
```
(c) Arguments

Figure 6.4.: Common examples for value coercion.

Figure 6.3 illustrates the pre-compilation of field accesses for a structured `Customer` record, rendered in the query as a JSON object. The string-based map lookups have been replaced by storage-specific expressions, which directly access the field values through positional access to the underlying array structure.

6.2.2. Eager Value Coercion

The importance of value coercion for processing semi-structured data is regularly underestimated. As the examples in Figure 6.4 show, it is an important aspect of many frequent and therefore performance-critical operations.

In the context of operations, which coerce arguments respectively subexpressions, it is beneficial to perform the coercion eagerly as part of the navigation routines, which yield the respective values. It shortens the code path, effectively reduces expensive data access, and prevents the creation of temporary objects. Note that eager coercion can be realized with both extended adapter interfaces and native operations.

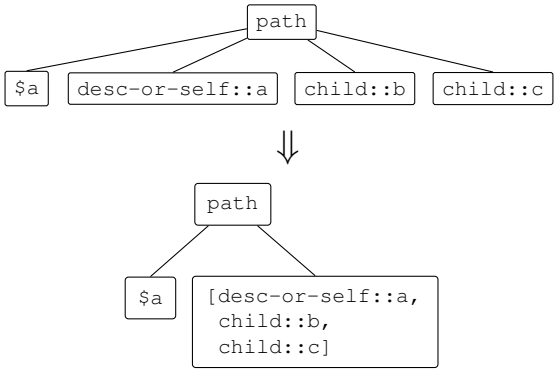


Figure 6.5.: Generalization of axis steps to a complex path step.

6.2.3. Path Processing

The generalization of individual navigation steps to a single multi-step path operation can deliver performance gains in the order of magnitudes. The coarser operation granule comes with less call overhead and has advantages in terms of code and data locality. Furthermore, efficient evaluation of path expressions is a key concern in most XML storages, which is why substantial performance improvements may be expected.

If the storage offers direct support for multi-step path operations, the compiler needs only a simple rewriting to merge individual step expressions as depicted in Figure 6.5. Note that this form of rewriting is also the reason, why we do not desugarize path expressions to FLWORs beforehand. It is much easier and more efficient to compile the original path expression with all steps at once instead of breaking them up into nested FLWORs and putting the scattered pieces later together again.

As explained in Section 3.2.3, XPath requires the result of a path expression to be free of duplicates and sorted in document order. Therefore, the canonical FLWOR-based evaluation of path expressions employs a post-processing step, e.g., a utility function `fs:ddo()`, for sorting and duplicate removal. However, expensive post-processing is in many cases not necessary, because navigation algorithms delivering unsorted results are uncommon and because duplicates can appear only under certain circumstances. Whether or not a path expression will de-

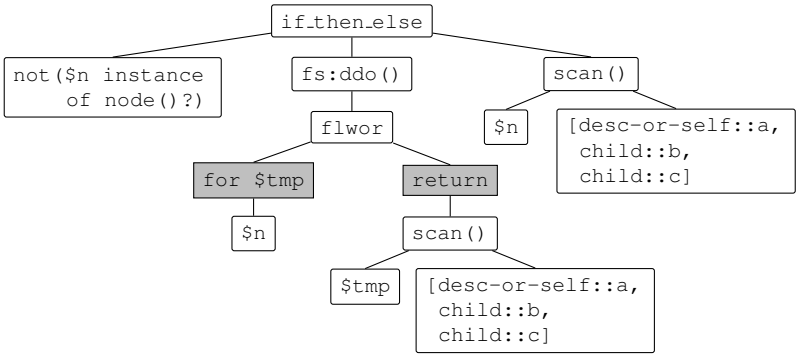


Figure 6.6.: Native path processing with runtime checks.

liver sorted and duplicate-free results can often be determined statically, but it is extremely complex for non-trivial paths [FHM+05].

Holistic path processing routines at the storage level also guarantee sorted and duplicate free results. For example, a popular evaluation strategy for paths in XML stores are localized subtree scans, which use special stack-based algorithms [BKS02] or metadata structures [GW97, MHSB12] to match a path against nodes streaming in. Hence, in addition to general efficiency gains, this form of evaluating multi-step paths on the storage relieves the runtime from expensive query analysis and post-processing overhead. Additional care is only required in situations where the results from multiple path evaluations have to be merged, i.e., if the path matching routine is called for several context items within the same path expression. In this case, the compiler has to generate guarded code with runtime checks as exemplified in Figure 6.6.

In practice, the pushdown of path expressions to the storage has limitations. Most storage implementations do not support paths where intermediate steps have additional predicates like in `$a/b[.x = 5]/c`. At most only a few systems are able to evaluate a restricted form of such patterns (see Section 6.3.1). Similarly, many storages do not offer extended support for uncommon axes like `previous-sibling`, `following`, etc. However, the dominating axes `child`, `descendant`, `descendant-or-self`, and `attribute` will be efficiently supported in most storage designs.

6.3. Bulk Processing

The importance of efficient bulk processing requires to look at data access operations particularly within the context of operator pipelines. Here we find potential for optimizing binding operations and data source accesses in general, but also opportunities for improving access locality across several pipeline stages. Our primary goal will be again to identify certain access patterns in order to map large parts of the query directly to native operations.

6.3.1. Twig Patterns

The simplest and, at the same time, one of the most frequent situations that we can exploit is a byproduct of the predicate-pullup rewriting shown in Section 5.1. It intended the reduction of tuples flowing through the pipeline as early as possible. The `Select` filter is floated up the pipeline to the earliest possible point, which is usually directly below the binding operator on which the predicate depends. The idea is now to combine the binding operator and the filter to avoid the creation binding tuples, which would otherwise be immediately discarded by the following filter. Furthermore, we want to exploit data locality by pushing predicate evaluation down to the storage.

Achieving the first goal is trivial. The binding expression and the predicate are combined to a filtered binding expression and the superfluous filter operator is removed. The second goal is not a self-runner, because the resulting filter expression can be of arbitrary kind and shape. However, the way structured and semi-structured data is processed will frequently lead to a certain access pattern, which we can exploit.

Pipelines are used to iterate over and process collections of data. Hence, they always start with one or several binding operators, which feed the data into the pipeline. They bind "data items" of a certain granule or super-structure, and the subsequent pipeline operators navigate within these logical granules to extract further information, e.g., for filtering. Accordingly, we can expect to frequently find the situation that the filter predicate navigates into the tree structure of the data item to be bound.

This form of filtering by a structural pattern is called *twig pattern matching*, because the downward paths reflect a tree-like structural pat-

tern that is matched against the data. The frequency and cost of twig pattern matching makes native storage support advantageous. Hence, the topic gained a lot of attention in research and efficient algorithms for different kinds of storages were developed [Mat09].

Simple but frequent twigs with a single branch and an optional content predicate like in `$products//product[./name]` are efficiently supported by almost every XML storage. Most designs support even more advanced algorithms for matching complex twigs with multiple branches and arbitrary sequences of child and descendant axes. As a result, many storages will allow to compile filtered binding expressions to native twig operations.

The capabilities and limitations of the various proposals are manifold so that we can here refer only to the literature. It should be emphasized, however, that the applicability of twig matching routines is easy to test, because twig patterns will appear in the pipeline respectively the AST very clearly. Again, this is owned to the fact that path expressions are not normalized to FLWORs.

6.3.2. Multi-bind Operator

Twig patterns appear in more flavors than simple combinations of binding operators and filters. Merely, the mentioned downward path matching against structured and semi-structured data items is a central aspect in most queries. Variable-bound items are navigated at many points for filtering, aggregation, result projection, etc.

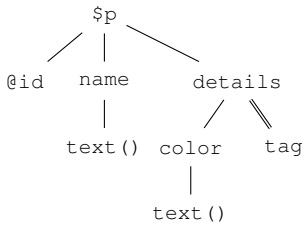
Consider the example given in Figure 6.7(a). From a bird's eye view, one can see how different parts of the same logical abstraction, a product item, are used. Aside in the filter condition, the `for`-bound variable is used multiple times as starting point for paths in the final result construction. The tree structure of the corresponding twig pattern is shown in Figure 6.7(b).

From the physical perspective, such a twig pattern can be processed efficiently if all branches are matched in a single operation, e.g., by fetching all parts of interest in a single subtree scan. Within a `for` loop, we achieve efficiency through access locality and by reducing the number of expensive I/O for disk-based storages.

Although the clauses of a FLWOR imply a logical evaluation order, the actual evaluation order can be different, e.g., because of lazy eval-

```
for $p in e1
where $p/@id = 47261
return
  <product>
   <name>
    {$p/name/text()}
   </name>
   <color>
     {$p/details/color/text()}
   </color>
   <tags>
     {$p/details//tag}
   </tags>
  </product>
```

(a) Query



(b) Twig pattern

Figure 6.7.: Twig pattern within a pipeline loop.

uation or rewritings performed by the compiler. Efficiency through physical and temporal locality is therefore not guaranteed. Surely, the compiler can never give such guarantees, because it will not be in control of the storage. However, it can perform rewritings, which allow to put the storage in charge of an efficient evaluation.

The strategy pursued is explained best with a running example for the query of Figure 6.7. First, all twig paths are extracted from their context by binding them to `let` variables as shown in Figure 6.8. Note, how simple the extraction is with the hierarchical AST. We only need to pull-out nested path expressions starting at the loop variable.

In the next step, all twig bindings are moved up to the respective `ForBind` operator. This is allowed because a `LetBind` reflects a nested loop with only a single iteration and exchanging such loops will not influence output order. In this sense, it is very similar to predicate pull-up. The resulting pipeline is shown in Figure 6.9.

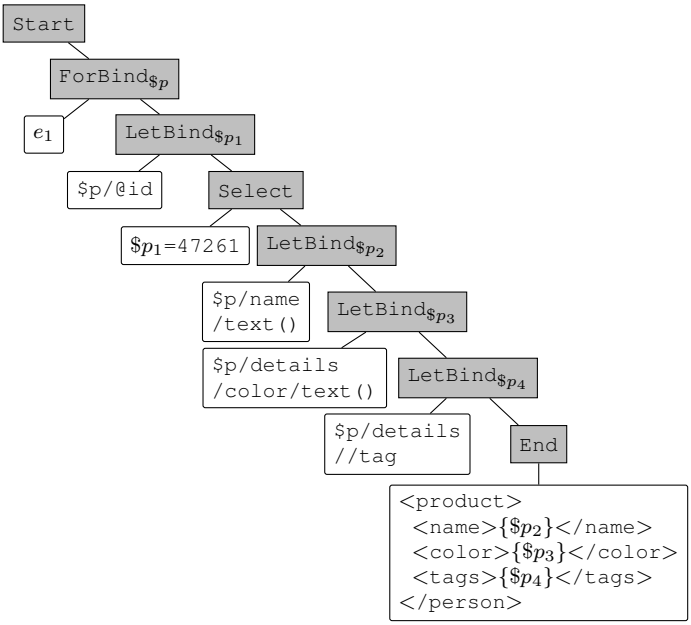


Figure 6.8.: Pipeline after extraction of twig branches.

The last step is the key. The initial binding expression, all filter twig branches, and all binding twig branches are merged to a single twig operation, which can be pushed down to the native storage.

At this point, it should be emphasized that twig algorithms are not limited to match branches for filtering only. Matched twig branches can also be returned as part of the result. The result of a twig operation is then a sequence of twig vectors, of which each consists of the root node of the matched structure and all nodes matched at the twig output branches.

For binding the twig result vector, i.e., the loop variable and the additional output paths, the `ForBind` is generalized respectively[2] – a feasible step, because the underlying `Bind` operator offers the possibility of binding multiple variables anyway. The resulting AST is shown in Figure 6.10.

[2]Alternatively, the twig output vector could also be bound as array value.

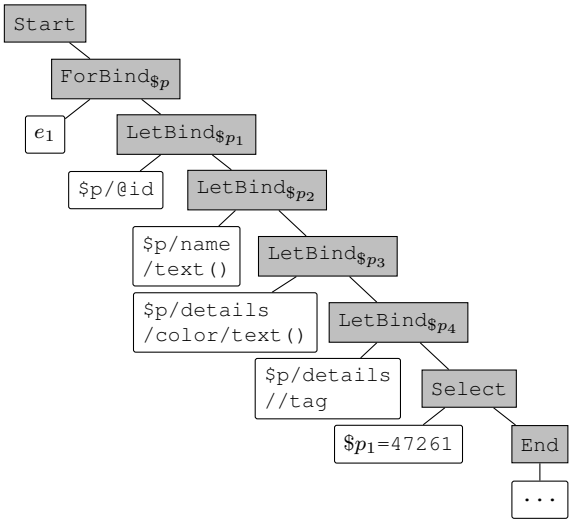


Figure 6.9.: Pipeline after pull-up of twig branches.

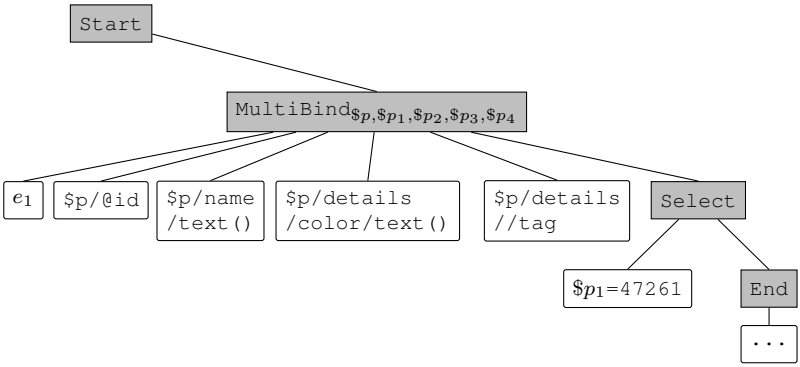


Figure 6.10.: Final pipeline with twig vector binding.

Treating all data matched by the twig pattern as a unit, we can look at the operation as a sort of projection for semi-structured data. Accordingly, twig output paths should be materialized eagerly to leverage locality benefits of twig matching. For example, the element constructor in the `return` clause copies all subtrees of the nodes bound to $\$p_4$ into the result. Therefore, it will not be sufficient to bind solely the single twig output nodes to variable $\$p_4$. The constructor would need to access the storage again to reconstruct each subtree. Instead, the twig should materialize small fragments, thus, saving the need for additional storage access. Similarly, the twig algorithm should perform coercion if necessary, too.

If statistics are not available at compile time, the decision whether or not eager materialization is appropriate should be made dynamically, e.g., if the substructure is rather small. Thereby it should be kept in mind that the additional bindings performed by this optimization already increase the memory footprint of pipeline tuples, which may have a considerable effect on the memory requirements of blocking operators.

6.3.3. Indexes

In addition to primary storage, database systems as target platform offer indexes as secondary access path for fast data retrieval. In this area, we can again leverage from extensive research, especially in the context of XML.

In contrast to relational systems, which specify indexes on column values, semi-structured data can be indexed by three classes of indexes [MHSB12]:

Value Indexes are the simplest form and map atomic values to a corresponding complex value, e.g., the text value of an XML attribute to the respective attribute node.

Path Indexes contain references to structures on a certain path. For example, a path index for the pattern `//product/name` will contain references to all `name` elements in a document collection, which are children of `product` elements.

Content-and-Structure Indexes (*CAS*) are a combination of the latter and enable value-based retrieval of structures on certain paths.

> For example, a CAS index for `//product/name[xs:string]` will allow to search for product names by string.

A specialty of all three index types is that they are scoped to specific types of values and structures on specific paths. In contrast, a relational index on a table contains an entry for each row.

From the compiler point of view, indexes yield a promising alternative for evaluating various kinds of path expressions and content predicates. Particularly interesting is the use of indexes for evaluating filtered binding expressions. For example, whenever the compiler detects a `for`-bound path expression, which is probably also augmented with an additional twig-like predicate as introduced in Section 6.3.1, it will try to find appropriate indexes for evaluating the binding expression. Especially highly selective twig patterns, which are matched against the whole document or collection (e.g., the path starts with `fn:doc()`), will benefit from the speed of focused data access through a single index or a combination of several indexes.

As always, the crucial step at compilation time is the analysis which of the indexes available are applicable and which combination of indexes is best. The index selection problem is known to be NP-hard, so we need to fall back on heuristics and, if available, statistics. Interestingly, an index can even be used if its scope does not fully match the twig pattern, i.e., if it is more general and delivers only candidate matches. In this case, the index is used to partially evaluate the twig pattern and the rest is evaluated on the primary storage. The other way around, i.e., the use of an index with a narrow scope for a more general query pattern, is possible if additional metadata is available, which guarantees that qualifying results will not be missed.

The actual rewriting and compilation steps are identical to normal path rewriting. Hence, we can refer again to [MHSB12] for details and examples on index matching and selection.

7. Parallel Operator Model

Even the most advanced compilation techniques stay without effect if the runtime cannot deliver expected performance. Query performance, in particular, stands and falls with the efficiency of bulk operations. In general, the classic pull-based model with the open-next-close protocol is a good choice for implementing operators. It is simple, efficient and most query algorithms are streamlined to it. However, current trends in multi-core and many-core architectures require new designs, which naturally embrace parallelism and utilize available resources. Such a design requires careful balancing of various aspects. One must not only identify, which parts of a query can be processed in parallel, but must also find a good balance of parallelism and overhead for scheduling and synchronization.

Achieving high parallelism is particularly difficult if the system is not tied to a particular data storage, because one cannot expect to be in control of important aspects like data partitioning. Instead, opportunities for effective parallelization must be identified, created, and exploited dynamically at runtime. Existing solutions for parallel processing in the pull-based model are not suitable, because parallelism is statically compiled into the query pipeline and often confined to single operators, e.g., a parallel join algorithm.

In the following, we present a novel push-based operator model, which dynamically partitions data flows and pipeline operators for parallel processing. In tradition of previous chapters, the concept developed assumes very little knowledge about the query and the actual data itself. Parallelization is performed automatically at runtime and adapts itself to the current workload situation. It neither presumes the availability of cost functions and statistics for the query plan nor does it depend on input of a specific kind, size, or shape. But if desired, the presented solution can be enriched with compile-time or runtime knowledge to improve performance.

7.1. Speedup vs. Scaleup

The goal of parallelization is called *speedup*. It is a measure for the performance improvement of a parallel execution in comparison to the serial execution.

The literature knows two definitions of speedup. The classic definition of Amdahl [Amd67] calculates speedup with regard to the faster time to completion of a parallel program. The alternative definition of Gustafson [Gus88] calculates speedup as the capability to use parallelism for compensating scaling of the problem size. In some contexts, it is therefore also called *scaleup* [DG92].

Which of the two definitions applies best depends on the concrete use case. Some scenarios require parallelism to reduce the time spent for a task, e.g., to improve the response time of a server, whereas others require parallelism to scale a system to larger data volumes. Interestingly, the two definitions must not necessarily lead to completely different systems. In many cases, a better parallel design will yield improvements with respect to both definitions.

Independent of the parallelization goal, all efforts will only pay off if a reasonably large part of the work can benefit from it. Accordingly, a high degree of parallelism must be established as fast as possible and also kept alive as long as possible. The concrete challenges that need to be addressed here are manifold. Sequential operations and I/O can turn out as bottlenecks as well as various forms of resource contention and load thrashing. Furthermore, it is generally difficult to distribute operations on semi-structured data.

In this setting, optimal parallel execution of a query is almost impossible to guarantee. Therefore, the execution framework presented pursues the philosophy to create at least *opportunities* for parallelism between operators and within operators. Whether they are exploited or not will depend on available resources at runtime. Therefore, speedup and scaleup achieved will be sensitive to various factors, but improve, the more chances for parallelism are taken and the less overhead is imposed by the infrastructure itself.

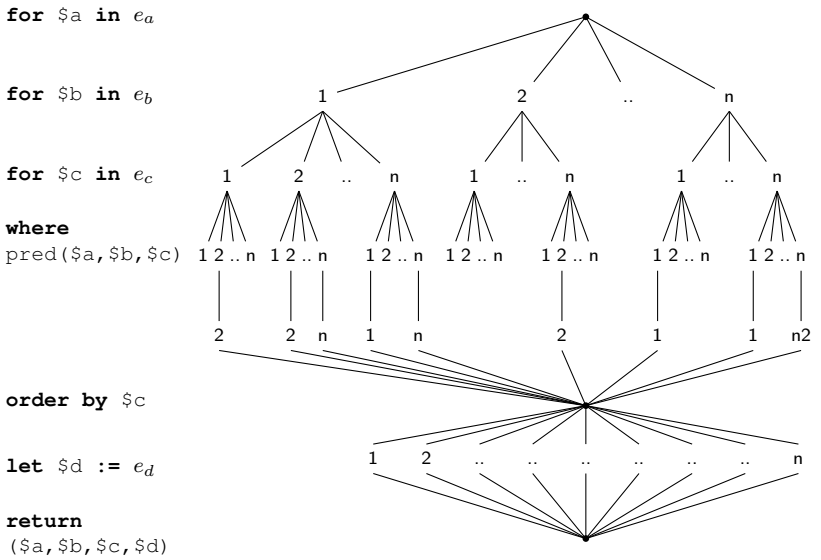


Figure 7.1.: FLWOR iteration tree.

7.2. Parallel Nested Loops

The idea for the parallel operator model builds on the parallelization of nested loops – a characteristic of operator pipelines that we mentioned several times. Figure 7.1 illustrates nested iterations for a typical FLWOR expression with several variable bindings, a `where` clause and an `order by` clause. Starting from a single context tuple which is passed to a `pipe` expression, sequences of `Bind` operators form a tree of nested iteration scopes. The fan-out of a node in the tree is determined by the size of the respective binding expression. In case of `Select` and `LetBind`, the fan-out is at most one, but a `ForBind` may easily reach millions. The operators `OrderBy`, `GroupBy`, and `Count`, as well as the final `End` create the corresponding fan-in and optionally a fan-out.

The parallel execution of nested loops has been extensively studied in the literature. A large part of the research work concentrated on techniques for re-organizing loop constructs and loop nests to eliminate loop-carried *dependences* [WB87]. The presence of dependences implies a full or partial order in which individual iterations of a loop must be executed to yield the correct result. For example, the simplest form of a loop-carried dependence is a counter variable, which is incremented and used in each iteration. In general programming, various other forms of dependences exist and typically require smart analysis and rewriting techniques to peal out exploitable parallelism from a loop nesting [KA02]. Note that dependences are of structural nature only and do not depend on the actual data itself.

Fortunately, operator pipelines are *per se* free of dependences, because data is immutable and variables are assigned only once within a loop. Furthermore, iterations do not share an intermediate state. Thus, operators can be freely parallelized. In the following, we can therefore safely focus on the actual creation and scheduling of parallel work.

Forms of Parallelism

The iteration graph in Figure 7.1 visualizes the two options for creating parallelism in an operator pipeline. *Pipeline parallelism* is achieved by "cutting" the graph horizontally into decoupled processing stages, which run in parallel. In the extreme case, each operator in the sequence is assigned to a separate process[1]. *Data parallelism* is achieved by cutting the graph vertically, i.e., multiple "copies" of each operator are applied in parallel to partitions of the input. Both strategies have individual advantages and drawbacks.

Pipeline parallelism is simple to realize and therefore very popular for parallelizing producer/consumer-like constellations. It is even in pull-based query plans very convenient. One must only introduce artificial exchange operators in the pipeline [Gra94].

On the flip side, pipeline parallelism is very sensitive to load skew. Optimally, the workload is balanced and the processing of all pipeline stages overlaps as much as possible. Practically, however, this is hardly

[1] The term *process* commonly denotes a unit of scheduling and program execution. For the sake of simplicity, we abstract from its concrete realization as operating system process, thread, etc.

the case. For example, binding operators act like multipliers and typically emit more input tuples than they consume, as rendered in Figure 7.1. In a parallel pipeline, this increases the load on the respective following operators.

Another form of workload skew occurs if one stage is more costly than others. The resulting load imbalance quickly disrupts the pipeline flow and results in poor parallelism. This is particularly bad as pipelines already need a warmup time before the first inputs reach later stages. Intermediate blocking operators like `OrderBy` further reduce the benefit of pipeline parallelism.

Last but not least, pipeline parallelism suffers from limited scalability. The maximum degree of parallelism is fixed by the pipeline structure, i.e., the approach cannot scale with the number of available resources.

Given a sufficiently large input, data parallelism scales, at least theoretically, much better, because it distributes the load over all available resources. However, data parallelism is relatively complex to realize and must be explicitly implemented within each operator. Especially, blocking operators must be coded carefully with parallel algorithms and low-overhead data structures.

In practice, the scalability of data parallelism is limited, because data dependencies and inherently serial operations do not allow to decompose programs entirely into data-parallel operations. The need to synchronize access to shared resources is another performance threat. Data-intensive scenarios particularly suffer here from I/O, because external storage is slow and does not well support random access.

Task Granularity

Splitting a query into independent units of work is the first and at the same time the most critical step of parallelization: Fine-grained work units achieve a good load balance, i.e., high parallelism, but cause a high scheduling overhead. Coarse-grained work units impose only minimal overhead, but then the system is likely to suffer from load skew. Translated into the context of parallel nested loops, we need to find a good balance between serial and parallel execution of loop iterations.

With respect to a single loop, the maximum degree of parallelism is achieved by scheduling each individual iteration as a separate, parallel task. However, any larger loop will anyway exceed the number of avail-

able processors by orders of magnitude, so that potential parallelism cannot be exploited. Besides, this approach results in a high overhead per iteration, which only pays off if the loop body is very expensive. Scheduling tasks for whole bunches of iterations in turn reduces the theoretical degree of parallelism, but amortizes overhead.

For optimal parallelism, loop iterations must be evenly distributed among the available processing units. However, an equal number of iterations per task achieves only a good balance if the cost of iterations is equally uniform. If some iterations are considerably more costly than others, the workload is again skewed. Optimal parallelization of loop nestings is especially difficult as it requires to consider the cost and, thus, the fan-out of inner loops, too. Accordingly, smart and fast loop partitioning is the key to high parallel performance.

7.2.1. Data Partitioning

The literature proposes various techniques and partitioning schemes for parallelizing nested loops. Conceptually, they share all similar ideas and schedule loop iterations either block-wise or with a constant stride over the input.

Unfortunately, most ideas cannot be directly applied in the query processing context, because the research of parallel systems focuses on loops over array-organized data in main memory and assumes that loop size is known beforehand and data can be accessed randomly using the loop variable as index. In our case, binding operators loop over sequences, which is a powerful abstraction for data processing, but also poisoned with a scalability problem: it is inherently sequential. The work cannot be naïvely partitioned by simply specifying ranges over a loop variable, because the size of a sequence is often unknown and not all kinds of sequences support efficient random access. Accordingly, effective realization of loop-level parallelism depends on the capabilities of the binding sequence and the runtime must find the best way to partition it as quickly as possible.

If a sequence supports random access, it is best partitioned with a divide-and-conquer strategy, because it creates very quickly a high fan-out, i.e., parallelism as shown at the left side in Figure 7.2. Even better, the recursive partitioning can run in parallel as well. In contrast, sequential binding sequences can only be partitioned step-wise as shown

at the right side of Figure 7.2. It is inferior to the divide-and-conquer scheme in every regard. It takes longer to reach the maximum fan-out and the splitting process itself is inherently sequential.

Surely, both partitioning schemes can also be combined. For example, if the input sequence reflects a scan operation, it can be fetched chunk-wise into memory, where it can be efficiently partitioned with the divide-and-conquer pattern. In the opposite situation, if the input sequence supports random access, but individually-computed chunks must be joined afterwards again in their initial order, e.g., at the `return`, splitting the input asymmetrically in a smaller head and a larger tail chunk reduces the need to buffer already processed results from "later" iterations. Hence, choosing the right partition size is crucial for CPU and memory efficiency.

Lazy binding sequences come with the additional penalty that making use of the sequence is already an expensive operation. Partitioning a lazy sequence can easily take a considerable share of the total processing time of a loop. In the worst case, the partitioning will even take longer than the actual parallel computation of the loop body. However, if the processing of a partition is more costly than the partitioning step, even sequential inputs can benefit from parallel processing.

Under our initial assumption that statistical information about inputs and the query itself is not available or too complex to infer, we cannot expect to find the optimal degree of parallelism at compile time. Therefore, we pursue an optimistic, dynamic variant of parallel loop scheduling. We use relatively small partitions but employ a lightweight scheduling mechanism, which accommodates both the sequential and the divide-and-conquer partitioning scheme and adaptively regulates the number of utilized processes.

7.2.2. Task Scheduling

The complement to input partitioning is the assignment of loop partitions to parallel processes in form of tasks. At the beginning, a query is assigned to a single process, which drives the computation by continuously creating subtasks for parallel execution. However, we do not directly assign each subtask to a parallel process. The system is designed to treat parallelism as *always optional*. It allows to increase or decrease the number of processes assigned to a query at any time.

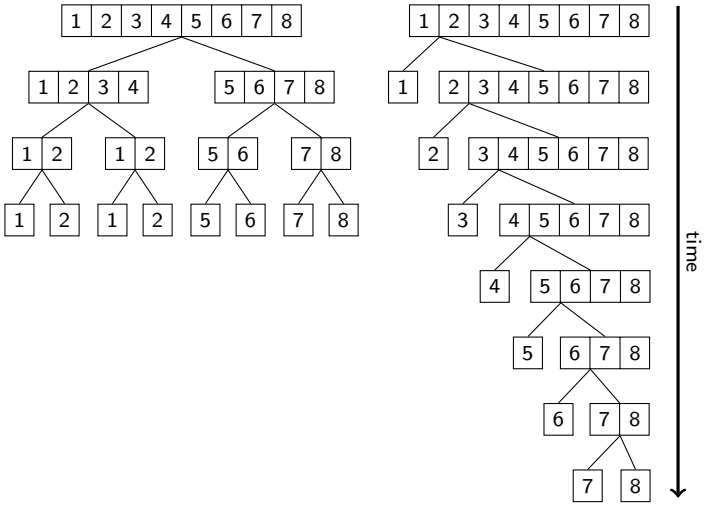


Figure 7.2.: Recursive and sequential input partitioning.

Therefore, the initial process does not only create the tasks, it also executes them. If additional processes are available or become available at runtime, they will take over some tasks and thereby potentially creating new subtasks themselves. If the system needs the helping processes again for other tasks, e.g., for a second query issued by another client, the initial process continues by processing the remaining subtasks itself. This adaptive approach is often referred to as *self-scheduling* [PK87, Pol88, KA02].

For load balancing, self-scheduling algorithms employ a technique called *work stealing*, which has been widely studied and implemented in the context of parallel shared-memory architectures [BL94, FLR98, Rei07, TCBV10]. The key idea of work-stealing schedulers is the use of one or several task queues in which processes place spawned tasks and from which parallel processes "steal". Therefore, lightweight and highly-concurrent queue operations stand in the center of interest of most work-stealing schedulers [HS02, CL05, SLS06].

Numerical and scientific applications, the usual adopters of work-stealing techniques, offer great chances for highly parallel input parti-

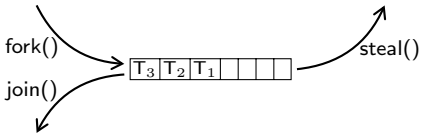


Figure 7.3.: The fork/join task dequeue of worker process.

tioning and result consolidation. Subtasks usually operate in-memory only and have predictable input and output sizes (e.g., matrix multiplication). In query processing, the situation is usually more complex, because the actual output size of individual subtasks may considerably differ (e.g., filter operations) and buffering of large intermediate results may require expensive I/O. Additionally, the scheduling mechanism must often obey (partial) ordering constraints, e.g., to accommodate output order preservation in a pipeline.

Fork/Join Model

The fork/join model is ideal to schedule tasks from inputs, which support efficient random access. The design principle is very popular in parallel programming, because it can be directly applied to any recursive divide-and-conquer algorithm [FLR98, Lea00]. A given task is recursively divided into smaller subtasks until it is small enough for sequential processing. During unwinding the recursion, the partial results are merged. The pseudocode in Listing 5 shows the skeleton of a fork/join computation.

Parallelism is achieved by processing only one half of the divided input whereas the second one is *forked*, i.e., it is made available for parallel processing. If the own share of the work has been processed, the spawned task is *joined*, i.e., the initiating process waits for its completion through a parallel process or completes it by itself, if necessary.

For efficient forking and joining, every process maintains its tasks in a double-ended queue (deque) as shown in Figure 7.3. One queue end is exclusively used by the owning thread for pushing (*fork*) and popping (*join*) tasks. The other queue end is solely accessed by free processes seeking for pending tasks (*steal*).

Listing 5 Parallel computation in fork/join model.

```
1: function COMPUTE(task)
2:     if task is small enough then
3:         compute task
4:         return result
5:     else
6:         split up task in subtask1 and subtask2
7:         schedule subtask2 for parallel execution (fork)
8:         compute subtask1
9:         wait for/complete subtask2 (join)
10:        merge results from subtask1 and subtask2
11:        return merged result
12:    end if
13: end function
```

Besides the elegant realization of lightweight parallelism at variable granularities – the task size depends on the recursion level at which it was created – fork/join has several other advantages. In practice, the number of steals is fairly low, because at the beginning larger subtasks are forked, which quickly saturate all free processes. Thereafter, the framework does not require any further load balancing. In contrast to many other parallelization schemes, the model also allows to compose nestings of parallel computations, i.e., every elementary subtask can be the root task of a nested parallel computation. Furthermore, fork/join algorithms have a good locality with respect to the input because processes execute forked subtasks in LIFO order. In fact, the processing order of fork/join with a single process is identical to sequential processing. In the general case, a task is seamlessly distributed over any number of processes.

Essential for the fork/join model are the fast creation of subtasks and a reasonably large recursion depth to create enough subtasks in the queue for stealing. Note, an overflow of the task queue is unlikely because of the logarithmic depth of a typical divide-and-conquer recursion. However, asymmetric partitioning patterns are prohibitive, because it provokes "steal-back" thrashing and results in poor locality.

Producer/Consumer Model

The basic producer/consumer model is suitable to schedule tasks from sequential inputs. The initiating process, the producer, continuously creates tasks and submits them to a simple FIFO queue, while one or more free processes, the consumers, poll the queue for new tasks to process them in parallel.

Even though this is the simplest of all parallel patterns, it has several advantages. Additional consumers can be dynamically added or removed and, if necessary, the single producer process can anytime take over the consumer part and process generated subtasks interleaved with the input partitioning. If sufficient computing resources are available, the producer can process the input almost without interruption and at very high speed. If the input is read from external storage, chances are good to utilize full sequential I/O bandwidth. Finally, the synchronization overhead of the task queue is as little as in the fork/join model. Further communication between parallel processes is not required.

In contrast to the fork/join model, the producer/consumer model is more sensitive to task granularity. Because the former creates tasks of different granularities, it is more likely to hit a partition size that leads to a good balance of parallelism and runtime overhead. A producer process, however, schedules only tasks of a certain size. If the tasks are too small, it provokes a high overhead; if they are too large, it costs parallelism. Clearly, one may collect queueing statistics and heuristics to adjust the size of subtasks dynamically, but dynamically determining the task size that balances the enqueue/dequeue ratio and optimally utilizes available processors, memory, and I/O is prohibitively costly.

Worker Pool Organization

To meet the requirements of both sequential and random access binding sequences, we use a pool of pre-allocated worker processes, which supports scheduling queues for fork/join tasks and producer/consumer tasks at the same time. A schematic overview of the pool is given in Figure 7.4. As standard in fork/join models, each worker has a local task dequeue, which is used by the owning process like a stack and by others like a queue. For producer/consumer tasks, the pool maintains a global task queue, which is uniformly accessed by all workers in a FIFO manner. Idle processes are queued until additional work is available.

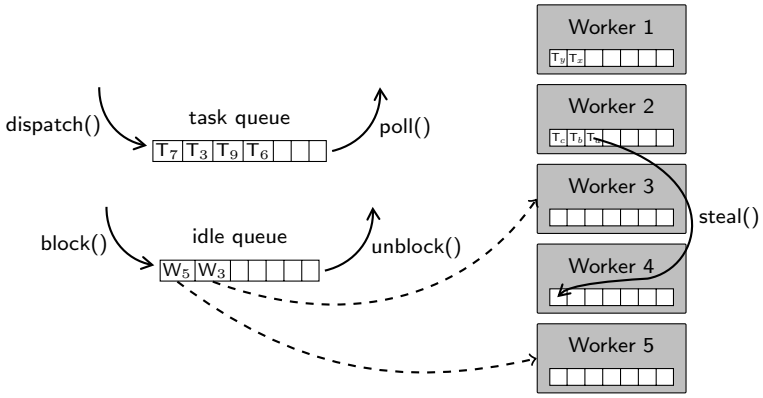


Figure 7.4.: The jork/join pool with 5 worker threads.

The number of allocated workers should be chosen with respect to the number of available of hardware contexts. A pool with less workers than hardware contexts cannot fully utilize the computing capabilities of the platform, but may be reasonable if some resources need to be reserved for system services etc. More workers than hardware contexts produce overhead for additional context switches, but may also keep the system utilized if processes are frequently suspended for I/O.

The main loop executed by each worker process is shown in Listing 6. First, the worker looks for a pending task at the tail of its local queue (line 4), then at the head of the global task queue (line 6), and finally at the heads of the local queues of other workers (line 9). This process continues until a pending task is found for execution (lines 12-13). To avoid needless spinning and expensive memory synchronization between parallel processes, the worker is suspended when a retry threshold is reached (lines 17-22). Before blocking, the worker enqueues itself in the idle worker queue (line 17) and then checks again if a new task was assigned to the worker after the local task queue was inspected. Without this re-check, it may happen that the *unblock()* signal gets missed if the worker is already removed from the idle queue (line 20).

The general task dispatch routine is shown in Listing 7. First, the idle queue is inspected (line 2). If a free worker was found, the task is

Listing 6 Main loop of worker processes.

```
 1: function RUN
 2:     retries ← 0
 3:     while TRUE do
 4:         t ← try_dequeue_from_local_tail()
 5:         if t is null then
 6:             t ← try_dequeue_from_pool_head()
 7:         end if
 8:         if t is null then
 9:             t ← try_steal_from_local_queues()
10:         end if
11:         if t is not null then
12:             execute(t)
13:             retries ← 0
14:         else if retries has not reached threshold then
15:             retries ← retries + 1
16:         else
17:             enqueue_in_idle_queue()
18:             t ← try_dequeue_from_local_tail()
19:             if t is null then
20:                 block(_self)
21:                 retries ← 0
22:             end if
23:         end if
24:     end while
25: end function
```

placed in the worker's local queue and the worker is unblocked (lines 4-5). Otherwise, the task is added to the global task queue, where it will be picked up by a running worker as soon as it has drained its local task queue.

The task join routine shown in Listing 8 is almost identical to the main loop of a worker. If the joined task is not yet completed (line 9), the joining worker uses the mean time to process pending tasks from the global queue (lines 11 and 20), the local queue (line 14), and the local queues of others (line 17). The flag *sequential* is used to ensure that tasks are always joined according to the caller's local forking scheme.

Listing 7 Dispatching fork/join and producer/consumer tasks.

```
 1: function DISPATCH(task)
 2:     w ← try_dequeue_from_idle_queue()
 3:     if w is not null then
 4:         enqueue_in_worker_queue(w, task)
 5:         unblock(w)
 6:         return TRUE
 7:     else
 8:         enqueue_in_task_queue(taks)
 9:         return FALSE
10:     end if
11: end function
```

This is crucial, because joining tasks in arbitrary orders may lead to overflowing task queues.

If a fork/join task is joined, the most recent task in the local queue will be the joined task itself. This way, execution of the found pending task (line 23) will be the end of the loop. However, if the joined task was stolen by a parallel worker, the loop continues seeking for further uncompleted tasks. After several unsuccessful attempts to find some work, the worker is suspended until the joined task is completed (line 17). The corresponding notification will be issued by the respective stealer.

7.3. Operator Sinks

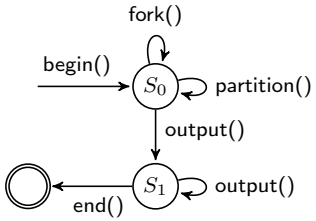
As shown in Section 4.3.2, the pull-based and push-based operator models have a fairly similar structure. Merely, they differ in the philosophy of whether something is actively produced or reactively delivered. In this aspect, the eager nature of the push-based model suits better to the parallel computation of individual loop iterations than the reactive pull-based model. The realization of parallelism within a sink pipeline is explained in the following sections in detail.

Listing 8 Task join routine.

```
1:  function JOIN(task, sequential)
2:      if sequential then
3:          done ← try_execute(task)
4:          if done then
5:              return
6:          end if
7:      end if
8:      retries ← 0
9:      while task is not finished do
10:         if sequential then
11:             t ← try_dequeue_from_pool_head()
12:         end if
13:         if t is null then
14:             t ← try_dequeue_from_local_tail()
15:         end if
16:         if t is null then
17:             t ← try_steal_from_local_queues()
18:         end if
19:         if t is null and not sequential then
20:             t ← try_dequeue_from_pool_head()
21:         end if
22:         if t is not null then
23:             execute(t)
24:             retries ← 0
25:         else if retries has not reached threshold then
26:             retries ← retries + 1
27:         else
28:             wait_for_task(task, _self)
29:             retries ← 0
30:         end if
31:     end while
32: end function
```

```
type Sink {
  void begin()
  void output(Buffer buffer)
  void end()
  Sink fork()
  Sink partition(Sink stop)
}
```

(a) Sink interface



(b) Sink state automata

Figure 7.5.: Extended operator sink model for parallel processing.

7.3.1. Parallel Data Flow Graphs

The basic begin-output-end interface of a sink has been extended as shown in Figure 7.5(a). The *output()* operation accepts a buffer of tuples instead of a single tuple at a time. The operations *fork()* and *partition()* were added to dynamically create a data-parallel graph of sinks. Except of the terminating Return, every sink writes to a single output sink, which is used according to the state diagram[2] in Figure 7.5(b).

First, we look at the normal case. Before the first output is emitted to a sink, the operation *begin()* is called to initialize it. It is followed by a sequence of *output()* calls, which passes over a buffer of tuples to be processed. Finally, *end()* signals the sink that all input tuples were received. In a non-blocking sink like ForBind, the *end()* is just propagated to the subsequent sink, but in a blocking operator like OrderBy, it triggers the sorting of all tuples received, which are then emitted to the next sink in a series of *output()* calls.

Parallelism within the pipeline is created by using worker processes in some sinks to actively "pump" data through the pipeline. If such a sink receives tuples via an *output()* call, it encapsulates the local processing of the received tuples together with the duty of propagating the result to the next sink in a task, which is submitted to the worker pool. Conceptually, this creates pipeline parallelism similar to the exchange operator [Gra94]. By dividing the work into multiple tasks, however,

[2]For the purpose of demonstrating the core concept, we excluded the handling of error conditions.

the sink is capable of creating data parallelism as well.

In general, multiple output tasks must not use the same sink in parallel, e.g., because it causes race conditions within the sink or because it results in chaotic tuple order. Therefore, each task must have its own private sink, which is created by *forking* the original one. The sink returned by calling *fork()* is an independent copy of the sink, which can be safely used in parallel. Of course, every fork sink can be forked itself.

Usually, forking requires to fork the subsequent sink, too. In fact, the forking process propagates through the pipeline to sinks, which do not operate on a per-tuple level, i.e., the blocking sink types OrderBy, GroupBy, and Return and the enumerator Count.

The operation *partition()* is a special variant of *fork()*, which creates independent forks for blocking sink types. The fork obtained by calling *partition()* on an OrderBy sink, for example, will only sort the tuples arriving at itself and at all "normal" forks of it. Its purpose will be explained in Section 7.3.4.

The forking mechanism turns the straight pipeline dynamically into a directed acyclic graph as depicted in Figure 7.6. Obviously, a sink graph shares similarity with the nested iteration tree of a FLWOR like the one in Figure 7.1. Accordingly, we distinguish between *fan-out* sinks (ForBind, LetBind, Select, ...) and *fan-in* sinks (OrderBy, GroupBy, Count, Return).

In principle, every sink type can make use of the forking mechanism. However, for practical and performance reasons, we use it only in ForBind sinks, because they embody the expensive looping nature[3] that we want to parallelize. Hence, ForBind sinks are the active drivers in the pipeline, which is indicated in Figure 7.6 by the shaded boxes around them.

7.3.2. Fan-Out Sinks

The fan-out sink types LetBind and Select are the simplest kinds of sinks. They process a single tuple at a time and output at most the same number of tuples as were received. Furthermore, they do not use

[3]The same looping nature is of course also present in, e.g., Join and WindowBind. For brevity, we want to focus the discussion on ForBind, but the presented mechanism can be transferred to other looping sink types, too.

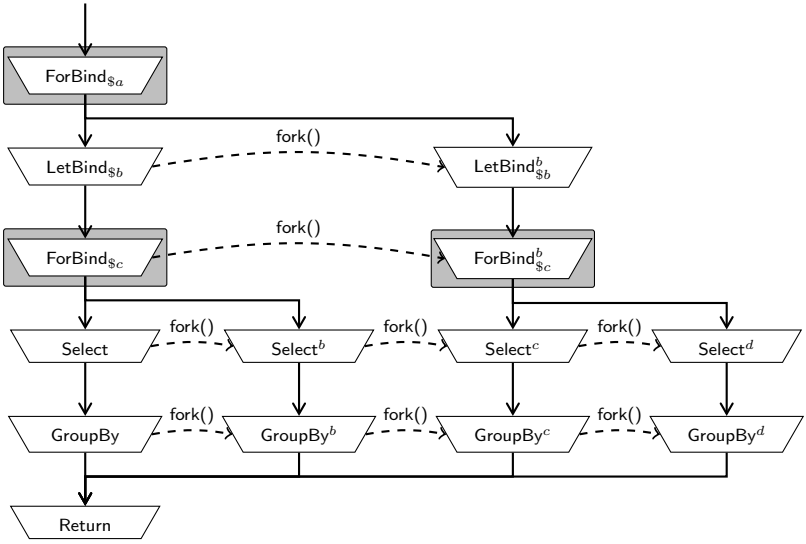


Figure 7.6.: Fork tree of operator sinks.

a shared data structure like a hash table so that the processing logic must not be adapted for parallel work at all.

A fork creates a simple copy for the simultaneously forked output sink. As example, Listing 9 demonstrates the simple structure of a LetBind[4] sink. Note that *begin()* and *end()* are not listed because they solely propagate the call to the next sink.

The implementation of ForBind can be done similarly, but, because we want it to drive the parallel computation, *output()* requires a bit more effort as shown in Listing 10. First, the output logic is decoupled from the ForBind by creating and executing for each input tuple a binding task, which encapsulates the tuple, the corresponding binding sequence, and an output sink (line 7). Note that each task is executed directly and not submitted to the worker pool. The scheduling of parallel work

[4]Note that we use an object-oriented style in the following listings: Variables starting with an underscore (e.g., *_sink*) denote member variables and the arrow notation (→) is used to indicate a method call.

Listing 9 Operations of a `LetBind` sink.

```
 1: function LetBind:output(buffer)
 2:     for each tuple in buffer do
 3:         sequence ← _binding_expr→evaluate(tuple)
 4:         output_tuple ← concat(tuple, sequence)
 5:         replace(buffer, tuple, output_tuple)
 6:     end for
 7:     _sink→output(buffer)
 8: end function

 9: function LetBind:fork()
10:     sink ← _sink→fork()
11:     fork ← create_letbind_sink(_binding_expr, sink)
12:     return fork
13: end function

14: function LetBind:partition(stopAt)
15:     sink ← _sink→partition(stopAt)
16:     partition ← create_letbind_sink(_binding_expr, sink)
17:     return partition
18: end function
```

will happen later if a bind task turns out to be large enough.

Noteworthy is that the sink is forked each time a new task is created (lines 5-6). This is necessary because we cannot know in advance if further bind tasks will be created, e.g., when *output()* is called again. However, we know that every task that will be created in the future must be completely decoupled from the previously created ones. Hence, each task needs to have its own fork of the sink, which in turn requires to keep always one fork spare. To signal the subsequent sink correctly the final *end()*, we must therefore "fake" proper use of the spare sink by issuing an extra *begin()* before *end()*.

The routine *process_partition()* shown in Listing 11 is the centerpiece of the framework. It is structured similarly to the fork/join pattern sketched in Listing 5. First, the binding sequence is partitioned into two halves, *part*1 and *part*2 (line 2). If the binding sequence is small enough to be handled by a single process, *part*2 will be empty and *part*1

Listing 10 Output operation of a `ForBind` sink.

```
1: function ForBind:output(buffer)
2:     for each tuple in buffer do
3:         sequence ← _binding_expr→evaluate(tuple)
4:         if sequence is not empty sequence then
5:             sink ← _sink
6:             _sink ← _sink→fork()
7:             task ← create_bind_task(sink, tuple, sequence)
8:             task→process_partition()
9:         end if
10:    end for
11: end function
```

– the binding sequence itself – is directly processed (line 4). However, if the binding sequence is large enough, it will be processed in parallel.

Sequential Binding

The sequential processing of small binding sequences or partitions is straightforward. The pseudocode is given in Listing 12. In a simple loop over the given input sequence, the output tuples are created by concatenating the sequence elements to the current context tuple (line 5) and added to an output buffer (line 6). At the end and if the buffer size is exceeded during the loop, all tuples produced so far are emitted to the output sink and the buffer is cleared (lines 7-10 and 12-13).

Parallel Binding of Random Access Sequences

If the binding sequence is large enough for parallel processing and supports random access, the two partitions are processed in the recursive divide-and-conquer strategy of the fork/join model (lines 6-11). The partitions are encapsulated in separate subtasks (lines 6-8), whereby the sink is forked for the second subtask. The latter is then forked for parallel execution, i.e., submitted to the worker pool, before the first one is executed by the current process (lines 9-10). Afterwards, the second task is joined (line 11). The recursion ends if the created partitions are small enough for sequential processing.

Listing 11 Execution logic of a bind task.

```
1: function BindTask:process_partition()
2:     part1, part2 ← split(_sequence, MIN_SIZE, MAX_SIZE)
3:     if part2 is empty then
4:         bind(part1)
5:     else if _sequence supports random access then
6:         sink2 ← _sink→fork()
7:         task1 ← create_bind_task(_sink, _tuple, part1)
8:         task2 ← create_bind_task(sink2, _tuple, part2)
9:         fork(task2)
10:        task1→process_partition()
11:        join(task2, false)
12:    else
13:        queue ← allocate_queue()
14:        while TRUE do
15:            task ← create_bind_task(_sink, _tuple, part1)
16:            if part2 is not empty then
17:                _sink ← _sink→fork()
18:            end if
19:            dispatch(task)
20:            enqueue(queue, task)
21:            if part2 is empty then
22:                break
23:            end if
24:            if size(queue) reaches threshold then
25:                dispatched ← dequeue(queue)
26:                join(dispatched, true)
27:            end if
28:            part1, part2 ← split(part2, MIN_SIZE, MAX_SIZE)
29:        end while
30:        while queue is not empty do
31:            dispatched ← dequeue(queue)
32:            join(dispatched, true)
33:        end while
34:    end if
35: end function
```

Listing 12 Bind operation in bind tasks.

```
 1: function BindTask:bind(part)
 2:     _sink→begin()
 3:     buffer ← allocate_buffer(BUF_SIZE)
 4:     for each element in part do
 5:         output_tuple ← concat(_tuple, element)
 6:         add(buffer, output_tuple)
 7:         if buffer is full then
 8:             _sink→output(buffer)
 9:             clear(buffer)
10:         end if
11:     end for
12:     _sink→output(buffer)
13:     clear(buffer)
14:     _sink→end()
15: end function
```

Parallel Binding of Sequential Sequences

Large sequential binding sequences are processed in a loop, which dispatches a binding task for the respective head partition to the global task queue of the worker pool (lines 15-19). The rationale behind this loop is the assumption that free workers will take care of the submitted tasks while the current process can continue with the – potentially expensive – iteration of the binding sequence. Note that each task created will be small enough to be processed without further partitioning.

To keep track of the tasks dispatched, they are locally held in a queue (line 20). If the queue size reaches a certain threshold, the oldest task is dequeued and joined to ensure that new tasks are not created faster than they are processed (lines 24-27). Furthermore, this mechanism guarantees continuous progress, because the joined task will be indirectly executed by the joining process if it has not yet been adopted by a free worker (see 7.2.2).

At the end of each iteration, the second partition is partitioned again, i.e., a new head chunk is pulled from the sequential input (line 28). The looping continues until the sequence has been fully consumed (lines 21-22). At the end, the remaining tasks in the submission queue are joined to ensure that all of them will be fully processed (lines 30-33).

Sequence Partitioning

The crucial point in the whole routine is clearly the partitioning of the binding sequence. Multiple aspects play here a role.

If the sequence is materialized, e.g., because the binding expression was eagerly evaluated, we know its size and can easily choose a suitable splitting point. Most likely, the materialized sequence also supports random access so that we can represent the partitions with lightweight pointers into the base sequence and must not cautiously keep an eye on how much additional memory we need for copying the data into materialized partitions.

If the sequence is lazy, which is the normal case, the partitioning is more difficult. The partitioning step will typically not know the actual size of the sequence, because this would require to compute the entire sequence. In general, it will also not be possible to create two lazy partitions by splitting up the computation logic encapsulated in a lazy sequence. Accordingly, the partitioning step will need to materialize at least the items for $part1$, whereas $part2$ reflects the remaining part of the partially-evaluated sequence. Note that this nicely fits into the parallel framework because the computation of the remaining items will likely happen in a separate worker process.

To keep control of the memory used for the materialization, the parameters MIN_SIZE and MAX_SIZE specify minimum and maximum buffer size limits for the partitioning step, respectively. If the binding sequence is smaller than MIN_SIZE, then $part1$ is the (materialized) sequence itself and $part2$ is empty. If the binding sequence is larger than MAX_SIZE, then the leading MAX_SIZE share of the binding sequence is materialized in $part1$ and the (un-materialized) rest in $part2$. Note, in the rather rare cases where a lazy sequence can be partitioned without any partial materialization (e.g. in a range expression `1 to 10000`), the two parameters can be completely ignored.

7.3.3. Fan-in Sinks

Fan-in sinks come in two flavors. Some are independent of the order in which tuples arrive; others require that the input is received in exactly the same order as in a serial execution. This character of a fan-in sink is not necessarily determined by the type of the operator; it can be the

case that one operator is realized by two different sink implementations of which the compiler chooses the appropriate one.

For example, if a FLWOR pipeline is evaluated in an ordered context, the Return sink will need to enforce the right (serial) output order. But if output order must not be preserved, the results from all forks can be collected in parallel, which is typically much faster.

In the following, we describe the skeletons for realizing both classes of fan-in sinks and explain how these skeletons are used to implement concrete sinks for the different operator types.

Concurrent Sinks

The easiest way to create an order-insensitive fan-in sink is to implement *fork()* so that it returns the sink itself instead of creating a real fork. Because every parallel branch in the fork tree will then write to the same sink, there are some aspects that need to be considered. First, one must be aware that, because of the parallel use, the operations *begin()*, *output()*, and *end()* will be called multiple times and in arbitrary schedules. Second, one must take appropriate measures against race conditions, because these calls may be concurrent.

Fortunately, the nature of a concurrent sink is transparent and every caller will just use its fork according to the protocol shown in Figure 7.5(b). Thus, we know that, even in a highly parallel environment, the very first call will be a *begin()* and the very last call will be an *end()*. Consequently, one must only keep track of how many "virtual" sinks have been forked and are still active, to find out which of the concurrent *begin()* and *end()* calls really indicate begin and end of the *output()* calls. Chronologically intermediate calls of *begin()* and *end()* may be safely ignored.

The basic skeleton for a concurrent sink is shown in Listing 13. It can be used for various types of order-insensitive sinks like OrderBy, GroupBy (hash-based), or Return (unordered).

For lightweight and highly parallel bookkeeping, we use two simple integer variables, *_state* and *_count*, for toggling the state and counting the number of active forks, respectively. Both variables are exclusively modified with atomic memory instructions, because locks or mutexes are prohibitively expensive under highly concurrent access.

The member variable *_state* is initialized with 0. While the value of *_state* is not 2, each caller of *begin()* will try with an atomic compare-and-swap operation to change the value from 0 to 1 (line 4). The first and only one that succeeds will perform the actual initialization (indicated by line 6), all others will be trapped in the spinlock (lines 3-9) until the winner has finished the initialization step and set the value of *_state* to 2 (line 7). All future calls of *begin()* will read the atomic variable only once (line 2), which is on most platforms a very cheap operation.

The bookkeeping in *end()* and *fork()* is similarly lightweight. They increment, respectively decrement the atomic variable *_count*, which is initialized with 1. If *end()* has decremented *_count* to 0, all "virtual" sinks have been closed and the sink can be finalized (lines 13-15).

Listing 13 Skeleton of a concurrent sink.

```
 1: function BEGIN()
 2:     state ← volatile_get(_state)
 3:     while state ≠ 2 do
 4:         first ← atomic_cmp_and_swap(_ready, 0, 1)
 5:         if first then
 6:             ...                          ▷ sink-specific initialization
 7:             volatile_set(_state, 2)
 8:         end if
 9:     end while
10: end function

11: function END()
12:     alive ← atomic_decrement_and_get(_count)
13:     if alive is 0 then
14:         ...                              ▷ sink-specific housekeeping
15:     end if
16: end function

17: function FORK()
18:     alive ← atomic_increment_and_get(_count)
19:     return _self
20: end function
```

The presented base routines are already concurrency-safe, but note that *output()* must still be protected against race conditions. In some cases, a simple mutex will be sufficient, e.g., in OrderBy for adding the given tuples to the sort buffer. In other cases, operation-specific synchronization or concurrent data structures (e.g., for the hash table in a GroupBy) will achieve better parallelism.

Note also that scalability of sorting and aggregation is often improved by organizing it in multiple parallel stages, e.g., sort-merge trees or multi-stage computation of additive aggregates like `sum()`, `min()`, and `max()`. Evidently, such advanced evaluation strategies can be realized in the sink model and with the presented skeleton, too.

Sink Chaining

Order-sensitive fan-in sinks can be realized without expensive sorting. The key observation for their realization is that they can restore the serial tuple order by themselves. The forking mechanism ensures that the fan-out created already produces a proper left-to-right order among the individual fork branches at each level. Each branch guarantees correct order for its own partition anyway, because it is executed by a single process. Accordingly, fan-in sinks can restore the serial order for tuples arriving in parallel by keeping track of the logical order relationships between created forks.

The cheapest way for controlling the serial ordering of sinks is illustrated in Figure 7.7. The forks are connected in a pointer chain, which points from each sink to the respective next one. Note that even if the sinks are forked concurrently in arbitrary schedules, the creation of a fork remains a local operation, which can be realized without any synchronization.

The tuples will arrive at the forks in a globally random but locally serial order. Hence, to restore the ordering, some tuples must be buffered locally until the logical serial process has reached the respective sink. The progress of this serial tuple flow is marked by a token, which is passed on from the left to right through the pointer chain. At the beginning, the token is at the left-most sink – the root sink.

The sink that owns the token is allowed to process the tuples received via *output()* directly. In parallel, all others can only perform eventual preprocessing steps, but must defer the actual serial part of the pro-

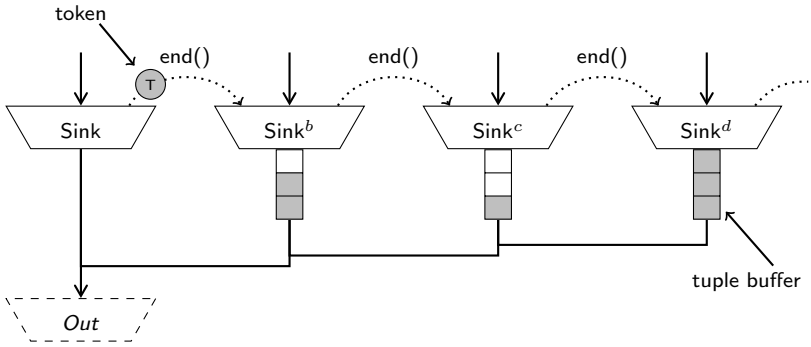


Figure 7.7.: Token passing and tuple buffering in serial fan-in sinks.

cessing. For example, a fork of a Return sink may evaluate the `return` expression if it does not possess the token, but it must not yet, e.g., append the result to the result sequence.

If *end()* is called on the owning sink, the token is passed on to the right sibling by following the next pointer. If *end()* has already been called for the sibling sink, any pending tuples are processed and the token is passed further to the next sibling in the chain. This propagation process continues until the token is handed over to an unfinished sink or the end of the chain is reached.

Token passing must be carefully coordinated and synchronized to work properly. Particularly, *output()* and *end()* require further explanation, whereas *begin()* and *fork()* are trivial and omitted for brevity.

The overall concept is easier understood by starting with the *output()* routine shown in Listing 14. The integer member variable *_token* holds the token state of the sink expressed here with the symbolic constants $NEED$, $AWAIT$, and HAS. The root sink initializes *_token* with HAS. All forks initialize *_token* with $NEED$. Note that we use an atomic variable because the state is accessed and modified concurrently.

Each time *output()* is called, the sink's token state is checked (line 2). If the sink has the token, possibly pending tuples from previous *output()* calls are handled first (line 4-6). Then, the actual tuples received are processed (indicated by line 7). Without the token, the given tuples are just added to a local buffer (line 9).

Listing 14 Conditional output handling in chained sinks.

```
 1: function OUTPUT(buffer)
 2:     t ← volatile_get(_token)
 3:     if t = HAS then
 4:         if has pending output then
 5:             process_pending()
 6:         end if
 7:         ...                              ▷ sink-specific tuple processing
 8:     else
 9:         add_pending(buffer)
10:     end if
11: end function
```

Pseudocode for *end()* is given in Listing 15. If the sink possesses the token (lines 14-20), pending output is processed if necessary (lines 15-17), the token is promoted (line 18), and sink-type-specific housekeeping operations (e.g., closing file handles and propagating *end()* to the output sink) are performed (indicated by line 19).

If the sink is at the beginning not in possession of the token, it tries to switch the token state atomically from $NEED$ to $AWAIT$ (line 5). If the compare-and-swap operation fails, the predecessor must concurrently have passed on the token[5] to the sink (line 12) and the algorithm continues just as if the token had been owned at the beginning of *end()*.

If the compare-and-swap operation succeeds, the predecessor is put in charge of processing all pending output and propagating the token. The current process can exit and continue with executing other tasks (line 10). However, if parallel workers fill the local buffers too fast for the token propagation process, uncontrolled parallelism will quickly lead to memory exhaustion. Hence, to give the slower token propagation the chance to catch up and free utilized buffer memory, the current worker is suspended if necessary until the token reaches the sink (lines 7-9).

The routine *promote_token*(), shown in Figure 16, propagates the token the next successor, which has not yet finished. As complement to the compare-and-swap operation in *end()*, it tries to switch the state

[5] A real implementation might also use a state to signal and propagate errors, but this is left out here for brevity.

Listing 15 Finalization of a chained sink.

```
 1: function END()
 2:     t ← volatile_get(_token)
 3:     if t = NEED then
 4:         _worker ← current_worker()
 5:         done ← atomic_cmp_and_swap(_token, NEED, AWAIT)
 6:         if done then
 7:             if suspend recommended then
 8:                 suspend(_worker)
 9:             end if
10:             return
11:         end if
12:         t ← volatile_get(_token)
13:     end if
14:     if t = HAS then
15:         if has pending output then
16:             process_pending()
17:         end if
18:         promote_token()
19:         ...                                  ▷ sink-specific housekeeping
20:     end if
21: end function
```

of the successor sink atomically from $NEED$ to HAS (line 4). If it is successful, the routine ends (line 6). Otherwise, the calling process wakes up the suspended worker, if necessary (line 8), takes care of the pending output left behind (lines 9-11), and performs the respective housekeeping (indicated by line 12). Afterwards, it starts a new attempt for the next successor in the chain (line 13). If the end of the chain is reached, sink-specific cleanup operations are performed (line 16).

The serialization of parallel tuple streams plays an important role in the framework, because the operator semantics defaults to order-preserving output. Besides the ordered version of Return, the skeleton described is also used for implementing Count or a sequential variant of GroupBy, which efficiently aggregates inputs that are already sorted by the grouping key. Furthermore, the mechanism can be used to realize a

Listing 16 Token propagation in chained sinks.

```
 1: function PROMOTE_TOKEN()
 2:     n ← _next
 3:     while n is not null do
 4:         done ← atomic_cmp_and_swap(n→_token, NEED, HAS)
 5:         if done then
 6:             return
 7:         end if
 8:         unsuspend(n→_worker)
 9:         if n has pending output then
10:             n→process_pending()
11:         end if
12:         ...                          ▷ housekeeping as in normal end()
13:         n ← n→_next
14:     end while
15:     if n is null then
16:         ...                                        ▷ finalize sink chain
17:     end if
18: end function
```

universal Valve operator, which can be plugged into a pipeline wherever a serialization of parallel output is required.

Technically, chained sinks are special, because they are the only components that need to manually interfere with the scheduler of the operating system. Like any form of serialization they can and will need to slow down parallelism in some situations and suspend worker processes. Note that deadlock freedom is nevertheless guaranteed, because both scheduling mechanisms use a binary partitioning scheme and always give execution precedence to the left partition. As a result, at least one process – the one which is assigned to the task with the current token-owning sink – will be able to make progress. The proof of this property is given in Appendix B.

7.3.4. Join Sink

The join operator belongs to a special category of fan-out operators. In contrast to a simple binding operator, it computes for each input tuple a combination of the results of two pipelines – the left input and the right input. As discussed in Section 5.2.3, there is often the chance to re-use the result of one or both input pipelines if their outcome is independent or at least partially independent of the input tuples received. In particular, we considered join groups, i.e., sequences of input tuples, which allow to re-use the inner join table, once it has been loaded with the output of the right input pipeline.

Exploiting join groups in a parallel setting is a challenging task, because tuples arrive randomly at multiple forks. Furthermore, the lookup table is a shared resource which is concurrently accessed by many sinks. Even though loading and probing happens in disjoint phases, everything must be properly timed and synchronized. Tuples from the left join input pipeline can be probed against the same lookup table only if they were produced from input tuples of the same join group. Altogether, this calls for are careful design, which impacts on parallelism as little as possible.

Figure 7.8 illustrates the conceptual realization of a join. In contrast to other operators, it consists of three individual sink types. The actual Join sink at the top orchestrates the join computation. For each input tuple received, it checks if the respective lookup table must be rebuilt, i.e., if the input tuple belongs to a different join group than the previously processed one. If necessary, it instantiates then a new sink pipeline for the right join input, which outputs to a Load sink. Then the respective input tuple is fed to this pipeline and the output tuples are loaded into the lookup table. When the lookup table has been built, the Join sink can pass the input tuple further to the left input pipeline, which in turn outputs a Probe sink. There, the join candidate tuples are probed against the lookup table and the actual join output tuples are constructed and emitted to the next pipeline operator.

Because a join group is actually a group of related tuples, which all belong to the same superordinate iteration, they appear sequentially in the input tuple stream. By exploiting this property and enforcing that all tuples from the same join grouping are processed together, we can avoid that the same lookup table is computed twice. Therefore, Join is

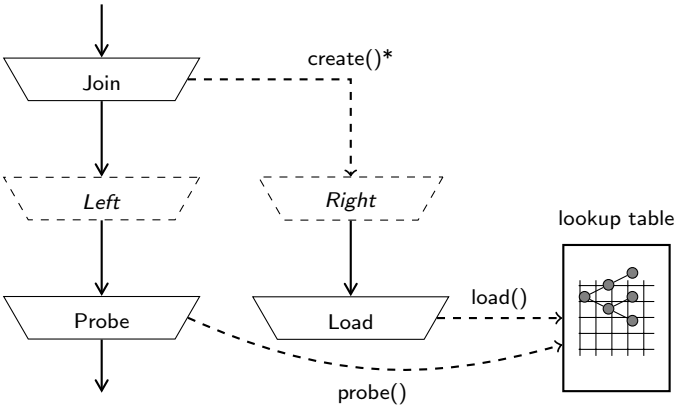


Figure 7.8.: Loading and probing of join lookup table.

realized as a chained sink, but with a less strict emphasis on sequential order. We are only interested in a partial order, which preserves the locality of join groups. Hence, the token passing is not used to control, which fork is allowed to perform output (i.e., to feed tuples to the left join pipeline), but which sink has the right to rebuild the lookup table.

Once the lookup table has been loaded by the fork that owns the token, all successor forks can probe this table in parallel. If a successor fork receives tuples from "later" join groups, it must wait for the token to load the respective lookup table itself or at least wait until one of its predecessors does it. Of course, we can apply the buffering mechanism again to keep parallelism alive as long as possible. Note also that the idea can be adapted to implement optimized variants of Concat, Union, and Intersect, too.

The left and the right join input pipelines must not obey any restrictions with respect to parallelism and forking. A Load sink is a concurrent fan-in sink, which adds arriving tuples to the lookup table in parallel. However, if the pipeline is evaluated in an ordered context, a serializing Valve must be prepended so that the lookup table can be loaded in the serialized tuple order. A Probe is a normal fan-out sink.

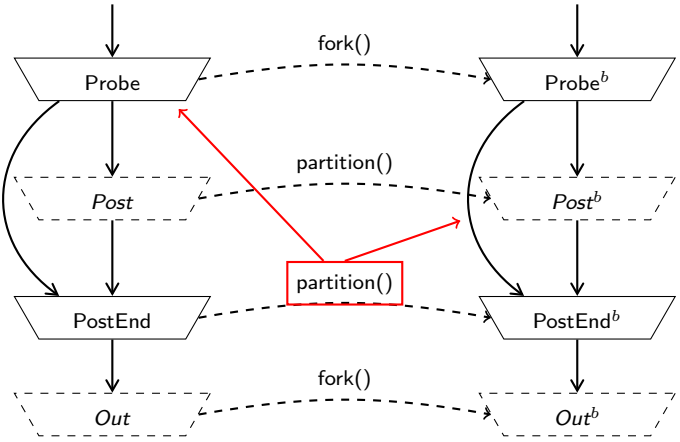


Figure 7.9.: Partitioned fork of optional post-join pipeline.

The realization of the optional post-join pipeline of `LeftJoin` is a simple extension of a normal join as shown in Figure 7.9. The optional post pipeline is directly attached to the Probe sink. It is separated from the actual join output pipeline by an additional PostEnd sink at the end. This fourth sink type serves two purposes. It is used as target to bypass the post-join pipeline if a probe tuple does not find a join partner, and it plays a special role as boundary for a special type of forking.

As detailed in Section 5.3.1, the post-join pipeline reflects a nested evaluation scope. Hence, each join result must be processed separately. In particular, this includes all relation-based operations (e.g., `GroupBy`), which must only span over tuples belonging to the same join match. Therefore, Probe issues *partition()* on the post join pipeline to create an independent sink branch for the join matches of each probe tuple.

As already mentioned, *partition()* is a special kind of *fork()* that creates independent forks for fan-in sinks. Like a normal fork operation, the call propagates through the pipeline. When the propagation reaches the PostEnd that was given by Probe as parameter, the PostEnd finalizes the post join partition and propagates it as a normal *fork()*.

As visualized in Figure 7.9, the partition mechanism is also used for forking. If *fork()* is called on a Probe sink, it is propagated through the optional pipeline as *partition()*. The corresponding PostEnd translates it then back into a normal *fork()*. A callback mechanism, indicated by the red arrows, establishes the connection between the newly created forks of Probe and PostEnd.

7.4. Performance Considerations

The framework presented is designed to create and maintain parallelism automatically. However, it is not completely "knob-less". Besides the worker pool size, there are several other factors, which influence parallel performance of a query. In the following, we consider three central aspects and discuss their influence on the general performance of the system.

7.4.1. Partitioning

The central factor in any parallel system is the granularity of parallel work. In the sink framework, it is controlled by the size of the binding sequence partitions created in a ForBind. As already mentioned, partitioning in a nested-loops scenario is difficult, because the cost of an iteration is hard to determine. Furthermore, the partitioning of sequential and lazy sequences is particularly critical.

Certainly, the framework will not be able to determine good partition sizes without statistics and cost estimates. But even if at least the size of a binding sequence is known at runtime, a good partition size is not obvious. Hence, the system or the user necessarily need to provide default values, whether they are appropriate or not.

Large partitions are preferable if the average loop cost per output tuple is low. However, coarse-grained partitioning can also result in poor parallelism and load skew. Furthermore, large partitions will consume a lot of memory if the partition mechanism must materialize lazy inputs. Smaller partitions are better for expensive loop bodies. They allow for higher parallelism, but may impose a large overhead per iteration. Ideally, the partition sizes can be controlled individually for individual pipeline levels. In many situations, domain knowledge and the nest-

ing depth of a binding operator already yields reliable information for choosing good partitioning sizes. Therefore, the prototype developed for this thesis allows to hint the compiler a good partition size.

7.4.2. Buffer Memory

Chained sinks use local buffers for storing already computed intermediate results. Clearly, the more memory they are allowed to allocate, the better they will be able to keep parallelism alive. However, a hard limit for the maximum total amount of buffer memory is hard to realize.

The buffer memory currently in use is a quickly fluctuating value. It is concurrently increased by parallel workers and decreased by the current token owner. Hence, controlling and enforcing the total amount of memory used exactly, would create a heavily contented hotspot. Therefore, our prototype relaxes the hard limit to reduce overhead. It uses a single atomic integer variable to keep track of the amount of buffer memory in use, but updates the value only if a sink is closed and some tuples have been added to the local buffer. Accordingly, it may happen that sometimes even more memory will be used than actually intended. However, if the memory use overruns the threshold, the suspension mechanism described will slow down parallelism until sufficient buffer memory has been reclaimed.

Nevertheless, the configured buffer memory is rather a guideline for the system and should be set defensive to avoid memory exhaustion. Practically, the maximum amount of overuse possible is indirectly determined by the number of available processes and the maximum size of the partitions emitted by the parent fan-out sink.

7.4.3. Process Management

The reason for rampant use of buffer memory is an imperfect parallel input provision for a serializing operation. A typical situation is illustrated in Figure 7.10. It shows a snapshot of the parallel output produced in a multi-level fork tree. The bar at the bottom visualizes the entire output that is fed to a chained fan-in sink. The dark-colored area at the bottom left of the bar stands for output already emitted and processed by the serializing sink. The light-colored areas show buffered output, which arrived too early for the serializing operation.

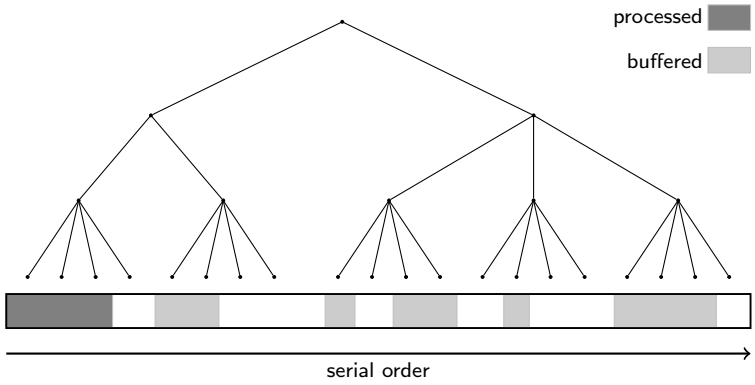


Figure 7.10.: Snapshot of serialized output in a parallel computation.

Obviously, less data had to be buffered if the parallel work concentrated on the left root branch of the fork tree first. But for several reasons, it is extremely difficult to avoid scattered output on the one hand and achieve high parallelism at the other hand. There are many unknown or uncertain factors in a pipeline like input sizes, selectivities, and operation overhead, which will influence the time needed to process a particular branch in the fork tree. Hence, it is almost impossible to make reliable estimates and optimize in the partitioning process. Furthermore, the fork tree grows recursively and in parallel, making it impossible to make globally optimal decisions without tampering parallelism through expensive control mechanisms.

Suspension of workers is the last resort when the memory shortage does not settle by itself. Although it is very effective to fight excessive memory usage, a better use of available computing resources is desirable. The actual goal is to reduce parallelism in the chained sink, but suspended processes are also unavailable for speeding up overall performance. Possibly, they were better utilized for other tasks or queries running in parallel. Pools with more workers than available hardware contexts and dynamically spawned processes are sometimes feasible solutions for compensating the temporary loss of computing power. However, they also increase the overhead for process management and context switching. Therefore, our prototype simulates only the suspension of a worker and processes other tasks while “waiting” for the token.

8. Evaluation

The concepts and techniques developed in this thesis have been implemented for empirical evaluation. In the following, we present and discuss the results of various experiments, in which we evaluated effectiveness and interplay between optimization rules and compilation techniques.

8.1. Experimental Setup

The compiler prototype implemented for this thesis is written in Java and uses XQuery as front-end language. To prove that our concepts are feasible to meet real-world requirements, we built a mature compiler, which has full coverage of XQuery 1.0, supports the `group-by` and `count` clauses of XQuery 3.0, and implements the XQuery Update Facility. Large parts of it have also been published as open-source XQuery engine called *Brackit*[1].

The optimizer performs rule-based optimization and applies sets of transformation rules in separate stages as detailed in Section 4.3.1. For testing specific optimizations, individual rules can be enabled and disabled as desired. Per default, the runtime of the prototype uses a standard pull-based operator model with open-next-close interface. For the parallelization experiments in Section 8.5, we also implemented the push-based operator model presented in Chapter 7.

To assess the compiler inside a database context, we implemented a native XDBMS prototype called *BrackitDB* from scratch. The system offers full ACID guarantees and internally uses the developed compiler and runtime for query processing. The XML store is inspired by the path-oriented storage of the native XDBMS *XTC* and stores the nodes of XML documents keyed by DeweyID labels in a specialized, disk-resident B$^+$-tree [HHMW05]. The storage supports fine-grained navi-

[1] Source code is available at `http://brackit.org`.

gation at the node level as well as efficient scan access. Furthermore, it features a rich set of advanced indexes for fast XML retrieval by content predicates and path predicates like in [MHS09].

The measurements were conducted on a dual Intel Xeon server with 4 cores at 2.66GHz, 4GB main memory, and two 500GB SATA drives in RAID level 3. The operating system was Ubuntu Linux 10.04 64-Bit with kernel version "2.6.32-24-server". For the parallelization benchmarks in Section 8.5, we used a quad Intel Xeon server with 24 cores at 2.93GHz, 96GB main memory, and a 1TB SATA disk drive under Ubuntu Linux 10.04 64-Bit with kernel version "2.6.32-41-server". The Java version used on both servers was Oracle Java 1.6.0 64-bit.

We report for all test queries the fastest runtime out of 10 timed runs. The timings include compilation time and serialization of the result to a `/dev/null` output stream. For the database benchmarks in Section 8.3, the results were completely transferred from the database server to the client residing on the same machine. For tests with Java-based systems, we ensured to perform a sufficient amount of warm-up runs for optimization by the JIT compiler before the measurements. The benchmark queries used can be found in Appendix C.

8.2. Main-memory Processing

Typical XQuery use cases for data integration and message processing operate on relatively small XML documents in main memory. Accordingly, we start with an evaluation, whether or not our prototype meets the standard requirements for this setting: fast compile times and a lightweight runtime for traversing XML trees.

8.2.1. Workload

To assess the XML performance of our prototype, we used the widely accepted XMark benchmark [SWK+02]. The suite consists of a data generator and 20 read-only queries, which cover a broad range of XQuery features and optimization challenges.

XMark models an auction platform with persons, items, bids, etc., which are represented as data-centric XML fragments and interspersed with small markup sections. The database consists of a single docu-

ment, which organizes the different entity types as clustered siblings under the root node. Although this kind of data organization is barely used in practice – semi-structured databases primarily consist of large collections of small, independent documents – XMark is a valuable tool to assess the quality of a query compiler, as its wide-spread use in research and industry shows.

Except for a few queries, which navigate paths with a `descendant` axis step, the benchmark solely makes use of normal `child` steps and `attribute` steps. Each query consists of a top-level FLWOR expression, which navigates into one or several document regions, where it performs more or less complex forms of filtering and result transformation. Thereby, every query addresses a certain optimization aspect, which requires the whole spectrum from logical transformation and pruning to physical optimization of operators and at the storage level. Accordingly, the main cost drivers in a query may vary considerably between two systems. In the following, we give a brief summary of the main challenges to give the reader a better insight into the results reported.

Query `Q1` is a point query, which searches for a person with a certain `id` attribute value. `Q2` and `Q3` evaluate path matching routines for multiple paths with (redundant) positional predicates. `Q4` contains a quantified expression as filter predicate, which requires to evaluate document order between fragments. `Q5` is a simple filter and count query, which also assesses the performance of the frequently required type casting from an uninterpreted string to a numeric value. `Q6` and `Q7` are scan-intensive queries, which contain multiple paths with `descendant` steps. `Q8` and `Q9` evaluate nested 1:n joins, respectively 1:n:1 joins of fragments from separate document regions. `Q10` performs an n:m join, which selectively copies many sub-elements from qualified fragments to construct complex results. `Q11` computes an n:m join, which returns a large number of results. `Q12` is almost identical to `Q11`, but contains an additional filter predicate, which reduces the result size. `Q13` performs only simple navigation and aims at fast reconstruction of the original document structure. `Q14` scans the document structure and performs substring matching on text content. `Q15` navigates a deep child path with 13 axis steps. `Q16` uses a deep child path with 10 steps as existential predicate. `Q17` uses a test for missing elements as filter predicate. `Q18` evaluates the efficiency of evaluating a simple user-defined function as predicate. `Q19` tests the sort performance with an `order by` clause.

Q20 performs multiple count aggregations by evaluating identical path expressions with similar filter predicates to manually compute disjoint partitions of qualified nodes.

8.2.2. Pipeline Optimization

In our first experiment, we addressed the basic performance of the prototype and effectiveness of pipeline optimizations. The optimizer configuration *nested* closely resembles standard XQuery semantics with nested evaluation and performs only basic pipeline rewritings like predicate pull-up. The *unnested* configuration additionally performs pipeline lifting and optimized aggregation. Finally, the configuration *join* employs the full set of pipeline optimizations including join rewriting.

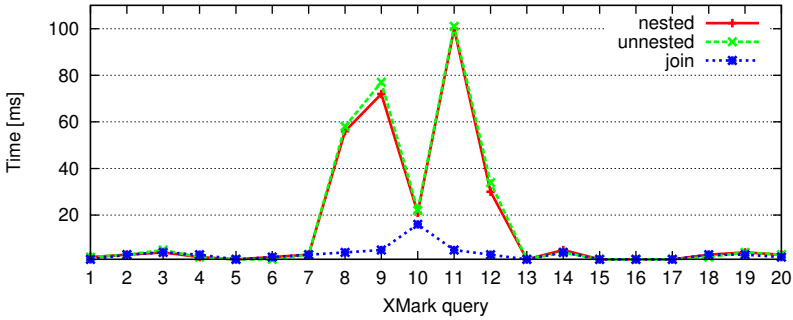
We ran the benchmark with scale factor 0.1, i.e., the queries were evaluated on a 12MB document in main memory. Path expressions were processed with basic navigation routines. The benchmark results are shown in Figure 8.1.

Although a 12MB document is already quite large for typical XML processing in main memory, the benchmark queries were not a real challenge for the prototype. Except for the join queries Q8-Q12, query times are below 5ms, which proves fast query compilation and efficient operators. However, the results for the join queries also demonstrate how nested `for` loops can quickly lead to dramatic increases in response time.

The difference between the configurations *nested* and *unnested* is only marginal. In most cases, nested evaluation even performed slightly better, because pipeline lifting results in larger tuples and, thus, higher overhead. The benefit of FLWOR unnesting as preparation step becomes apparent for the join queries. With the optimizer configuration *join* enabled, all implicit joins where detected by the compiler, which led to performance gains up to a factor of 20. The small resisting spike for Q10 is caused by the relatively complex result construction, which navigates 11 short paths for each join match to collect data.

8.2.3. Competitors

In the second experiment, we compared main-memory performance of our engine against current state-of-the-art competitors using the XMark



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
nested	2	3	4	2	1	2	3	56	72	21
unnested	2	3	5	2	1	1	3	58	77	22
join	1	3	4	3	1	2	3	4	5	16

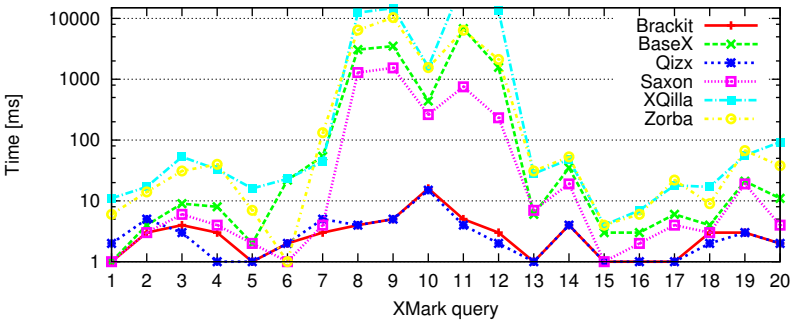
	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20
nested	100	30	<1	5	1	1	1	3	4	3
unnested	101	34	1	4	1	1	1	2	4	3
join	5	3	<1	4	<1	1	1	3	3	2

Figure 8.1.: XMark performance for different pipeline optimization levels on a 12MB document in main memory.

query set. The test field consists of both mature and widely used processors [BAS, Kay08, BBB+09, XQI, QIZ] and research or in-development systems [MXQ, QEX, VXQ]. Except for implementation details and individual optimization capabilities, all systems operate on pluggable main-memory stores and evaluate queries in the standard nested fashion with lazy iterators over intermediate results.

Because of the spread in runtime and system performance, we chose again scale factor 0.1. Except of XQilla and Zorba, which are realized in C++, all tested engines are implemented in Java and were configured for a maximum heap size of 1.5GB. Figure 8.2 shows the fasted runtimes for each query and system. Note, to keep the diagram readable, the graph includes only the results of the fastest six processors.

Clearly, our prototype shares the lead with Qizx and delivers high performance for each query type. Obviously, Qizx is the only other engine for main memory that was also able to detect and exploit join semantics in `Q8`-`Q12`. All other engines suffered from the nested loops.



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
Brackit	1	3	4	3	1	2	3	4	5	16
BaseX 6.7	1	4	9	8	2	22	54	3066	3504	440
MXQuery 0.6.0	106	147	200	147	142	166	299	1.0e6	1.5e6	8354
Qexo 1.11	9	11	16	Err	Err	Err	Err	Err	287	67
Qizx/open 4.1	2	5	3	1	1	2	5	4	5	15
Saxon HE 9.3	1	3	6	4	2	1	4	1288	1537	262
VXQuery 0.1	3	23	43	19	6	181	371	5058	1756	Err
XQilla 2.2.4	11	17	53	33	16	23	44	12426	14755	1598
Zorba 1.4.0	6	14	31	40	7	<1	132	6466	10305	1554

	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20
Brackit	5	3	<1	4	<1	1	1	3	3	2
BaseX 6.7	6777	1571	6	35	3	3	6	4	21	11
MXQuery 0.6.0	1.6e6	1.1e5	150	158	116	104	134	161	199	381
Qexo 1.11	Err	Err	5	17	6	8	7	1	24	Err
Qizx/open 4.1	4	2	1	4	<1	<1	1	2	3	2
Saxon HE 9.3	752	233	7	19	1	2	4	3	19	4
VXQuery 0.1	8969	8863	9	123	4	11	30	Err	81	18
XQilla 2.2.4	40922	13342	28	48	4	7	18	17	56	93
Zorba 1.4.0	6404	2116	31	53	4	6	22	9	67	38

Figure 8.2.: Comparison of XMark performance of XQuery engines on a 12MB document in main memory.

Only Saxon, often regarded as unofficial XQuery reference implementation, achieved slightly better results than the other engines without join support. It is able to retain intermediate results in on-demand indexes, which help to reduce the effects of quadratic scaling in nested loops. For all other queries, Saxon, BaseX, XQilla, and Zorba performed solidly, but did not reach the performance of our prototype and Qizx.

VXQuery is still in a very early development phase and execution failed for `Q10` and `Q18`. However, from the current state, it seems likely that future versions will achieve performance comparable to the broad field of mature processors without join support.

Qexo is the only engine, which generates plain Java byte code from a query plan. However, it is also still in a highly experimental stage and fails for many queries. Furthermore, Qexo was not able to exploit the potential advantage of direct code generation, because we tested ad-hoc performance and the Java just-in-time compiler does not immediately optimize newly generated code.

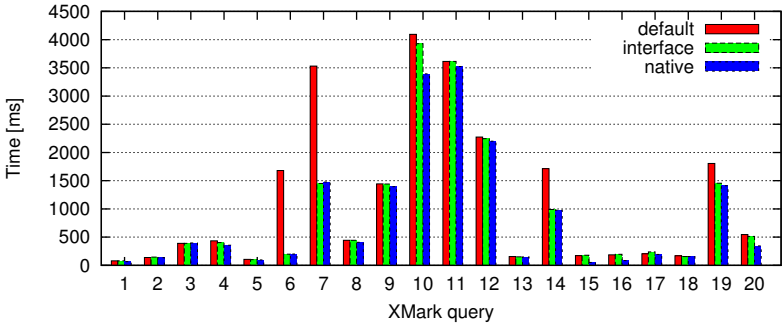
MXQuery delivered the slowest results and revealed serious performance problems even for simple queries. Our analysis confirmed that the system suffers from the overhead of a very fine-grained, token-based data representation [BBB+09].

8.3. XML Database Processing

In the second series of experiments, we evaluated the compiler inside a native XDBMS environment, which is a greater challenge than a normal main-memory setup. The compiled query plans must prove to handle larger data volumes and efficiently interact with external storage.

8.3.1. Access Optimization

To evaluate effectiveness of optimized access to disk-resident XML, we ran the XMark benchmark in BrackitDB for three configurations. In the first configuration (*default*), the queries are compiled in the same manner as in the main-memory setup, i.e., the XML tree was navigated with basic operations and all filter predicates were applied within the engine itself. In the second configuration (*interface*), the compiler makes use of an enhanced storage interface and translates individual path steps to dedicated navigation routines. For example, a path step `./person` is translated to a single storage call, which returns only matching element nodes for the given context node. At runtime, the storage then jointly evaluates navigation steps and node filter operations. In the third configuration (*native*), we extended this approach further and let the system perform storage-specific optimizations at compile time. Sin-



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
default	80	139	391	435	107	1680	3531	445	1444	4094
interface	81	146	389	405	101	198	1450	445	1442	3929
native	65	137	396	356	96	201	1478	408	1394	3395

	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20
default	3614	2273	157	1714	172	187	205	170	1806	545
interface	3614	2246	151	990	179	195	239	155	1455	512
native	3523	2200	148	979	52	87	194	157	1411	345

Figure 8.3.: XMark performance for different data access optimization levels on a 112MB document in BrackitDB.

gle navigation steps and sequences of child steps are directly compiled to native operations without the indirection of an adapter interface.

For the experiment, we stored the benchmark document with scale factor 1 (112MB) in BrackitDB and report the timings for warm buffers. Note that we did not create any supporting indexes on the document, as we intended to evaluate differences between compilation and optimization strategies for navigating XML.

The results of the experiment are shown in Figure 8.3. In comparison to the main-memory experiment, navigation on disk-resident XML is not only slower, the larger benchmark document also emphasizes the importance of data access locality. Accordingly, we observed a greater performance loss for those queries, which inspect larger parts of the document. In the *default* configuration, particularly queries with descendant axes (`Q6`, `Q7`, `Q14`) suffered from the expensive data access, because they scan large document regions. The join queries `Q8`-`Q12` suffered, too, because they access large parts of the document when

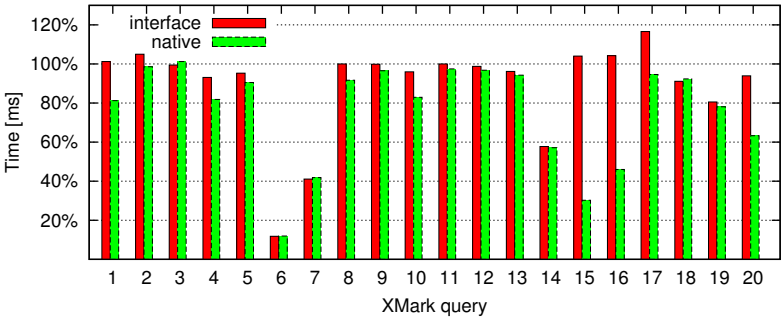


Figure 8.4.: Relative performance gain of data access optimizations.

loading the join tables. Higher response times due to poor data locality can be observed for `Q19`, because it sorts all data before accessing the storage again to construct the result.

It is not a surprise that the storage-optimized configurations outperformed the default configuration. However, the relative speed-up varies considerably between the queries. The greatest performance gain was achieved in queries with descendant axes (`Q6`, `Q7`, `Q14`) and long downward paths (`Q15`, `Q16`), which offer the highest potential for aggressive short-circuiting of navigation and filtering. In contrast, the join queries show only minor improvements, because the amount of data accessed and random accesses after joining the data dominate.

As Figure 8.4 shows, the difference between the optimizer configurations *interface* and *native* is relatively small. Only for the long paths in `Q15` and `Q16`, native compilation clearly outperformed adapter-based navigation and delivered the result more than two times faster. In this configuration, long paths were not evaluated by traversing the document tree node-wise for matching the child axis steps. Instead, the compiler directly leveraged the storage system of BrackitDB, which can evaluate paths alternatively as subtree scans with cheap candidate filtering. This option always pays off if a subtree scan is cheaper than matching a path node-wise. As built-in heuristic, the compiler performs this optimization for child paths with more than 6 axis steps.

In summary, this experiment proves feasibility and importance of compiler-optimized data access. But note that the results reported do not mark the optimum. Performance will be improved if content predicates are pushed down and evaluated directly during document navigation. Indexes and holistic twig pattern matching, as shown in Chapter 6, will lead to further improvements as similar attempts for the related XDBMS prototype XTC proved in [MHSB12]. However, note also that the more crucial set-oriented aspects including correlated nestings have already been optimized in the independent pipeline rewriting and join rewriting stages.

8.3.2. Scalability

We repeated the previous benchmark with the *native* configuration of BrackitDB for the scale factors 0.001-10 (1.2MB-11GB) to investigate the scalability of the system.

As the results in Figure 8.5 show, both the query plans and the system itself scale solidly. However, we observe that poor access locality has a notable effect on performance for large documents. In `Q19`, for example, the additional data accesses after the sorting step resulted in a lot of random I/O for the 11GB document. As a typical scenario for twig-like data access, our prototype will benefit from implementing the "piggy-backing" techniques described in Section 6.3.2. Note, the growth in response time for `Q11` and `Q12` is characteristic for the workload, because the result size grows quadratically with the size of the document.

8.3.3. Competitors

For comparing the quality of our compiler with dedicated XQuery compilers for persistent XML storage, we ran the XMark benchmark in scale factor 1 also for other XML(-enabled) database systems. Aside the general efficiency of query evaluation, this experiment gives insight into advantages and disadvantages of specific storage designs.

Like BrackitDB, eXist [Mei09] uses a prefix-based numbering scheme for encoding hierarchical structure and stores XML nodes with their prefix keys in a B^+-tree. Additionally, eXist generates default indexes

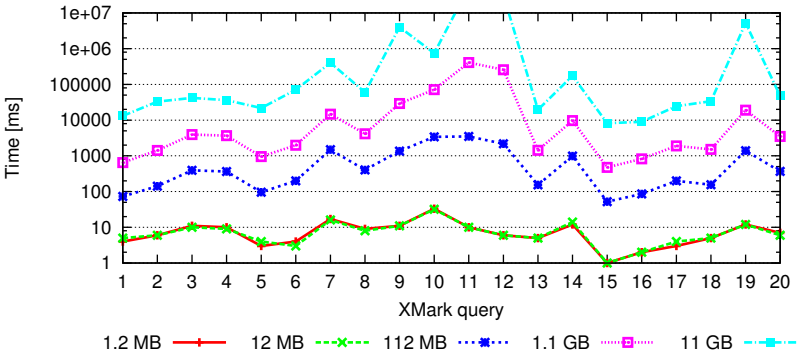


Figure 8.5.: Execution times of XMark benchmark for different scale factors in BrackitDB.

for accessing elements and attributes by name. In contrast to BrackitDB, however, it does not maintain a dynamic schema, which can be exploited, e.g., for complex path matching or indexing.

A lightweight alternative to prefix-based node labels are range-based numbering schemes [HHMW05]. Instead of addressing nodes with stable but variable-length keys, they use simple pairs of integers. However, because such labels are not stable, already small structural updates may require a re-labeling of large document regions, which is particularly expensive if these changes must be propagated to secondary indexes. Accordingly, storages which use such a labeling scheme are favorable only for read-intensive workloads. In our test field, two systems use a range-based encoding.

BaseX [BAS] stores XML nodes in an array-like sequential file, which is read page-wise from disk. The nested XML structure is encoded with the so-called *pre-post* numbering scheme [BGvK+05]. In the default configuration, BaseX creates a path summary for faster path resolution and additional content indexes for text nodes and attributes.

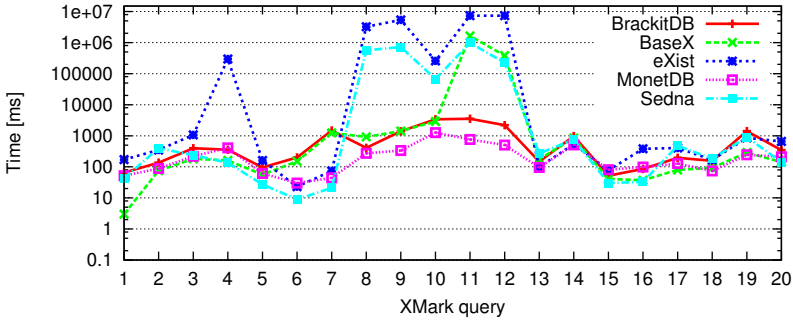
MonetDB [BGvK+06] uses a relational storage engine, which separates document structure and content in ternary tables. The node labeling scheme used is an enhanced variant of the pre-post numbering scheme, which reduces the re-labeling cost for structural document updates [BFG+06].

Sedna [FGK06] stores XML trees as linked structures, where each node maintains pointers to its siblings, its first child and its parent. For mapping this structure transparently to external storage, Sedna uses an intermediate layer, which translates virtual memory addresses to data blocks on disk. An additional metadata structure, which reflects the dynamic schema of a document, organizes nodes on the same path in chained data blocks. Like a path index for all paths in a document, it serves as efficient gateway for direct jumps into the document structure.

As in the previous experiment, we stored the benchmark document in each database, but did not perform manual tuning, e.g., by defining additional indexes. However, note again that some systems create indexes per default. As in the main memory benchmark, we adjusted the maximum heap size for the Java-based systems to 1.5GB, which was sufficient for each system to handle the workload. Figure 8.6 shows the fastest runtimes measured for each system. For comparison with BrackitDB, we repeat here the results for the *native* configuration.

As a result of the different storage designs, the general performance characteristic of each system is less homogeneous than in the main-memory experiment. Some systems achieve extraordinary fast results for certain queries, but this is often not the result of superior compilation techniques. Merely, those queries hit a sweet spot of the underlying XML storage. For example, BaseX can leverage its default attribute index to answer the point query `Q1`. Accordingly, it is more than one order of magnitude faster than all other systems, which need to search for the qualifying node in the document itself. Sedna outperforms all others in `Q6` and `Q7`, because its metadata structure provides direct access to all nodes, which qualify for paths with `descendant` steps.

The curve shapes indicate that the quality of our query plans is competitive throughout all queries. Recognizing and processing joins efficiently again proves to be an advantage over most competitors. Only MonetDB also handled the join queries without a drastic decrease in performance. For simpler queries, however, BrackitDB did not reach the best marks of the other systems. A closer look at the queries and the competitors reveals that query plan quality is here not the decisive factor. Merely, the other systems benefit from faster storage engines or had additional indexes available.



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
BrackitDB	65	137	396	356	96	201	1478	408	1394	3395
BaseX 6.7	3	75	181	161	56	150	1242	917	1427	2902
eXist 1.4.0	172	348	1071	2.9e5	160	23	74	3.2e6	5.3e6	2.5e5
MonetDB 1.1.11	52	88	220	409	61	30	44	273	334	1260
Sedna 3.4.66	43	398	240	137	27	9	21	5.6e5	7.2e5	66860

	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20
BrackitDB	3523	2200	148	979	52	87	194	157	1411	345
BaseX 6.7	1.6e6	3.8e5	176	841	41	37	80	93	288	142
Exist 1.4.0	7.3e6	7.3e6	99	540	78	380	405	168	855	660
MonetDB 1.1.11	760	501	96	508	81	98	127	74	244	210
Sedna 3.4.66	9.7e5	2.2e5	271	762	30	33	474	184	899	142

Figure 8.6.: Comparison of XMark performance for XML databases on a 112MB document in main memory.

For example, eXist was able to use the default element index for performing index-based path navigation, which is especially advantageous in queries with a descendant step (`Q6`, `Q7`, `Q14`). In turn, BrackitDB dominated eXist in queries which do not benefit from the availability of a basic structural index – even though both use a similar storage layout.

For BaseX, we observe the expected performance advantage of a range-based node labeling scheme. For all non-join queries aside `Q1`, where BaseX can leverage a content index, the query plans compiled by BaseX and BrackitDB are almost identical. But in the end, the lightweight comparison of fixed-size node labels in BaseX made the difference, because it allows for cheaper navigation logic. Note, with regard to our compilation approach, this result is not deciding. If we used our

compiler to evaluate queries on the same storage, we would benefit from the better performance in the same way.

MonetDB obviously pairs a highly efficient runtime with a capable query compiler. The cheap range-based labeling and the heavily tuned relational storage play well together. In conjunction with a specialized path processing operator [GvKT03] and join recognition, the system delivers consistently good results. For simple queries, Sedna achieves similar performance and sometimes even better results than MonetDB. However, Sedna is not able to process the joins in `Q8`-`Q12` efficiently.

A direct comparison of both systems with BrackitDB is problematic. First, they are implemented in C/C++ and therefore have numerous advantages with respect to I/O and memory management. Second, the entirely different storage layouts and, thus, different evaluation strategies make it hard to compare the quality of the query plans. MonetDB, for example, emits relational query plans and lives from raw scan performance, whereas Sedna profits from efficient path processing through the index-like metadata structure.

In the end, the ability to recognize and exploit joins remains the only aspect we can identify as a clear advantage of MonetDB and our prototype.

8.4. Relational Data

For showcasing efficiency and versatility of our compilation approach, we setup an experiment in which we processed regular relational data instead of semi-structured XML.

8.4.1. Workload

To model a realistic workload, we took the dataset of the relational decision support benchmark TPC-H [TPC12] and measured performance for XQuery versions of the SQL queries `Q2` and `Q6`. We chose these two because they represent typical classes of relational queries and challenge different aspects of a query engine. `Q6` is a simple filter and aggregation operation over a single table. `Q2` is a complex join query over 5 tables with a correlated subquery with another join of 4 tables.

In the queries, we modeled tables as sequences of structured record objects, where the column values can be accessed through the field access operator (=>). For reading the tables with correct data types, the custom XQuery function `parse-schema()` loads respective schema information from a separate XML configuration file. The function returns a virtual record object, which instantiates sequence abstractions for expressing table scans with `for` loops.

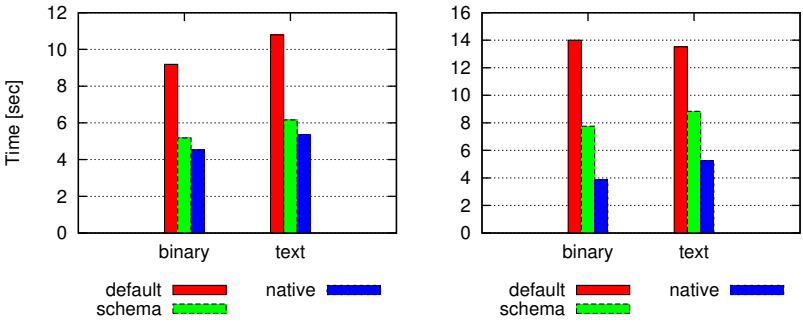
The queries were evaluated over normal files in which we stored the relational data. We tested two setups. In the first setup, we ran the queries directly on the '|'-separated text files generated by the *dbgen* tool of the benchmark. In the second setup, we stored the table data in files with a simple binary encoding.

The database generated for the measurements had a total size of 1GB (scale factor 1). The accessed subset of the database in query `Q2` is only about 140MB large, but because of many joins and a subquery, efficient evaluation is challenging. Query `Q6` scans the *lineitem* table (726MB raw data, ∼6Mio rows) and sums up the revenue of each qualified row.

8.4.2. Data Access Optimization

For processing regularly structured table data as fast as a relational system, the compiler must optimize column accesses and filter operations. Hence, we evaluated the compiler with the naïve configuration *default* and two optimized configurations *schema* and *native*. The configuration *schema* inspects at compile time, which columns are accessed by the query and propagates respective projection information as argument to the `parse-schema()` function. Furthermore, the compiler replaces dereference expressions for column accesses with cheaper positional access routines. The configuration *native* additionally extracts pipeline filters to evaluate them directly when scanning the base tables. The results are given in Figure 8.7.

As expected, the optimized configurations outperformed the standard configuration considerably. Configuration *schema* almost halved runtime for both queries through skipping unused columns when reading the input and through cheap positional column access. The additional push-down of predicates in the *native* configuration further improved performance. The scan-dominated query `Q6` ran almost two times faster than with configuration `schema`.



(a) TPCH Query 2 (b) TPCH Query 6

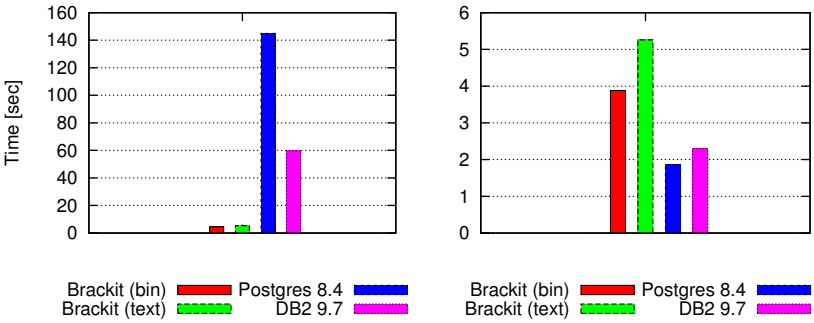
Figure 8.7.: Execution times of TPC-H benchmark queries `Q2` and `Q6`.

8.4.3. Comparison with RDBMS

To put the results of our prototype in relation to state-of-the-art systems, we measured query times for the SQL versions of `Q2` and `Q6` on the relational databases PostgreSQL 8.4 and DB2 9.7. To get comparable results, we did not configure supporting indexes, but created detailed table statistics and assigned sufficient memory to the database buffer and the queries to ensure that both systems created efficient plans and performed the computation entirely in memory. Neither system had to perform any disk access when running with warm buffers. Figure 8.7 shows the fastest results for each system.

The pictures of the two queries look very different. In the complex join query, our prototype is more than an order of magnitude faster than the relational systems. This clear result is caused by a better handling of the correlated subquery, which is performed in all systems using nested loops. Like both DBMS, our compiler generated a bushy operator tree with hash joins and a final sort. But in contrast to the relational systems, our prototype is able to reuse hash join tables in the nested query between iterations, as detailed in Section 5.2.3.

For `Q6` the relational systems are about 2 times faster than our prototype on binary files. However, this is the result of a more efficient scan-and-filter logic rather than the result of a superior query plan.



(a) TPCH Query 2 (b) TPCH Query 6

Figure 8.8.: System comparison for TPC-H queries `Q2` and `Q6`.

In summary, this experiment underlines the great utility of pairing a versatile data processing language like XQuery with a set-oriented, storage-independent runtime. With minimal effort, we can perform general data processing tasks on top of different data models and representations.

8.5. Parallel Processing

In our final measurement series, we evaluate the push-based operator model and the dynamic parallelization framework.

8.5.1. Workload

For assessing our parallelization framework, we took the data generator of the TPoX benchmark [NKS07] to setup a simple test scenario. Inspired by the original benchmark queries, we derived three basic query types, which allowed us to focus on the behavior of individual operators and for which we could easily investigate the influence of the partitioning scheme used.

The generated XML collections represent complex customer and order records in a financial application scenario. The data generator creates batches of 50,000 (∼320MB) and 500,000 (∼715MB) documents per file,

respectively. The advantage of this approach is that the documents in each batch can be read both sequentially and randomly. For random access, the data generator produces an additional metadata file per batch, which contains the offsets of document boundaries. For reading and parsing the documents from a batch file within XQuery, we provided respective input functions like `read-batch()`, `parse()`, and `parse-batch()`.

The default partitioning size for binding expressions was set to 1, but overridden within the queries with XQuery pragmas to hint the compiler to partition batch file sequences in chunks of 50 documents. The size of the worker pool was varied in each experiment. The maximum Java heap size was limited to 3GB.

8.5.2. Filter and Transform Query

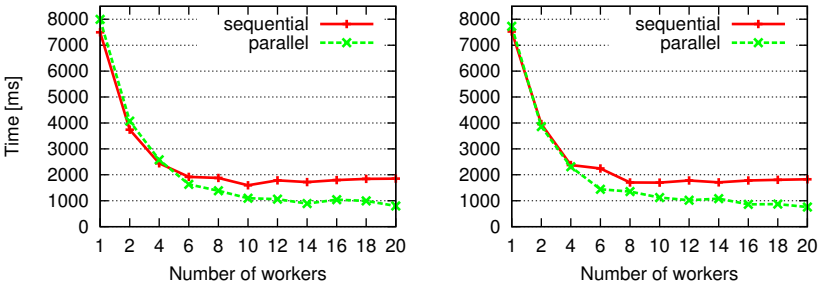
In the first experiment, we evaluated a simple filter query over a single customer batch file. In the outer `for` loop, the documents are read with the function `read-batch()` as plain strings from the batch file. Parsing and further processing takes places in the loop body.

For testing both partitioning schemes, we used two variants of the input function for reading the batch file. The lazy sequence returned by the first variant can only be partitioned sequentially, the lazy sequence returned by the second variant can be partitioned recursively with the divide-and-conquer pattern.

The queries were evaluated in both ordered mode and unordered mode, i.e., the `return` clause was compiled in the former case to an order-preserving chained sink and in the latter case to a concurrent sink.

Figure 8.9 shows the results for different worker pool sizes. All setups show a great scaling behavior with respect to the number of available workers. Because of the high selectivity of the query – only 0.02% of the documents in the batch matched the query predicate – the difference between ordered mode and unordered mode was negligible.

The query with sequential partitioning was only slightly slower than the one with divide-and-conquer partitioning. The reason for this good result is the deferred parsing of the documents within the loop body. Because the documents were read from the batch file only as cheap strings, sequential partitioning was relatively fast and led to good speedups.



(a) ordered (b) unordered

Figure 8.9.: Execution times of parallel filter query.

For comparison we repeated the experiment with a slightly modified query, in which the documents were directly parsed by the input function `parse-batch()`. The result in Figure 8.10 shows now clearly the advantage of divide-and-conquer partitioning. Because parsing as major cost driver in the query is now integral part of the sequential partitioning process, the parallel infrastructure cannot benefit from parallelism at all. The parallel partitioning scheme, however, still scales with the number of available workers as before, because expensive parsing only takes places when the partitions are small enough for sequential processing. This result underlines the importance of cheap and effective partitioning.

8.5.3. Group and Aggregate Query

The query in the second experiment groups the accounts of rich customers in a batch by nationality and aggregates the results. The rest of the setup is identical to the previous experiment.

The results in Figure 8.11 show again good speedups for larger worker pools. However, performance gains for sequential partitioning are not as good as in the filter query. The reason is a disadvantageous balance of partitioning cost and cost per loop iteration. Because this query does not contain a complex filter predicate, the load shifts towards the loop binding with the more expensive sequential partitioning. Note, the divide-and-conquer scheme is not affected by a cheaper loop body.

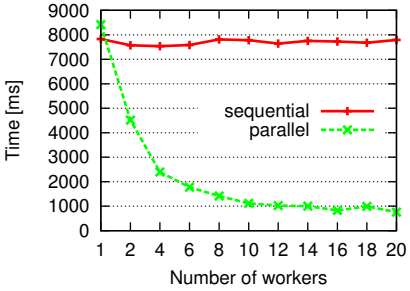
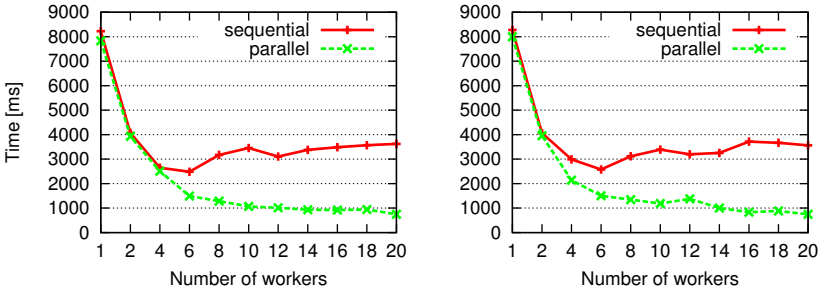


Figure 8.10.: Execution times of parallel filter query with expensive input binding.



(a) ordered (b) unordered

Figure 8.11.: Execution times of parallel group query.

.

8.5.4. Join Query

For evaluating join performance of the parallel operator model, we used a join query over an *order* and a *customer* batch. Both input batches are bound in nested `for` loops and combined in a `where` clause. The compiler recognizes the join semantics and compiles the query to a pipeline with join operator. The results are shown in Figure 8.12.

As in the previous experiments, the speedup for parallel partitioning is almost ideal and still strong for sequential partitioning. For the first time, we observe also some clear scalability problems when running a

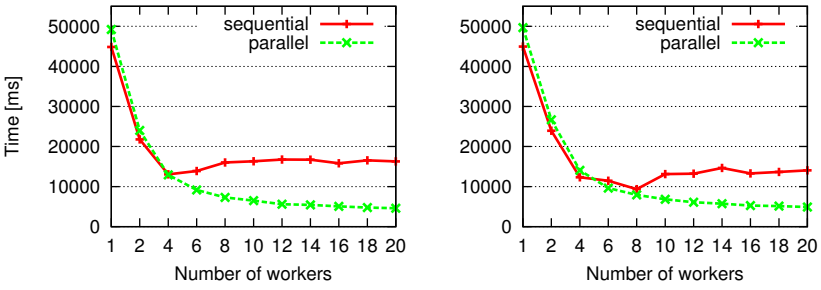


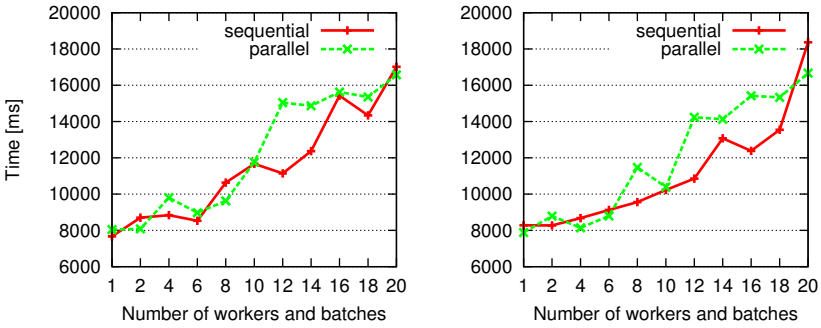
Figure 8.12.: Execution times of parallel join query.

query in ordered mode. As detailed in Section 7.3.4, evaluating a join in ordered mode requires the join operator to serialize the loading of the hash table. Accordingly, parallel performance degrades the more entries have to be loaded. The fact that we only observe a slowdown for the sequential partitioning indicates us that this scheduling pattern is more prone to disruptions, e.g., if workers are suspended during serialization.

8.5.5. Scalability

At the end of the TPoX measurements series, we evaluated the filter query and the group query again, but step-wise scaled both worker pool size and number of processed batches. Therefore, we nested the original queries inside a loop over the number of batches to be processed. For the sequential variant, this means that individual batches are still read sequentially, but multiple batches can be read in parallel. Note, we could not run this test for the join query, because the Java runtime used cannot efficiently manage applications which require more than 4GB heap.

As the results in Figure 8.13 show, the growth in input size was greatly compensated by the higher degree of parallelism. In the largest setting, the 20 times larger input resulted in only about 2.25 times longer execution times.



(a) Filter query (b) Group query

Figure 8.13.: Results for scaling worker pool size and input size.

In this experiment, the higher locality of sequential partitioning even outperformed divide-and-conquer partitioning for 12 parallel workers and more. With growing degrees of parallelism, more batches were read in parallel at multiple positions, which stressed the I/O subsystem and degraded performance of the file system buffer.

8.5.6. XMark Benchmark

As our final experiment, we repeated the XMark benchmark in the XDBMS setting with the push-based operator model and with different pool sizes. The results are shown in Figure 8.14.

Except for small deviations, we do not observe any effect through parallelization. The reason for this lies again in the partitioning of the `for`-bound input sequences. XMark evaluates solely path expressions as binding sequences, which can only be processed sequentially and usually navigate over large parts of the input document. Accordingly, a considerable fraction of total query time is spent for evaluating binding sequences and, hence, partitioning is too slow to achieve parallel speedup. To benefit from data parallelism in such single-document databases, a system needs efficient alternatives like a secondary index to evaluate binding sequences fast enough.

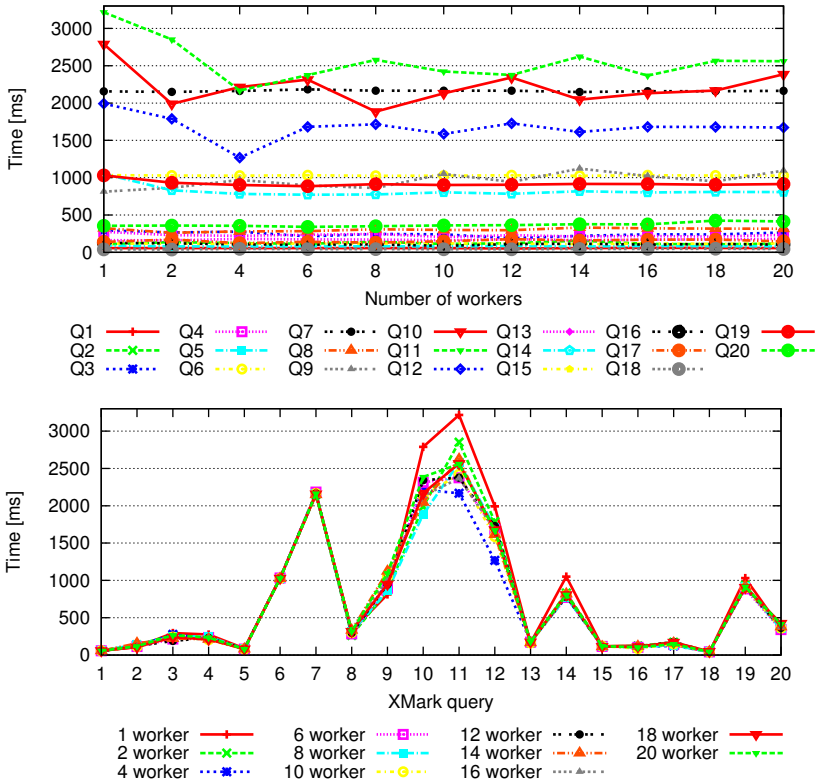


Figure 8.14.: Execution times of parallel XMark benchmark per query in BrackitDB on a 112MB document.

As positive result of this final test, we can state that the design goal of a low-overhead framework, which creates at least opportunities for parallel processing has been achieved. Even poorly parallelizable situations are not unnecessarily burdened with additional overhead by the parallel operator framework.

8.6. Evaluation Summary

The main observations from our empiric evaluation can be summarized as follows:

1. The compilation approach presented provides a solid basement for a complex real-world language like XQuery. The experiments with relational workloads proved also that declarative languages like SQL may be easily mapped to this infrastructure.

2. Our design is able to deliver competitive performance in various setups and on different storage platforms. In main memory, our prototype excels with fast compile times and a lightweight runtime. In disk-based settings, we depend on storage-specific optimizations to compete with specialized compilers. In several experiments, we could demonstrate the potential of such optimizations, which are simplified by the strict separation of query logic and data access routines.

3. Set-oriented optimization and join processing is and must always be a key concern in a query processor. Our approach successfully recognized and handled joins in all benchmark queries and, thus, outperformed most competitors in this discipline.

4. Physical optimization is an important building block to achieve high performance. We showed that we can support various kinds of storages and different levels of storage abstraction. The evaluated spectrum reaches from simple wrapper interfaces to native operations. Our results in the XDBMS context have shown that optimization efforts only pay off, if the granularity of operations and not the storage itself is the limiting factor.

5. Several experiments proved that our push-based operator model is able to process queries in parallel without additional assistance by the user or the compiler. We also demonstrated that our approach scales for both input size and number of available processors. The advantage of partitioning data dynamically in a divide-and-conquer fashion was evident in all experiments. However, our tests also showed that queries, which can read input only sequentially, greatly benefit from parallel processing if the sequential input can be accessed and partitioned efficiently.

9. Related Work

Considering the broad spectrum of topics touched by this thesis, it is not possible to narrow related work to a few areas. Therefore, the following concentrates on giving insight into the most important works, which pursue similar objectives or served as inspiration.

Because declarative, database-style query processing is the leading theme, we start with a brief walkthrough of query language history to emphasize the various connections to decades of database research. Thereafter, we present related query and data processing languages, and discuss other approaches to build flexible data processing platforms. Finally, we review related work in the field of XQuery compilation.

9.1. A Short History of Query Languages

The revolutionary simplicity of the relational data model, which was introduced in [Cod70], provided the ground for vivid research activity in the field of declarative query languages. In tandem with the development of SQL as front-end language, the relational model became a success story that lasts until today.

Very soon, practical reasons lead to first extensions of the initial relational algebra, e.g., to incorporate aggregation [Klu82, OO83]. More radically, several works introduced the so-called *nested relational algebra* to overcome the shortcomings of flat relations in 1NF [JS82, OO83, RKS88]. In essence, they complemented the relational algebra with *nest* and *unnest* operations for relation-valued columns.

Although nested relational algebra was never adopted in practice, its core ideas paved the way for object-relational systems [Dit86]. Inspired by the abstraction concepts of object-oriented programming, they introduced complex data types, object identity, inheritance, and polymorphism and integrated them into SQL-based or SQL-like query languages [RS87, CD88, LR89]. For the first time, application-level con-

cepts and abstractions could be directly modeled with custom data types in both database storage and queries.

In addition to the novel abstraction concepts for *structured entities* in the context of database systems, extensibility aspects have been another driving force in the object-relational movement [BBG+88, CH90, Gra94]. In a sense, object(-relational) databases made the first step towards a separation of query processing logic and physical data representation. Thereby, the concepts of nested relational algebra continuously contributed to optimization efforts of nested queries, which are typical for the navigation of linked structures [SPSW90, CM94, KM94].

Despite enormous research efforts and availability of commercial-class ORDBMS products, the object-relational "wave" ([SM95]) finally ebbed in favor of the simpler and more robust relational systems. But the next big trend was already in sight. New application fields emerged, which processed or produced large amounts of graph-structured data like in geographic applications, transportation and logistics, and biology [Woo12]. Because processing of arbitrarily structured graphs considerably differs from requirements in classic data management, the development of graph databases and query languages lead to decoupled research fields [AG08]. However, a special class of graph-like data – tree-like structures and hierarchical data – crystallized, spread in form of XML and other semi-structured formats, and became the new trend in database research [Bun97, Abi97, FLM98, W3C98b].

Besides various new challenges at the storage level, semi-structured data introduced several new aspects in querying processing. The flexibility aspects and the lack of a rigid schema required some form of tree pattern matching techniques to locate data of interest in hierarchical structures. Also, the need to construct and access nested structures raised again. Notably, most query languages still had a close similarity to the familiar `SELECT-FROM-WHERE` constructs of SQL [AQM+97, BFS00].

In the wake of the XML navigation language XPath [W3C99] and the XML query language Quilt [RCF00], which finally lead to the development of XQuery, tailored *tree algebras* were developed to better meet the requirements of querying collections of labeled, ordered tree structures [BT99, FSW00, JLST02, SA02]. Interestingly, none of them found its way from research platforms to commercial XML-DBMS. Instead, one pursued fairly different strategies for compiling XQuery/XPath in-

cluding nested relational algebra [BHKM05], extensions of the relational query graph model (QGM) [ÖSW08, Mat09], and purely relational encodings [GRT08].

At the language level, Quilt introduced a paradigm-shifting novelty: an explicit looping construct over variable-bound input sequences. The idea of nested loops was a step back from full declarative query processing, but it suited two major concerns perfectly. It allowed to draw and process collections of data from various sources and it enabled traversals within nested tree structures. The notion of a stream of variable-binding tuples within these loops bridged the gap to the efficient set-oriented techniques from the relational world.

9.2. Related Languages and Data Models

The common denominator of all proposals for declarative processing of object-like and semi-structured data is the representation of data as variations of tree or graph structures that is accompanied by some sort of navigation facilities [Abi97]. The star player in this field is certainly XQuery/XPath, which is also put in the center of this work and discussed in detail. In the following, we summarize other important approaches and systems of research and industry.

9.2.1. Lorel Query Language

The *Lore* (*Lightweight Object Repository*) project is a prototype DBMS for semi-structured data [MAG+97]. The underlying data model is the *Object Exchange Model*, short OEM, which was initially developed as a self-describing data interchange model for heterogeneous environments [PGmW95] and later slightly adjusted to facilitate its use in the *Lorel* query language [AQM+97].

An OEM object is a triple with the fields type, value, and object-ID. The type field indicates either an atom object like a string, an integer, or a floating point number, or a set object. The value field is the actual payload. The object-ID is a unique identifier of the object.

The set type is the composition type in OEM. It models complex structures by enumerating pairs of labels and object identifiers in the value of set-typed objects. Each pair represents a labeled edge to a

child object. The label carries the meaning of the relationship, e.g., an edge to a string object that is labeled "street" reflects the name of the street in a complex address record. Virtually, a set value in OEM is an unordered map. Array values are not supported, but might be emulated by a complex value with integer-labeled edges. Cyclic data structures are generally permitted, although typical use cases yield tree structures.

The query language supports set-oriented queries over OEM graphs in the classic `SELECT-FROM-WHERE` structure. Declarative updates are supported, too. A traversal over the edges of an OEM graph can be specified as a path, which can be a simple sequence of edge labels or a complex pattern with optional labels, regular expressions, and wildcards. A path can also specify label and object variables at individual paths steps. They are assigned to the matched OEM objects during a traversal and can be used as anchors for further traversals, e.g., in the `WHERE` or `SELECT` clause. The output of a query is an OEM instance, rooted under an artificial `answer` object.

9.2.2. UnQL

The *UnQL* query language [BFS00] uses structural recursion and pattern matching, two popular techniques in functional programming, to query tree-organized semi-structured data. Like the OEM model of the Lorel query language, UnQL represents data as unordered, edge-labeled trees, which consist of atomic and non-atomic nodes. In contrast to OEM, it does not treat nodes as identifiable objects, i.e., a non-atomic node is an anonymous set of label/tree pairs. Accordingly, UnQL specifies equality solely by value.

Queries are written as `select-where` constructs. The `where` clause specifies a structural query pattern, which is translated into an algebra of function compositions, which use argument pattern matching to perform a recursive traversal through the tree structure. The query pattern can contain variable bindings, which can be referred to in the `select` clause to construct the result.

The `where` clause may consist of multiple patterns, which allows to specify joins. Furthermore, UnQL supports nested queries within the `select` clause to group results or to express outer join semantics.

The structural recursion in UnQL can be evaluated as a standard top-down recursion, which is equivalent to a nested-loops-like tree traversal. Alternatively, the recursion pattern can be transformed to bulk operations by using dependent joins [Flo99, MHM04], which allows to perform set-oriented optimization [BFS00].

9.2.3. TQL

TQL [CG04] takes a unique approach to querying semi-structured data. It is also a query language for edge-labeled, unordered trees, called *information trees*. It is a logic-based language and, thus, it contrasts to most SQL-influenced proposals with a quite different query syntax.

A query has the form `from Q |= A select Q'` and is interpreted as a matching of the subject `Q` (the data) against the formula `A`, which yields a set of variable bindings, for which the parameterized result expression `Q'` is evaluated. The result of the query is the concatenation of the results of all evaluations of `Q'`.

A formula describes a query pattern, which consists of structural and first-order logic operators. The former are matched against the information tree, the latter serve as combinators and quantifiers for variables. Wildcard patterns and a recursion operator allow to match complex patterns in the tree at arbitrary depths. With the universal quantifer `foreach`, TQL provides a similar feature like XQuery's `for` loops. However, the construct is limited to the tree pattern logic.

Details about the implementation of TQL are given in [CFG02], but information about the efficiency of the fully algebraic compilation is not available.

9.2.4. Object Query Language (ODMG)

Query languages for object(-relational) structures have a close affinity to semi-structured data. Perhaps the most influential one is the *Object Query Language* (*OQL*), which was standardized by the Object Data Management Group (ODMG) [Coo97]. The OQL is defined for graphs of typed, class-based objects as they are common in object-oriented languages. Supported composition types for collection-based object properties are Array, Set, Bag, and List.

The syntax of OQL leans very close on SQL and supports almost the same expressiveness for quantification (e.g., exists), grouping, sorting, and aggregation. The result of a query is a collection of objects.

Navigation within the object graph is equivalent to dereference operations in a normal programming language. In contrast to languages for semi-structured data, however, it is not possible to match complex path patterns with wildcards and variable-length paths.

Although OQL turned out to be too complex to become ever implemented, it widely influenced efforts in the object-relational world. Many ideas served as blueprint for the object-relational extensions of SQL [SQL99] and a great variety of O/R mapping middleware.

9.2.5. Rule-based Object Query Language

The work in [SR92] presents a rule-based query language – also entitled as *OQL* – and an algebra for object-oriented databases. Objects are identifiable instances of classes, which are defined as composites of atomic values and other composites. Supported composition types are Tuple, Set, and List, for which distinct null values (nulltuple, nullset, nulllist) exist.

The algebra defines six sets of operators for working with objects, tuples, lists, and sets, and for list/set conversion. Further, it defines an operator *ASSIGN* for assigning an object to a variable and an operator *REPEAT_UNTIL* for evaluating recursive queries.

A query is a sequence of statements, which are either variable assignments or rules. A rule consist of a head and a body, which consists of a set of generators and qualifiers that implement the matching logic. Navigation in the object graph is performed with path expressions. Nesting and unnesting for set-valued types are performed explicitly via rules.

A direct relationship between practically relevant set-oriented concepts, algorithms, and optimization principles does not exist. In this sense, the whole nature of the language is similar to Datalog [SKS06].

9.2.6. SQL:1999 and SQL:2003

With the SQL revisions SQL:1999 [SQL99] and SQL:2003 [SQL03], the former pure relational standard was equipped with plenty new options for representing and querying structured and semi-structured data.

It introduced a rich set of object-relational features like structured data types, reference types, typed tables, and table hierarchies, which come close to many features of ODMG's OQL model, but are less general and easier to implement.

SQL:1999 added the anonymous collection types *ROW* and *ARRAY*, which made multi-valued columns possible, but could not be nested to form arbitrary nestings as required for semi-structured data. SQL:2003 added the collection type *MULTISET* and finally allowed arbitrary nestings of all collection types [Tü03].

The counterpart to collection type constructors is the table-valued function `UNNEST`, which allows, e.g., to use collection values in the `FROM` clause of a `SELECT` statement. As of today, however, many DBMS only provide very limited support for collection types if any.

The third novelty of SQL:1999 found much better adoption in commercial systems: the new data type XML and the language extension SQL/XML. They enabled support for hybrid query processing over both relational and XML data. Queries can be formulated in either SQL or XQuery; data in the respective "foreign" format can be queried through special mapping functions and data constructors. For these new features, most DBMS manufacturers developed tailored XML storages with extensive indexing support and other DBMS-class features like transactions, etc. [NvdL05, Rys05, LKA05].

9.3. Data Processing Languages

Data processing techniques increasingly find applications outside the traditional database context. New extensions and APIs for general-purpose programming languages give the application logic direct access to the efficient, set-oriented infrastructure. A second, very active community creates dedicated data processing languages as an abstraction layer for the MapReduce framework and other platforms for distributed, large-scale data processing [IBY+07, DG08, STL11]. Such languages naturally operate beyond the narrow "closed system assumption" of a database query language and embrace semi-structured data from various sources. Therefore, and because of their scripting capabilities, they are especially interesting for this thesis. In the following, we particularly spotlight the language features of the various approaches.

9.3.1. JSON and Jaql

Jaql [BEG+11] is a declarative scripting language targeting at large-scale data analysis in the Hadoop MapReduce framework [Whi09]. It builds upon an extension of the JSON data model, i.e., it includes structured records and arrays and a variety of common atomic data types. Furthermore, Jaql supports functions as data type, which is of essential utility in the operational model of the language.

A Jaql script is a sequence of statements like library imports, variable assignments, and expressions. The set of supported expression types is relatively small. It comprises Boolean logic, arithmetic, conditional branching, function calls, and data constructors. The latter include various forms of path expressions, for accessing and projecting values of from arrays and records. Flexible (path) pattern matching for semi-structured data as, e.g., in XQuery/XPath, is not supported.

The strength of Jaql is a composable set of array-based, higher-order functions, called operators, which implement bulk processing logic. For a convenient chaining of operators, the language provides a special pipe operator -> as syntactical variant of normal function composition. The pipe operator visualizes the data flow from one operator to the other. Built-in are streaming operators like `FILTER`, `TRANSFORM` (implements a functional map operation), and `EXPAND` (implements unnesting), the blocking operators `SORT` and `GROUP`, and a `JOIN` operator. Because extensibility is a major concern, Jaql operators are higher-order functions that can be customized by user-defined code, e.g., a comparison function. Furthermore, Jaql can be extended with custom operators.

Because the language is intended for data processing in MapReduce clusters, it typically operates on huge collections of relatively small data items, which reside in distributed file systems or key/value stores. They are loaded and converted into the internal data model via customizable I/O adapters.

Although Jaql is purposely not a declarative query language in the classical sense – the user wires the query operators by itself – the compiler can leverage knowledge about built-in operators and perform (simple) optimizations like predicate pushdown, etc. If available, full or partial schema information can also be leveraged during optimization.

The compiler's main concern is the identification of opportunities to process parts of a script in the parallel MapReduce environment. Therefore, it identifies operations, which can be applied independently over partitions of the input collection (e.g., `TRANSFORM`), or which perform a distributable aggregation (e.g., inside `GROUP`). These are compiled to dedicated Jaql functions, which can be manually optimized by the developer (source-to-source translation) or directly deployed as Map and Reduce functions in the distributed framework.

9.3.2. Pig Latin

Pig Latin [ORS+08] is a high-level language for MapReduce clusters, which is positioned in between the fully declarative world of SQL and the low-level, procedural world of manual Map and Reduce functions. A Pig Latin program is compiled into MapReduce jobs, which are carried out in a Hadoop cluster [Whi09].

The data model of Pig Latin consists of the four types: atom, tuple, bag, and map. Collection types can be arbitrarily nested; a bag is considered as a collection of tuples. The supported expression types are constants, conditionals, field/map access and projection, function calls, and flattening of nested collections.

A program is a hand-crafted data flow graph of typical query operations. The data flow is assembled as a sequence of statements, which perform an operation, and bind the result to a variable. Variable references in subsequent statements implement the actual pipelining of intermediate results.

At the beginning, input data is loaded with the `LOAD` command, which references a bag-structured input collection and, optionally, specifies its schema. Afterwards, the bag is processed with various bulk commands like `FILTER`, `FOREACH`, `COGROUP`, `JOIN`, `ORDER`, etc. All of them are of higher-order nature and are parameterized by expressions or functions. For nested data, `FOREACH` supports a limited form of nested bulk processing (e.g., filtering and sorting), but only at the direct child level. Processing of arbitrarily nested semi-structured data or advanced structural pattern matching is not possible.

The execution of commands, i.e., the actual data processing, is deferred until a `STORE` command occurs. The lazy approach enables pipelining and other cross-command optimizations like early filtering.

Pig, the runtime of Pig Latin, creates a logical plan for the commands that operate on every referenced bag. If the resulting bag is finally stored with `STORE`, the logical plan is compiled to a sequence of MapReduce jobs, which is executed in the cluster. Thereby, the compiler puts special care on the effective mapping of groupings and aggregations to the MapReduce framework.

9.3.3. LinQ

LINQ (*Language-Integrated Query*) [MBB06] is a query abstraction for data processing within the Microsoft .NET framework. It introduces standard patterns for querying and updating data, which can be virtually mapped to almost any kind of data – independent of whether the "database" is an in-memory collection of objects, a relational SQL database, or an XML document. Therefore, the framework blends the programming language view on a collection type, e.g., a simple list, with a database view considering a list as a queryable data source. At the inner core, LINQ queries embody many ideas of functional programming like lazy evaluation and monad comprehensions [Mei07] and wrap them in the object model of the .NET runtime.

The SQL-like queries are first-class citizens in the .NET languages C# and Visual Basic and are carried out by the backing *LINQ Provider* of the respective collection. In case of a simple in-memory list, the provider processes the query directly on the list; in case of a relational database, the provider generates SQL, sends the query on the database server, and makes the result available to the .NET program as a collection value.

Conceptually, LINQ is data-model agnostic in the sense that query operations (`select`, `where`, `orderby`, etc.) are performed on collections of any kind of value. To overcome the heterogeneity of data models, it makes use of the properties of an object representation of the data. A list of point objects, for example, can be filtered with `where point.x > 3`. The query capability for a semi-structured format like XML is obtained by representing it as a DOM-like tree structure [W3C98a]. With this object abstraction, it is possible to expose structural relationships as collection-valued methods and use them to perform flexible path pattern matching within queries. For example, an XPath expression like `$docs//book` over an XML resource can be evaluated in LINQ as `docs.Descendants("book")`.

The actual query execution through a LINQ provider can be realized in various forms. In the most simple case, a custom provider exposes the queried resource as a collection value and uses the default, in-memory query logic. More complex but also more efficient solutions can translate, compile, and execute (parts of) the actual query expression tree by themselves. This form of flexible consideration of querying capabilities is comparable to the integration strategies of heterogeneous database systems [CHS+95].

9.3.4. Database-backed Programming Languages

Several approaches pursue the idea of delegating data-intensive tasks transparently from the program to external DBMS.

`Kleisli` [Won98] is a data integration system that has been developed for the special requirements of bioinformatics. Kleisli hosts a high-level functional query language, called `CPL` (Collection Programming Language), which focuses on database-backed processing of "bulk" data types. To accommodate the needs of data representation in bioinformatics, the data model of CPL supports arbitrarily nested records, sets, lists, bags, and variants thereof. A CPL query uses a comprehension syntax [Wad90], which allows to filter, unnest, and transform the input. The comprehension is translated to Nested Relational Calculus [BNTW95], algebraically optimized, and then send to an external DBMS for processing.

`Links` [CLWY06] is a programming language for web applications, which compiles to both client-side JavaScript code and server side SQL code. It also uses a comprehension syntax to compile data access operations into SQL code. In contrast to Kleisli, however, it supports only the flat relational model.

`Ferry` [GMRS09] is compilation framework, which is able to offload data-intensive computations to a relational database. Like Kleisli, it supports arbitrarily nested tuples and lists, but also puts considerable effort to reduce the number of queries that are generated for queries over nested structures. Therefore, Ferry uses a lifting technique, which was pioneered in the relational XQuery processor Pathfinder for querying nested XML structures [GRT08]. Ferry features its own research language, but has also been integrated into Ruby and in a library for Haskell [GGSW11].

9.4. Extensible Data Processing Platforms

Various contributions before already aimed at building a versatile and reusable data processing platform. Basically, they can be distinguished as compiler-centric approaches and system-centric approaches. The former pursue an extensible compiler infrastructure, which can be customized for new kinds of data and operations. The latter design the database platform as a kind of virtual machine to which front-end languages and data models are mapped.

9.4.1. Compiler Infrastructures

Extensible compiler infrastructures have been studied in the context of extended relational systems and object-based systems. Three of the most influential works are presented in the following; other major approaches can be found in [Bat88, CDG+90, SRH90, SPSW90, VB96].

Starburst

The *Starburst* project [HFLP89] introduced a new compiler model for extending classic relational DBMS technology. The featured SQL extension builds on table expressions as building blocks, which can represent any kind of table-structured data source like base tables, view definitions, function applications, etc. Queries are internally represented in the *query graph model* (QGM) as directed graphs of high-level query operators, which are translated into physical operators for execution. A single node in the logical query graph can thereby reflect multiple physical operations at once, e.g., a scan, a projection, and a join. QGM has been recognized as particularly useful for join enumeration and other optimization techniques, but also offers an extension point for new operators. QGM was adopted by the relational DBMS DB2, where it primarily drives relational workloads, but also successfully serves as basis for object-relational and XML extensions.

Garlic

Garlic [RAH+96, CHS+95] is a data integration middleware, which provides an SQL-based query interface for heterogeneous and distributed

data sources, which supports path expressions, nested collections, and methods. The backbone is a flexible wrapper architecture, which encapsulates all data sources behind object-based interfaces. The specialty of Garlic is the flexible offloading of query functionality to individual wrappers, without the need to know the data sources themselves. Instead, all wrappers involved are identified and then participate directly in the query planning process. Queries over multiple data sources are decomposed and, depending on the capabilities of the wrapper implementation, delegated to the sources. Orchestration and lacking functionality is handled by the Garlic query processor itself. The optimizer generates alternative query plans, by asking each wrapper, which share of the query plan it can handle and at which cost. The final plan is the combination with the cheapest cost.

Volcano

The *Volcano* system [Gra94] is a data flow query processing system, which abstracts all physical operators as pull-based iterators with the open-next-close protocol. Queries are translated from a logical algebra, e.g., relational algebra, to Volcano's physical algebra, which implements standard query processing algorithms like filters and joins. Set-oriented processing and interpretation of data is strictly separated by parameterizing iterators for support functions, which implement the actual data access logic. The framework provides extensibility in various dimensions. New data models are introduced as abstract data type and new support functions. New query capabilities or access methods are introduced by implementing the iterator model.

9.4.2. Database Languages

Database languages pursue a two-step compilation approach for realizing flexible data processing platforms. A front-end compiler logically maps the data model and query constructs of a concrete language like SQL to a generic storage and query infrastructure, where the database language compiler performs the physical optimization for the mapped query. Naturally, performance directly correlates with the capabilities of the storage and the quality of data mapping.

Without going into technical details, the following shortly discusses two approaches, which base on radically different physical layouts. Further noteworthy approaches can be found in [Gü89] and [OBBt89].

MIL

MIL [BK99] is an intermediate language for building read-intensive query processing systems on top of the extensible database system Monet. It is designed for relational and object-oriented applications and uses a fragmented data model, i.e., MIL operates on binary tables (BAT) of primitive data types only. This allows for high scan performance by clustering related attributes, but requires the MIL program to reconstruct related data by itself. For reading a structured record, e.g., the BATs containing the respective attribute values must be joined. However, respective equi joins are relatively cheap, because MIL is designed for memory-resident data and can perform them often in an array-like fashion with positional lookups in fixed-length BATs.

The language forms an algebra of side-effect-free bulk operators which cover standard query constructs like filter, join, aggregation, and grouping. A MIL program is a script of bulk operator statements, basic control structures, and variable bindings for intermediate results. A script is typically generated by the compiler for the front-end language, which logically optimizes the query and decides about the execution order of the algebraic BAT operators ("query strategy"). The MIL interpreter itself analyzes the script and choses suitable algorithms implementing them ("tactics"). Therefore, the system maintains sets of properties for BATs, which help to exploit physical properties like orderings. Multithreaded parallelism is supported in two ways. Statements with parallelizable BAT operators can be annotated with the desired degree of parallelism. Parallel execution of multiple statements can be specified by putting them inside parallel blocks.

MIL and the heavily optimized Monet infrastructure has been shown to achieve high performance for relational analytic workloads. In tandem with the Pathfinder compiler (see 9.5.2), it also proved to be highly efficient for running XQuery logic, too.

FAD

FAD [DV92] is the strongly-typed interface language for the parallel database system Bubba [BAC+90]. It is a declarative language for transient and persistent collections of sets, tuples, and structured objects, which closely resembles the expression-style of functional general-purpose languages.

A FAD program consists of a composition of actions, which are distinguished in basic actions like value construction and selection, and higher-order actions for implementing control logic and bulk logic. A *let* construct binds intermediate results to variables, explicit looping constructs and operations like *filter*, *pump*, and *group* provide the tool set for bulk processing. Data items have an identity and side-effecting updates are allowed in some, but not all action types. For type safety, proper definition of all data types is required.

Aside translating a program to operations of the parallel infrastructure, the compiler can optimize distributed data access. Higher-level algebraic optimization is not performed.

9.5. XQuery Compiler

The XQuery compilers available considerably differ in functionality, performance, and scalability, because they are designed for special use cases and target platforms. The spectrum reaches from lightweight stand-alone solutions for main-memory data to embedded compilers for database-resident data, and specialized solutions for full-text search, streaming applications, etc.

The character of a compiler can be distinguished as either iterative or set-oriented. Iterative compilers follow closely the XQuery standard and evaluate queries in the normal nested-loops fashion. Set-oriented compilers like the one presented in this thesis, translate queries first into a more general plan representation or algebra to compute independent subexpressions and multiple iterations with more efficient operations, in a different order, or even in parallel. An overview of XQuery processing models is presented in [BBB+09].

9.5.1. Iterative Compilers

Nested, iterator-based evaluation is wide-spread in main-memory engines, which focus on fast XML transformation and data extraction. The direct compilation of expressions is simple but quick and easily integrates all new programming features of XQuery 3.0 like function items and partial function application. Because most compilers represent data closely to the XQuery standard as sequences of items, they can run on top of different data layouts.

The dynamic context is usually represented as a mutable set of variables. It is simple and naturally fits the processing model. Some implementations use a global context object, whereas others split the context and store bindings locally in the scope of the corresponding sequence iterator. In this case, the local scopes must be recursively passed on to subexpressions and the entire dynamic context turns into a hierarchical leaf-to-root composite.

The disadvantage of the mutable dynamic context is an implied sequential evaluation order of context-dependent expressions. It requires special treatment in individual situations to perform common optimizations like parallel computation of (partial) results or application of an efficient join algorithm instead of computing and filtering the Cartesian product. As a result, processors often suffer from nested-loops semantics and poor scalability. Because of its simplicity, it is, nevertheless, the most common design used in XQuery processors for main memory today [Kay08, Mei09, BBB+09, Grü10].

9.5.2. Set-oriented Compilers

Set-oriented compilers are prevalent in XML-enabled database backends and excel with scalability and extensive XML index support. However, most of them require data in a specialized internal layout or lack support for certain language features.

Compilers for native XML storages are often tailored for a particular data representation and XML node labeling scheme [HHMW07, OOP+04]. As a result, the optimizer is often faced with special operators and must cope with unusual query plan shapes, or sophisticated dependencies between operators.

Compilers for relational target platforms suffer from the complexity of XQuery and need elaborate concepts to map queries and data to relational algorithms and data layouts. Relational thinking requires them to radically rewrite the whole query, which severely complicates the realization of language concepts beyond plain "Select-Project-Join" and requires an advanced optimizer to compile efficient query plans.

In both types of systems, the programming aspects of XQuery like user-defined functions, recursion, and higher-order functions usually fall behind. They just do not fit into the picture of database-style processing. This disqualifies such systems as runtime environment for general data programming tasks. Nevertheless, users often accept partial language conformance as long as a system meets their compatibility and performance needs. On the flip side, they cannot take advantage of one of XQuery's biggest strengths – the ability to represent, interpret, and process different kinds of data from diverse sources.

In the following, we summarize the key aspects of the most prominent approaches.

Pathfinder

Pathfinder [GRT08] is a compiler for relational backends which requires to have all data, i.e., item values, sequences, and entire XML documents, encoded in a ternary table layout. Variable bindings and iteration scopes are *loop-lifted* to turn nested-loops into operations on "unrolled" tables. A specialized join operator speeds up XML processing [GvKT03]. Pathfinder is especially successful on top of the MonetDB backend [BGvK+06] because the ternary table layout and the frequent equi-join operations for loop-lifting suit the system's infrastructure. However, the strict relational view complicates the sometimes subtle semantics of XQuery (e.g., in comparisons) as well as its functional aspects. Furthermore, loop-lifting causes query plans to quickly grow in size and complexity [GMR09].

XTC

XQGM is the logical, tuple-based operator graph representation of a query in XTC [Mat09]. It is based on the relational query graph model (QGM) [HFLP89] and introduces so-called correlated edges, which indi-

cate context dependencies, i.e., nestings, between operators. Extensive unnesting rules eliminate these correlations and apply independent join operations instead. Special consideration is thereby given to opportunities for twig join processing. After logical optimization, a great variety of physical operator alternatives and evaluation strategies is available to compile XQGM into an executable plan. Query rewriting and optimization within XQGM is a non-trivial task, because the correlated edges turn the operator tree into a directed graph.

NAL

NAL [MHM04] is an algebra of tuple-based operators, which accommodate XQuery's FLWOR bindings. They consume and produce nested tuples forming sequences of sets of variable bindings. After translating a query into algebraic form, the compiler applies a set of equivalence rules to rewrite nested queries to more efficient set-oriented constructs. In principle, the approach is applicable to any storage, but even without considering physical properties, the pattern matching for the rewritings is quite complex. Furthermore, the approach assumes nested tuples, which generally complicates variable handling during compilation and the implementation of efficient physical operators.

Galax

Galax [RSF06] comes with a feature-complete XQuery algebra, which distinguishes between XML operators, tuple operators, and a third group of explicit boundary operators, which connect the two other parts of the algebra. Similar to the compiler developed in this thesis, expression trees are compiled directly and FLWORs are compiled to tuple operator trees. Nestings are modeled as dependent join operations. Galax also supports basic optimizations for unnesting and value-based joins. Further optimizations are not mentioned.

IBM pureXML

IBM's *pureXML* extends the QGM-based query representation of the relational database DB2 for joint processing of XML and relational data [ÖSW08]. FLWORs are modeled in the set-oriented QGM by combinations of new `ForEach` quantifier types and existing table-valued

operators like SELECT. All other expression types are realized as conventional scalar functions. XML processing is widely delegated to the native storage in form of path matching routines. Although the mature optimizer can be reused, several minor and major rewritings are still necessary to yield efficient XQuery. Furthermore, the compiler heavily depends on the advanced storage engine to pushdown complex path patterns.

10. Summary and Future Work

This thesis presented a complete walk-through of a retargetable compiler infrastructure and runtime platform for processing structured and semi-structured data.

Starting from a specification of basic ingredients for declarative query processing and scripting, we derived a compiler around a top-down query representation, which naturally models variable bindings and nested evaluation scopes. We obtained a hybrid design based on plain expression trees with integrated operator pipelines. It localizes expensive bulk sections in a query and allows to optimize and compile them separated from remaining aspects like data access operations, functions, and arithmetics.

The concept developed is fully composable, i.e., query nestings may be arbitrarily deep and rewrite rules are independent of the surrounding expression. Furthermore, the approach is not limited to certain query constructs and custom functionality can be plugged in flexibly as functions, custom expression types and even custom operators.

On top of the query representation, we presented the realization of proven set-oriented optimization techniques, which can be reduced to algebraic transformations in the functional monadic core.

For efficient data handling at runtime, we demonstrated compilation strategies and extension points for tailoring the compilation process to a specific data store.

For exploiting the capabilities of modern multi-core and many-core architectures, we introduced a novel push-based operator model, which is capable to dynamically parallelize query sections. We gave detailed insight into effective data partitioning and load balancing mechanisms as well as the realization of the most important types of query operators.

Finally, we empirically evaluated all facets of the approach for different data sets and in various settings.

10.1. Conclusions

In the rear-view mirror, the concept of a data programming language as proposed in this thesis is a composition of ideas, which originate from all epochs in query processing. It borrows the functional scripting-style and the nested-loops-based query logic from XQuery/Quilt, but strips it from the narrow focus on XML[1]. Instead, it picks up the extensibility idea from the object-relational world and uses the skeleton of looping constructs, variable bindings, and tuple streams for bulk processing of any kind of structured and semi-structured data.

At the data level, we embrace the idea of abstract data types from object-relational settings, because it implies the encapsulation of data access routines for system-internal optimization. However, we abstain from other "golden rules" ([ABD+89]) like object identity, persistence, and concurrency, because they have proven to be sources of inefficiency (e.g., cyclic data structures) or to be inappropriate in the context of a query language. At the same time, we emphasize the need for schema-less structural pattern matching in semi-structured data. Because of the latter, the data model builds on tree-structured compositions of array and map structures, which subsumes the representation of structured data as well.

Blending so many ideas in a single concept always embodies the danger of inefficiency through too many abstraction layers. Therefore, we oriented ourselves at the lesson learned from the success of the relational model – simplicity is key – and imported as much practical knowledge about relational query processing as possible. Furthermore, we aimed at a simple compilation and evaluation process, based bulk processing on simple function composition, and stepped back from attempts to implement a universal but complex (nested) algebraic solution.

In the end, the concepts developed and orchestrated in this thesis proved to be sustainable to carry our vision of building a compiler and runtime kernel, which integrates the best-of-breed concepts from decades of data management and can serve as solid basement for the rapid development of tailored query and data processing systems.

[1] As a good play, the Quilt grammar itself embodied *XPathExpression* as terminal symbol [RCF00].

10.2. Outlook and Future Work

As usual in a thesis, it is not possible to exhaustively cover and discuss all topics addressed or touched. Even though the corner stones for the compiler and the parallel query runtime have been laid in this work, a wide field of topics is left open for future research and improvements. Many opportunities for improvements will concretize from practical use, refinement, and evolution of the prototype developed, but others deserve thorough consideration in theoretic contexts as well.

Traditionally, query optimization provides the largest playground for additional work. At the logical level, state-of-the-art algorithms like join enumeration, statistics, cost-based query optimization have to be ported and evaluated. Interesting is also the examination of how structures and idioms of concrete front-end languages interact with applicability and effectiveness of optimization rules. Likely, one will identify sets of universally applicable and specific optimizations, which can be bundled to optimization profiles for different languages and environments.

At the physical level, this work only scratched at the surface of possible optimizations. Aside smart tracing and mapping routines for particular classes of storages, plenty of challenges are worth to look at. In the days of cache-optimized storage structures and algorithms, for example, a portable compiler framework needs to go new ways for augmenting query plans with additional information for exploiting and optimizing data locality. Where current relational systems can rely on fixed-size tuple structures and metadata, more powerful concepts are needed to model and trace data sizes and locality within the compilation process. Efficient handling of variable-sized, tree-structured data and interaction with external storage in query operators is also important for practical use cases.

Regarding the parallelizing operator model, there are also opportunities for improvements. Although parallelization is performed autonomously, it still relies on predefined thresholds for the partitioning step and buffer management to keep parallelism and memory consumption under control. This aspect is open for consideration of dynamic approaches from the field of self-tuning systems. Improved scheduling mechanisms may take the overall progress and memory requirements of a query into account. Last but not least, scalability demands suggest to extend capabilities for automatized parallelism to distributed platforms.

A. Translation of Operator Pipelines

Listing 17 Translation of operator pipeline.

```
 1: function COMPILE_PIPE_EXPRESSION(ast)
 2:     operator ← child(0, ast)
 3:     end ← create_end()
 4:     start ← compile_operator(operator, end)
 5:     expr ← create_pipe_expression(start, end)
 6:     return expr
 7: end function

 8: function COMPILE_OPERATOR(ast, end)
 9:     if ast is START then
10:         op ← compile_start(ast, end)
11:     else if ast is END then
12:         e ← child(0, ast)
13:         expr ← compile_expression(ast)
14:         set_expression(end, expr)
15:         op ← end
16:     else if ast is FORBIND then
17:         op ← compile_forbind(ast, end)
18:     else if ... then
19:         ...
20:     end if
21:     return op
22: end function
```

Listing 18 Translation of ForBind operator.

```
1: function COMPILE_FORBIND(ast, end)
2:     var ← run_variable(ast)
3:     pos ← pos_variable(ast)
4:     bind(table, var)
5:     bind(table, pos)
6:     expr ← child(0, ast)
7:     e ← compile_expression(expr)
8:     out ← child(1, ast)
9:     o ← compile_operator(out, end)
10:    bind_v ← unbind(table, var)
11:    bind_p ← unbind(table, pos)
12:    op ← create_forbind_operator(e, o, bind_v, bind_p)
13:    return op
14: end function
```

B. Suspend in Chained Sinks

Chained sinks (see Section 7.3.3) are the only components in the push-based operator model, which take influence on the scheduling of tasks, respectively workers. The following proof shows that deadlocks will not occur, even if worker processes are temporarily suspended in the *end()* routine of chained sinks.

Proof. Let t_1 and t_2 be two binding tasks created by a process p for two consecutive partitions of a binding sequence. Let the respective partitions of t_1 and t_2 be small enough so that they can output without further partitioning to chained sibling sinks s_1 and s_2, respectively. Assume the chain token is owned by s_1.

In the fork/join model, t_1 will be executed directly by p. In the producer/consumer model, t_1 is guaranteed to be executed by either p or by a free worker f. In both cases, t_1 is completed without suspending the executing process and the token will be passed on from s_1 to s_2.

If t_2 was not stolen or adopted by a second worker w before p joins t_2 after completion of t_1, p will execute t_2 itself without being suspended. Thereafter, the token will be owned by s_2's right sibling in the fork tree.

If t_2, however, was assigned to a parallel worker w, three constellations may occur:

a) If s_2 receives the token before w calls *end()* on s_2, any pending and all further incoming tuples will be processed by w itself and t_2 is completed without suspending w. The token will be propagated to s_2's right sibling.

b) If w calls *end()* on s_2 before the token is received and sufficient buffer memory is available, t_2 is completed by w, but any pending tuples are left behind at s_2. They will be taken care of when the process that executes t_1 passes on the token over from s_1 to s_2. Thereafter, this process will propagate the token to s_2's right sibling.

c) If w calls *end()* on s_2 before the token is received and available buffer memory is insufficient, w is suspended. When s_2 receives the token from the process that executes t_1, w is restarted and continues the execution of t_2 by processing all pending and all further incoming tuples. At the end, w calls *end()* on s_2 to finalize the sink, and the token gets propagated to s_2's right sibling.

As shown, the token from s_1 will be always propagated over the sibling sink s_2 to the next sink in the chain. As we know that always the left-most sink starts with the token and is handled by a worker, we can deduce by induction that at least one worker will always be active and propagate the token. This property holds for the chained sinks at the leaf level of any fork tree and because the left-to-right execution precedence is guaranteed for all corresponding tasks, cyclic wait conditions cannot occur. □

C. Benchmark Queries

C.1. XMark

Q1

```
let $auction := doc("auction.xml") return
for $b in $auction/site/people/person[@id = "person0"]
return $b/name/text()
```

Q2

```
let $auction := doc("auction.xml") return
for $b in $auction/site/open_auctions/open_auction
return <increase>{$b/bidder[1]/increase/text()}</increase>
```

Q3

```
let $auction := doc("auction.xml")
return for $b in $auction/site/open_auctions/open_auction
       where zero-or-one($b/bidder[1]/increase/text()) * 2
             <= $b/bidder[last()]/increase/text()
return
  <increase
  first="{$b/bidder[1]/increase/text()}"
  last="{$b/bidder[last()]/increase/text()}"/>
```

Q4

```
let $auction := doc("auction.xml") return
for $b in $auction/site/open_auctions/open_auction
where
  some $pr1 in $b/bidder/personref[@person = "person20"],
       $pr2 in $b/bidder/personref[@person = "person51"]
  satisfies $pr1 << $pr2
return <history>{$b/reserve/text()}</history>
```

Q5

```
let $auction := doc("auction.xml") return
count(
  for $i in $auction/site/closed_auctions/closed_auction
  where $i/price/text() >= 40
  return $i/price
)
```

Q6

```
let $auction := doc("auction.xml") return
for $b in $auction//site/regions return count($b//item)
```

Q7

```
let $auction := doc("auction.xml") return
for $p in $auction/site
return
  count($p//description) +
  count($p//annotation) +
  count($p//emailaddress)
```

Q8

```
let $auction := doc("auction.xml") return
for $p in $auction/site/people/person
let $a :=
  for $t in $auction/site/closed_auctions/closed_auction
  where $t/buyer/@person = $p/@id
  return $t
return <item person="{$p/name/text()}">{count($a)}</item>
```

Q9

```
let $auction := doc("auction.xml") return
let $ca := $auction/site/closed_auctions/closed_auction return
let $ei := $auction/site/regions/europe/item
for $p in $auction/site/people/person
let $a :=
  for $t in $ca
  where $p/@id = $t/buyer/@person
  return
    let $n := for $t2 in $ei
              where $t/itemref/@item = $t2/@id
              return $t2
    return <item>{$n/name/text()}</item>
return <person name="{$p/name/text()}">{$a}</person>
```

Q10

```
let $auction := doc("auction.xml") return
for $i in distinct-values($auction/site/people/person/
                          profile/interest/@category)
let $p :=
  for $t in $auction/site/people/person
  where $t/profile/interest/@category = $i
  return
    <personne>
      <statistiques>
        <sexe>{$t/profile/gender/text()}</sexe>
        <age>{$t/profile/age/text()}</age>
        <education>{$t/profile/education/text()}</education>
        <revenu>{fn:data($t/profile/@income)}</revenu>
      </statistiques>
      <coordonnees>
        <nom>{$t/name/text()}</nom>
        <rue>{$t/address/street/text()}</rue>
        <ville>{$t/address/city/text()}</ville>
        <pays>{$t/address/country/text()}</pays>
        <reseau>
          <courrier>{$t/emailaddress/text()}</courrier>
          <pagePerso>{$t/homepage/text()}</pagePerso>
        </reseau>
      </coordonnees>
      <cartePaiement>{$t/creditcard/text()}</cartePaiement>
    </personne>
return <categorie>{<id>{$i}</id>, $p}</categorie>
```

Q11

```
let $auction := doc("auction.xml") return
for $p in $auction/site/people/person
let $l :=
  for $i in $auction/site/open_auctions/open_auction/initial
  where $p/profile/@income > 5000 * exactly-one($i/text())
  return $i
return <items name="{$p/name/text()}">{count($l)}</items>
```

Q12

```
let $auction := doc("auction.xml") return
for $p in $auction/site/people/person
let $l :=
  for $i in $auction/site/open_auctions/open_auction/initial
  where $p/profile/@income > 5000 * exactly-one($i/text())
  return $i
where $p/profile/@income > 50000
return <items person="{$p/profile/@income}">{count($l)}</items>
```

Q13

```
let $auction := doc("auction.xml") return
for $i in $auction/site/regions/australia/item
return <item name="{$i/name/text()}">{$i/description}</item>
```

Q14

```
let $auction := doc("auction.xml") return
for $i in $auction/site//item
where contains(string(exactly-one($i/description)), "gold")
return $i/name/text()
```

Q15

```
let $auction := doc("auction.xml") return
for $a in
  $auction/site/closed_auctions/closed_auction/annotation/
   description/parlist/listitem/parlist/listitem/text/
   emph/keyword/text()
return <text>{$a}</text>
```

Q16

```
let $auction := doc("auction.xml") return
for $a in $auction/site/closed_auctions/closed_auction
where
  not(
    empty(
      $a/annotation/description/parlist/listitem/
       parlist/listitem/text/emph/keyword/text()
    )
  )
return <person id="{$a/seller/@person}"/>
```

Q17

```
let $auction := doc("auction.xml") return
for $p in $auction/site/people/person
where empty($p/homepage/text())
return <person name="{$p/name/text()}"/>
```

Q18

```
declare namespace local = "http://www.foobar.org";
declare function local:convert($v as xs:decimal?) as xs:decimal?
{
  2.20371 * $v (: convert Dfl to Euro :)
};
let $auction := doc('auction.xml') return
for $i in $auction/site/open_auctions/open_auction
return local:convert(zero-or-one($i/reserve))
```

Q19

```
let $auction := doc("auction.xml") return
for $b in $auction/site/regions//item
let $k := $b/name/text()
order by zero-or-one($b/location) ascending
return <item name="{$k}">{$b/location/text()}</item>
```

Q20

```
let $auction := doc("auction.xml") return
<result>
  <preferred>
    {count($auction/site/people/
             person/profile[@income >= 100000])}
  </preferred>
  <standard>
    {
      count(
        $auction/site/people/person/
         profile[@income < 100000 and @income >= 30000]
      )
    }
  </standard>
  <challenge>
    {count($auction/site/people/
             person/profile[@income < 30000])}
  </challenge>
  <na>
    {
      count(
        for $p in $auction/site/people/person
        where empty($p/profile/@income)
        return $p
      )
    }
  </na>
</result>
```

C.2. TPCH

Q2 (XQuery)

```
declare ordering unordered;
declare variable $schema-file external;
let $schema := rel:parse-schema($schema-file, (), ())
for $p in $schema=>part,
    $s in $schema=>supplier,
    $ps in $schema=>partsupp,
    $n in $schema=>nation,
    $r in $schema=>region
where $p=>p_partkey eq $ps=>ps_partkey
  and $s=>s_suppkey eq $ps=>ps_suppkey
  and $p=>p_size eq 15
  and fn:ends-with($p=>p_type, 'BRASS')
  and $s=>s_nationkey eq $n=>n_nationkey
  and $n=>n_regionkey eq $r=>r_regionkey
  and $r=>r_name eq 'EUROPE'
let $supplycost :=
   for $ps in $schema=>partsupp,
       $s in $schema=>supplier,
       $n in $schema=>nation,
       $r in $schema=>region
    where $p=>p_partkey eq $ps=>ps_partkey
      and $s=>s_suppkey eq $ps=>ps_suppkey
      and $s=>s_nationkey eq $n=>n_nationkey
      and $n=>n_regionkey eq $r=>r_regionkey
      and $r=>r_name eq 'EUROPE'
    return $ps=>ps_supplycost
where $ps=>ps_supplycost eq min($supplycost)
order by $s=>s_acctbal descending,
         $n=>n_name, $s=>s_name, $p=>p_partkey
return
  {
    s_acctbal : $s=>s_acctbal,
    s_name : $s=>s_name,
    n_name : $n=>n_name,
    p_partkey : $p=>p_partkey,
    p_mfgr : $p=>p_mfgr,
    s_address : $s=>s_address,
    s_phone : $s=>s_phone,
    s_comment : $s=>s_comment
  }
```

Q2 (SQL)

```
SELECT s_acctbal, s_name, n_name, p_partkey,
       p_mfgr, s_address, s_phone, s_comment
FROM part, supplier, partsupp, nation, region
WHERE p_partkey = ps_partkey
  AND s_suppkey = ps_suppkey
  AND p_size = 15
  AND p_type LIKE '%BRASS'
  AND s_nationkey = n_nationkey
  AND n_regionkey = r_regionkey
  AND r_name = 'EUROPE'
  AND ps_supplycost = (
        SELECT min(ps_supplycost)
        FROM partsupp, supplier,
             nation, region
        WHERE p_partkey = ps_partkey
        AND s_suppkey = ps_suppkey
        AND s_nationkey = n_nationkey
        AND n_regionkey = r_regionkey
        AND r_name = 'EUROPE'
      )
ORDER BY s_acctbal DESC,
         n_name, s_name, p_partkey
FETCH FIRST 100 ROWS ONLY
```

Q6 (XQuery)

```
declare ordering unordered;
declare variable $schema-file external;
let $schema := rel:parse-schema($schema-file, (), ())
for $l in $schema=>lineitem
where $l=>l_discount le 0.07
  and $l=>l_discount ge 0.05
  and $l=>l_shipdate ge xs:date("1994-01-01")
  and $l=>l_shipdate lt xs:date("1994-01-01") +
                        xs:yearMonthDuration("P1Y")
  and $l=>l_quantity lt 24
let $revenue := $l=>l_extendedprice * $l=>l_discount
group by *
return
  { revenue : sum($revenue) }
```

Q6 (SQL)

```
SELECT sum(l_extendedprice * l_discount) AS revenue
FROM lineitem
WHERE l_shipdate >= date ('1994-01-01')
  AND l_shipdate < date ('1994-01-01') + 1 year
  AND l_discount between .06 - 0.01 and .06 + 0.01
  AND l_quantity < 24
```

C.3. TPoX

Filter query (sequential)

```
declare default
    element namespace "http://tpox-benchmark.com/custacc";
for $xml in
    ( bit:partition min=50 max=50 queue=6 )
    { tpox:read-batch('batch-1.xml') }
let $customer := bit:parse($xml)/Customer
let $balance := (
     for $account in $customer/Accounts/Account
     let $obalance := $account/Balance/OnlineActualBal
     where $obalance > 900000
       and $account/Currency = "EUR"
       and ($customer/Nationality = "Greece"
            or $customer/Nationality = "Germany")
       return $obalance
     )
where not(empty($balance))
return
<premium_customer id="{$customer/@id}">
    <name>{$customer/ShortNames/ShortName/text()}</name>
    <balance>{max($balance)}</balance>
    <nationality>{$customer/Nationality/text()}</nationality>
</premium_customer>
```

Note:
The divide-and-conquer versions of the above and the following queries differ solely in the input function used. The divide-and-conquer versions read batches with the function `tpox:read-batch-parallel()`.

Filter query (sequential, expensive binding)

```
declare default
    element namespace "http://tpox-benchmark.com/custacc";
for $doc in
    ( bit:partition min=50 max=50 queue=6 )
    { tpox:parse-batch('batch-1.xml') }
let $customer := $doc/Customer
let $balance := (
     for $account in $customer/Accounts/Account
     let $obalance := $account/Balance/OnlineActualBal
     where $obalance > 900000
       and $account/Currency = "EUR"
       and ($customer/Nationality = "Greece"
            or $customer/Nationality = "Germany")
       return $obalance
     )
where not(empty($balance))
return
<premium_customer id="{$customer/@id}">
    <name>{$customer/ShortNames/ShortName/text()}</name>
    <balance>{max($balance)}</balance>
    <nationality>{$customer/Nationality/text()}</nationality>
</premium_customer>
```

Filter query (sequential, multiple batches)

```
declare default
    element namespace "http://tpox-benchmark.com/custacc";
declare variable $count external;
for $batch in (1 to $count)
for $xml in
    ( bit:partition min=50 max=50 queue=6 )
    { tpox:read-batch(concat('batch-', $batch, '.xml')) }
let $customer := bit:parse($xml)/Customer
let $balance := (
     for $account in $customer/Accounts/Account
     let $obalance := $account/Balance/OnlineActualBal
     where $obalance > 900000
       and $account/Currency = "EUR"
       and ($customer/Nationality = "Greece"
            or $customer/Nationality = "Germany")
       return $obalance
     )
where not(empty($balance))
return
<premium_customer id="{$customer/@id}">
    <name>{$customer/ShortNames/ShortName/text()}</name>
    <balance>{max($balance)}</balance>
    <nationality>{$customer/Nationality/text()}</nationality>
</premium_customer>
```

Group query (sequential)

```
declare default
    element namespace "http://tpox-benchmark.com/custacc";
for $xml in
    ( bit:partition min=50 max=50 queue=6 )
    { tpox:read-batch('custacc/batch-1.xml') }
let $customer := bit:parse($xml)/Customer
where $customer/Accounts/Account/Balance/
       OnlineActualBal > 500000
let $nationality := $customer/Nationality
group by $nationality
return
<nation>
  <name>{$nationality}</name>
  <count>{count($customer)}</count>
</nation>
```

Group query (sequential, multiple batches)

```
declare default
    element namespace "http://tpox-benchmark.com/custacc";
declare variable $count external;
for $batch in (1 to $count)
for $xml in
    ( bit:partition min=50 max=50 queue=6 )
    { tpox:read-batch(concat('custacc/batch-', $batch, '.xml') }
let $customer := bit:parse($xml)/Customer
where $customer/Accounts/Account/Balance/
       OnlineActualBal > 500000
let $nationality := $customer/Nationality
group by $nationality
return
<nation>
  <name>{$nationality}</name>
  <count>{count($customer)}</count>
</nation>
```

Join query (sequential)

```
declare default
    element namespace "http://www.fixprotocol.org/FIXML-4-4";
declare namespace c="http://tpox-benchmark.com/custacc";
for $ordxml in
    ( bit:partition min=50 max=50 queue=6 )
    { tpox:read-batch('order/batch-1.xml') }
let $ord := bit:parse($ordxml)/FIXML/Order
for $custxml in
     ( bit:partition min=50 max=50 )
     { tpox:read-batch('custacc/batch-1.xml') }
let $cust := bit:parse($custxml)/c:Customer
where $ord/OrdQty/@Cash>3000
  and $cust/c:CountryOfResidence = "Germany"
  and $cust/c:Accounts/c:Account/@id = $ord/@Acct/fn:string(.)
return $cust
```

Bibliography

[ABD+89] Malcolm Atkinson, François Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley Zdonik. *The Object-Oriented Database System Manifesto*. 1989.

[Abi97] Serge Abiteboul. *Querying Semi-structured Data*. In *Proc. ICDE*, LNCS, vol. 1186, Springer, 1997, pp. 1–18.

[AG08] Renzo Angles and Claudio Gutierrez. *Survey of Graph Database Models*. ACM Computing Surveys, 40(1):1–39, February 2008.

[Amd67] Gene M. Amdahl. *Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities*. In *Proc. AFIPS (Spring)*, vol. 30, 1967, pp. 483–485.

[AQM+97] Serge Abiteboul, Dallan Quass, Jason Mchugh, Jennifer Widom, and Janet Wiener. *The Lorel Query Language for Semistructured Data*. International Journal on Digital Libraries, 1:68–88, 1997.

[ASS85] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, MA, 1985.

[BAC+90] Haran. Boral, William. Alexander, Larry. Clay, George. Copeland, Scott. Danforth, Michael. Franklin, Brian. Hart, Marc. Smith, and Patrick. Valduriez. *Prototyping Bubba, A Highly Parallel Database System*. IEEE Trans. on Knowl. and Data Eng., 2(1):4–24, March 1990.

[BAS] BaseX XML Database. `http://basex.org`.

[Bat88] Don S. Batory. *Concepts for a Database System Compiler.* In *Proc. PODS*, 1988, pp. 184–192.

[BBB+09] Roger Bamford, Vinayak R. Borkar, Matthias Brantner, Peter M. Fischer, Daniela Florescu, David A. Graf, Donald Kossmann, Tim Kraska, Dan Muresan, Sorin Nasoi, and Markos Zacharioudaki. *XQuery Reloaded.* PVLDB, 2(2):1342–1353, 2009.

[BBG+88] D. S. Batory, J.R. Barnett, J.F. Garza, K.P. Smith, K. Tsukuda, B.C. Twichell, and T.E. Wise. *GENESIS: An Extensible Database Management System.* IEEE Transactions on Software Engineering, 14(11):1711 –1730, 1988.

[BEG+11] Kevin S. Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Y. Eltabakh, Carl-Christian Kanne, Fatma Özcan, and Eugene J. Shekita. *Jaql: A Scripting Language for Large Scale Semistructured Data Analysis.* PVLDB, 1272–1283, 2011.

[BFG+06] Peter Boncz, Jan Flokstra, Torsten Grust, Maurice van Keulen, Stefan Manegold, Sjoerd Mullender, Jan Rittinger, and Jens Teubner. *MonetDB/XQuery - Consistent & Efficient Updates on the Pre/Post Plane.* In *Proc. EDBT*, 2006, pp. 1190–1193.

[BFS00] Peter Buneman, Mary Fernandez, and Dan Suciu. *UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion.* VLDB Journal, 9(1):76–110, March 2000.

[BGvK+05] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. *Pathfinder: XQuery—the relational way.* In *Proc. VLDB*, 2005, pp. 1322–1325.

[BGvK+06] Peter A. Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. *MonetDB/XQuery: A fast XQuery Processor Powered by a Relational Engine.* In *Proc. SIGMOD*, 2006, pp. 479–490.

[BHKM05] Matthias Brantner, Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. *Full-fledged Algebraic XPath Processing in Natix.* In *Proc. ICDE*, 2005, pp. 705–716.

[BK99] Peter A. Boncz and Martin L. Kersten. *MIL Primitives for Querying a Fragmented World.* VLDB Journal, 8(2):101–119, October 1999.

[BKS02] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. *Holistic Twig Joins: Optimal XML Pattern Matching.* In *Proc. SIGMOD*, 2002, pp. 310–321.

[BL94] Robert D. Blumofe and Charles E. Leiserson. *Scheduling Multithreaded Computations by Work Stealing.* In *Proc. FOCS*, 1994, pp. 356–368.

[BNTW95] Peter Buneman, Shamim Naqvi, Val Tannen, and Limsoon Wong. *Principles of Programming with Complex Objects and Collection Types.* Theoretical Computer Science, 149:3–48, 1995.

[BT99] Catriel Beeri and Yariv Tzaban. *SAL: An Algebra for Semistructured Data and XML.* In *Informal Proc. SIGMOD Workshop on The Web and Databases*, 1999, pp. 37–42.

[Bun97] Peter Buneman. *Semistructured Data.* In *Proc. PODS*, 1997, pp. 117–121.

[CD88] Michael J. Carey and David J. DeWitt. *A Data Model and Query Language for EXODUS.* In *Proc. SIGMOD*, 1988, pp. 413–423.

[CDG+90] Michael J. Carey, David J. Dewitt, Goetz Graefe, David M. Haight, Joel E. Richardson, Daniel T. Schuh, Eugene J. Shekita, and Scott L. V. *The EXODUS Extensible DBMS Project: An Overview.* In *Readings in Object-Oriented Database Systems*, Morgan Kaufmann, 1990, pp. 474–499.

[CFG02] Giovanni Conforti, Orlando Ferrara, and Giorgio Ghelli. *TQL Algebra and its Implementation.* In *Proc. IFIP*, 2002, pp. 422–434.

[CG04] Luca Cardelli and Giorgio Ghelli. *TQL: A Query Language for Semistructured Data Based on the Ambient Logic.* Mathematical Structures in Computer Science, 14(3):285–327, 2004.

[CH90] Michael Carey and Laura Haas. *Extensible Database Management Systems.* SIGMOD Record, 19(4):54–60, December 1990.

[CHS+95] Michael J. Carey, Laura M. Haas, Peter M. Schwarz, Manish Arya, William F. Cody, Ronald Fagin, John Thomas, John H, and Edward L. Wimmers. *Towards Heterogeneous Multimedia Information Systems: The Garlic Approach.* In *Proc. RIDE Workshop DOM*, 1995, pp. 124–131.

[CL05] David Chase and Yossi Lev. *Dynamic Circular Work-stealing Deque.* In *Proc. SPAA*, 2005, pp. 21–28.

[CLWY06] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. *Links: Web Programming Without Tiers.* In *Proc. FMCO*, Springer, 2006.

[CM94] Sophie Cluet and Guido Moerkotte. *Nested Queries in Object Bases.* In *Proc. DBPL*, 1994, pp. 226–242.

[Cod70] Edgar F. Codd. *A Relational Model of Data for Large Shared Data Banks.* Communications of the ACM, 13(6):377–387, 1970.

[Cod80] Edgar F. Codd. *Data Models in Database Management.* In *Proc. SIGMOD Workshop on Data Abstraction, Databases and Conceptual Modelling*, vol. 11, 1980, pp. 112–114.

[Cod90] Edgar F. Codd. *The Relational Model for Database Management: Version 2*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[Coo97] Richard Cooper. *Object Databases: An ODMG Approach*, Database Technology Series, International Thomson Computer Press, 1997.

[Cro06] Douglas Crockford. *The application/json Media Type for JavaScript Object Notation (JSON).* July 2006, RFC 4627 (Informational).

[DG92] David J. Dewitt and Jim Gray. *Parallel Database Systems: The Future of High-performance Database Systems.* Communications of the ACM, 35:85–98, 1992.

[DG08] Jeffrey Dean and Sanjay Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters.* Communications of the ACM, 51(1):107–113, January 2008.

[Dit86] Klaus R. Dittrich. *Object-oriented Database Systems (extended abstract): The Notions and the Issues.* In *Proc. OODBS*, 1986, pp. 2–4.

[DV92] Scott Danforth and Patrick Valduriez. *A FAD for Data Intensive Applications.* IEEE Trans. on Knowl. and Data Eng., 4(1):34–51, February 1992.

[Feg98] Leonidas Fegaras. *Query Unnesting in Object-oriented Databases.* In *Proc. SIGMOD*, 1998, pp. 49–60.

[FGK06] Andrey Fomichev, Maxim Grinev, and Sergey Kuznetsov. *Sedna: A Native XML DBMS.* In *Proc. SOFSEM*, 2006, pp. 272–281.

[FHM+05] Mary Fernández, Jan Hidders, Philippe Michiels, Jérôme Siméon, and Roel Vercammen. *Optimizing Sorting and Duplicate Elimination in XQuery Path Expressions.* In *Proc. DEXA*, 2005, pp. 554–563.

[FLM98] Daniela Florescu, Alon Levy, and Alberto Mendelzon. *Database Techniques for the World-Wide Web: A Survey.* SIGMOD Record, 27:59–74, 1998.

[Flo99] Daniela Florescu. *Query Optimization in the Presence of Limited Access Patterns.* In *Proc. SIGMOD*, 1999, pp. 311–322.

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. *The Implementation of the Cilk-5 Multithreaded Language*. In *Proc. SIGPLAN*, vol. 33, 1998, pp. 212–223.

[FSW00] Mary Fernandez, Jerome Simeon, and Philip Wadler. *An Algebra for XML Query*. Proc. FST TCS 2000, LNCS, vol. 1974, 2000, pp. 11–45.

[GGSW11] George Giorgidze, Torsten Grust, Nils Schweinsberg, and Jeroen Weijers. *Bringing Back Monad Comprehensions*. In *Proc. ACM Symposium on Haskell*, 2011, pp. 13–22.

[GMR09] Torsten Grust, Manuel Mayr, and Jan Rittinger. *XQuery Join Graph Isolation: Celebrating 30+ Years of XQuery Processing Technology*. In *Proc. ICDE*, 2009, pp. 1167–1170.

[GMRS09] Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. *FERRY: Database-supported Program Execution*. In *Proc. SIGMOD*, 2009, pp. 1063–1066.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993.

[Gra94] Goetz Graefe. *Volcano: An Extensible and Parallel Query Evaluation System*. IEEE Trans. on Knowl. and Data Eng., 6(1):120–135, February 1994.

[GRT08] Torsten Grust, Jan Rittinger, and Jens Teubner. *Pathfinder: XQuery Off the Relational Shelf*. IEEE Data Eng. Bull., 31(4):7–14, 2008.

[Gru99] Torsten Grust. *Comprehending Queries*. Tech. report, 1999.

[Grü10] Christian Grün. *Storing and Querying Large XML Instances*. Ph.D. thesis, University of Konstanz, Konstanz, 2010.

[Gus88] John L. Gustafson. *Reevaluating Amdahl's Law*. Communications of the ACM, 31:532–533, 1988.

[GvKT03] Torsten Grust, Maurice van Keulen, and Jens Teubner. *Staircase Join: Teach a Relational DBMS to Watch Its (Axis) Steps.* In *Proc. VLDB*, 2003, pp. 524–535.

[GW97] Roy Goldman and Jennifer Widom. *DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases.* In *Proc. VLDB*, 1997, pp. 436–445.

[Gü89] Ralf Hartmut Güting. *Gral: An Extensible Relational Database System for Geometric Applications.* In *Proc. VLDB*, 1989, pp. 33–44.

[HFLP89] Laura M. Haas, Johann-Christoph Freytag, Guy M. Lohman, and Hamid Pirahesh. *Extensible Query Processing in Starburst.* SIGMOD Record, 18(2):377–388, 1989.

[HHMW05] Michael Peter Haustein, Theo Härder, Christian Mathis, and Markus Wagner. *DeweyIDs - The Key to Fine-Grained Management of XML Documents.* In *Proc. SBBD*, 2005, pp. 85–99.

[HHMW07] Theo Härder, Michael Peter Haustein, Christian Mathis, and Markus Wagner. *Node Labeling Schemes for Dynamic XML Documents Reconsidered.* Data & Knowledge Engineering, 60(1):126–149, 2007.

[HS02] Danny Hendler and Nir Shavit. *Non-blocking Steal-half Work Queues.* In *Proc. PODC*, 2002, pp. 280–289.

[Hut99] Graham Hutton. *A Tutorial on the Universality and Expressiveness of Fold.* Journal of Functional Programming, 9(4):355–372, July 1999.

[IBY+07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. *Dryad: Distributed Data-parallel Programs from Sequential Building Blocks.* Proc. SIGOPS, 41(3):59–72, 2007.

[JLST02] H. Jagadish, Laks Lakshmanan, Divesh Srivastava, and Keith Thompson. *TAX: A Tree Algebra for XML.* Proc. DBPL, LNCS, vol. 2397, Springer, 2002, pp. 149–164.

[JS82] Gerhard Jaeschke and Hans-Jörg Schek. *Remarks on the Algebra of Non-first Normal Form Relations.* In *Proc. PODS*, 1982, pp. 124–138.

[JW07] Simon Peyton Jones and Philip Wadler. *Comprehensive Comprehensions.* In *Proc. SIGPLAN Workshop on Haskell*, 2007, pp. 61–72.

[KA02] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*, Morgan Kaufmann, San Francisco, CA, USA, 2002.

[Kay08] Michael Kay. *Ten Reasons Why Saxon XQuery is Fast.* IEEE Data Eng. Bull., 31(4):65–74, 2008.

[Klu82] Anthony Klug. *Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions.* ACM Journal, 29(3):699–717, July 1982.

[KM94] Alfons Kemper and Guido Moerkotte. *Object-oriented Database Management: Applications in Engineering and Computer Science*, Prentice-Hall, Inc., 1994.

[LA04] Chris Lattner and Vikram Adve. *LLVM: A Compilation Framework for Lifelong Program Analysis Transformation.* In *Proc. CGO*, 2004, pp. 75–86.

[Lea00] Doug Lea. *A Java Fork/Join Framework.* In *Proceedings ACM Java Grande*, 2000, pp. 36–43.

[Lie87] Henry Lieberman. *Object-oriented Concurrent Programming.* MIT Press, Cambridge, MA, USA, 1987, pp. 9–36.

[LKA05] Zhen Hua Liu, Muralidhar Krishnaprasad, and Vikas Arora. *Native XQuery Processing in Oracle XMLDB.* In *Proc. SIGMOD*, 2005, pp. 828–833.

[LR89] Christophe Lécluse and Philippe Richard. *The O2 Database Programming Language.* In *Proc. VLDB*, 1989, pp. 423–432.

[LS88] Barbara Liskov and Luiba Shrira. *Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems*. SIGPLAN Not., 23(7):260–267, jun 1988.

[MAG+97] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallas Quass, and Jennifer Widom. *Lore: A Database Management System for Semistructured Data*. SIGMOD Record, 26(3):54–66, September 1997.

[Mat09] Christian Mathis. *Storing, Indexing, and Querying XML Documents in Native Database Management Systems*. Ph.D. thesis, University of Kaiserslautern, July 2009.

[MBB06] Erik Meijer, Brian Beckman, and Gavin Bierman. *LINQ: Reconciling Object, Relations and XML in the .NET Framework*. In *Proc. SIGMOD*, 2006, pp. 706–706.

[Mei07] Erik Meijer. *Confessions of a Used Programming Language Salesman*. SIGPLAN Not., 42(10):677–694, October 2007.

[Mei09] Wolfgang Meier. *eXist: An Open Source Native XML Database*. In *Proc. Web, Web-Services, and Database Systems*, 2009, pp. 169–183.

[MHM04] Norman May, Sven Helmer, and Guido Moerkotte. *Nested Queries and Quantifiers in an Ordered Context*. In *Proc. ICDE*, 2004, pp. 239–250.

[MHS09] Christian Mathis, Theo Härder, and Karsten Schmidt. *Storing and Indexing XML Documents Upside Down*. Computer Science - R&D, 24(1-2):51–68, 2009.

[MHSB12] Christian Mathis, Theo Härder, Karsten Schmidt, and Sebastian Bächle. *XML Indexing and Storage: Fulfilling the Wish List*. Computer Science - R&D, 27(2), 6 2012.

[MXQ] MXQuery XQuery Processor. `http://http://mxquery.org`.

[NKS07] Matthias Nicola, Irina Kogan, and Berni Schiefer. *An XML Transaction Processing Benchmark*. In *Proc. SIGMOD*, 2007, pp. 937–948.

[NvdL05] Matthias Nicola and Bert van der Linden. *Native XML Support in DB2 Universal Database*. In *Proc. VLDB*, 2005, pp. 1164–1174.

[OBBt89] Atsushi Ohori, Peter Buneman, and Val Breazu-tannen. *Database Programming in Machiavelli, A Polymorphic Language with Static Type Inference*. 1989, pp. 46–57.

[OO83] Zehra Meral Özsoyoglu and Gultekin Özsoyoglu. *An Extension of Relational Algebra for Summary Tables*. In *Proc. SSDBM*, 1983, pp. 202–211.

[OOP+04] Patrick E. O'Neil, Elizabeth J. O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. *ORDPATHs: Insert-Friendly XML Node Labels*. In *Proc. SIGMOD*, 2004, pp. 903–908.

[ORS+08] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. *Pig Latin: A Not-so-foreign Language for Data Processing*. In *Proc. SIGMOD*, 2008, pp. 1099–1110.

[ÖSW08] Fatma Özcan, Normen Seemann, and Ling Wang. *XQuery Rewrite Optimization in IBM DB2 pureXML*. IEEE Data Eng. Bull., 31(4):25–32, 2008.

[PGmW95] Yannis Papakonstantinou, Hector Garcia-molina, and Jennifer Widom. *Object Exchange Across Heterogeneous Information Sources*. In *Proc. ICDE*, 1995, pp. 251–260.

[PK87] Constantine D. Polychronopoulos and David J. Kuck. *Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers*. IEEE Transactions on Computers, C-36(12):1425 –1439, dec. 1987.

[PL92] Simon L. Peyton Jones and David R. Lester. *Implementing Functional Languages: A Tutorial*, Prentice Hall, 1992.

[Pol88] Constantine D. Polychronopoulos. *Parallel Programming and Compilers*, Kluwer Academic Publishers, Norwell, MA, USA, 1988.

[QEX] Qexo XQuery Processor. `http://www.gnu.org/software/qexo`.

[QIZ] Qizx XQuery Processor. `http://www.xmlmind.com/qizx`.

[RAH+96] Mary Tork Roth, Manish Arya, Laura M. Haas, Michael Carey, F. William Cody, Ronald Fagin, Peter M. Schwarz, Joachim Thomas, and Edward L. Wimmers. *The Garlic Project*. In *Proc. SIGMOD*, 1996, p. 557.

[RCF00] Jonathan Robie, Don Chamberlin, and Daniela Florescu. *Quilt: an XML Query Language*. 2000, `http://www.almaden.ibm.com/cs/people/chamberlin/quilt_euro.html`.

[Rei07] James Reinders. *Intel Threading Building Blocks*, O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007.

[Rey72] John C. Reynolds. *Definitional Interpreters for Higher-order Programming Languages*. In *Proc. ACM Conference - Volume 2*, 1972, pp. 717–740.

[RKS88] Mark A. Roth, Herry F. Korth, and Abraham Silberschatz. *Extended Algebra and Calculus for Nested Relational Databases*. ACM Transactions on Database Systems, 13(4):389–417, October 1988.

[RS87] Lawrence A. Rowe and Michael Stonebraker. *The POSTGRES Data Model*. In *Proc. VLDB*, 1987, pp. 83–96.

[RSF06] Christopher Re, Jérôme Siméon, and Mary F. Fernández. *A Complete and Efficient Algebraic Compiler for XQuery*. In *Proc. ICDE*, 2006, p. 14.

[Rys05] Michael Rys. *XML and Relational Database Management Systems: Inside Microsoft SQL Server 2005*. In *Proc. SIGMOD*, 2005, pp. 958–962.

[SA02] Carlo Sartiani and Antonio Albano. *Yet Another Query Algebra for XML Data*. In *Proc. IDEAS*, 2002, pp. 106–115.

[SAC+79] Patricia G. Selinger, Morton M. Astrahan, Don D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. *Access Path Selection in a Relational Database Management System.* In *Proc. SIGMOD* (New York, NY, USA), SIGMOD '79, ACM, 1979, pp. 23–34.

[SJ75] Gerald Jay Sussman and Guy L Steele Jr. *Scheme: An Interpreter for Extended Lambda Calculus.* In *MEMO 349, MIT AI LAB*, 1975.

[SKS06] Abraham Silberschatz, Henry Korth, and S. Sudarshan. *Database Systems Concepts*, 5 ed., McGraw-Hill, Inc., New York, NY, USA, 2006.

[SLS06] William N. Scherer, III, Doug Lea, and Michael L. Scott. *Scalable Synchronous Queues.* In *Proc. PPoPP*, 2006, pp. 147–156.

[SM95] Michael Stonebraker and Dorothy Moore. *Object Relational DBMSs: The Next Great Wave*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995.

[SPSW90] Hans-Jörg Schek, H.-Bernhard Paul, Marc. H. Scholl, and Gerhard Weikum. *The DASDBS Project: Objectives, Experiences, and Future Prospects.* IEEE Trans. on Knowl. and Data Eng., 2(1):25–43, March 1990.

[SQL99] ANSI/ISO/IEC 9075-2:1999. *Information Technology - Database languages - SQL - Part 2: Foundation (SQL/-Foundation).* 1999.

[SQL03] ANSI/ISO/IEC 9075-2:2003. *Information Technology - Database languages - SQL - Part 2: Foundation (SQL/-Foundation).* 2003.

[SR92] Manojit Sarkar and Steve Reiss. *A Data Model and A Query Language for Object-Oriented Databases.* Tech. report, Providence, RI, USA, 1992.

[SRH90] Michael Stonebraker, Lawrence A. Rowe, and Michael Hirohama. *The Implementation of POSTGRES.* IEEE Trans. on Knowl. and Data Eng., 2(1):125–142, March 1990.

[STL11] Robert J. Stewart, Phil W. Trinder, and Hans-Wolfgang Loidl. *Comparing High-level Mapreduce Query Languages.* In *Proc. APPT*, 2011, pp. 58–72.

[SWK+02] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. *XMark: A Benchmark for XML Data Management.* In *Proc. VLDB*, 2002, pp. 974–985.

[TCBV10] Alexandros Tzannes, George C. Caragea, Rajeev Barua, and Uzi Vishkin. *Lazy Binary-splitting: A Run-time Adaptive Work-stealing Scheduler.* In *Proc. SIGPLAN PPoPP*, 2010, pp. 179–190.

[TPC12] Transaction Processing Performance Council TPC. *TPC Benchmark H (Decision Support), Standard Specification, Revision 2.14.4.* 2012.

[TW89] Phil Trinder and Philip Wadler. *Improving List Comprehension Database Queries.* In *Proc. TENCON*, 1989, pp. 186–192.

[Tü03] Can Türker. *SQL:1999 & SQL:2003 - Objektrelationales SQL, SQLJ & SQL/XML.* 2003.

[VB96] Emilia E. Villarreal and Don S. Batory. *Rosetta: A Generator of Data Language Compilers.* In *Proc. Symposium on Software Reuse*, 1996.

[VXQ] VXQuery XQuery Processor. `http://incubator.apache.org/vxquery`.

[W3C98a] W3C. *Document Object Model (DOM) Level 1 Specification Version 1.0 – W3C Recommendation 1 October, 1998.* 1998, `http://www.w3.org/TR/REC-DOM-Level-1`.

[W3C98b] W3C. *XML-QL: A Query Language for XML – Submission to the World Wide Web Consortium 19. August 1998.* 1998, `http://www.w3.org/TR/NOTE-xml-ql`.

[W3C98c] W3C. *XML Query Language (XQL).* 1998, `http://www.w3.org/TandS/QL/QL98/pp/xql.html`.

[W3C99] W3C. *XML Path Language (XPath) Version 1.0 – W3C Recommendation 16 November 1999*. 1999, http://www.w3.org/TR/xpath.

[W3C04a] W3C. *XML Information Set (Second Edition) – W3C Recommendation 4 February 2004*. 2004, http://www.w3.org/TR/xml-infoset.

[W3C04b] W3C. *XML Schema Part 0: Primer Second Edition – W3C Recommendation 28 October 2004*. 2004, http://www.w3.org/TR/xmlschema-0.

[W3C06a] W3C. *Extensible Markup Language (XML) 1.1 (Second Edition) – W3C Recommendation 16 August 2006, edited in place 29 September 2006*. 2006, http://www.w3.org/TR/xml11.

[W3C06b] W3C. *XQuery Update Facility – W3C Working Draft 11 July 2006*. 2006, http://www.w3.org/TR/xqupdate.

[W3C07] W3C. *XML Path Language (XPath) 2.0 W3C Recommendation 23 January 2007*. 2007, http://www.w3.org/TR/xpath20.

[W3C09] W3C. *Namespaces in XML 1.0 (Third Edition) – W3C Recommendation 8 December 2009*. 2009, http://www.w3.org/TR/xml-names.

[W3C10a] W3C. *XQuery 1.0: An XML Query Language (Second Edition) – W3C Recommendation 14 December 2010*. 2010, http://www.w3.org/TR/xquery.

[W3C10b] W3C. *XQuery 1.0 and XPath 2.0 Formal Semantics*. 2010, http://www.w3.org/TR/xquery-semantics.

[W3C10c] W3C. *XQuery 3.0: An XML Query Language*. 2010, http://www.w3.org/TR/xquery-30.

[W3C10d] W3C. *XQuery Scripting Extension 1.0 – W3C Working Draft 8 April 2010*. 2010, http://www.w3.org/TR/xquery-sx-10.

[W3C11a] W3C. *XPath and XQuery Functions and Operators 3.0 – W3C Working Draft 13 December 2011.* 2011, `http://www.w3.org/TR/xpath-functions-30`.

[W3C11b] W3C. *XQuery and XPath Data Model 3.0 – W3C Working Draft 13 December 2011.* 2011, `http://www.w3.org/TR/xpath-datamodel-30`.

[W3C11c] W3C. *XSLT and XQuery Serialization 3.0 – W3C Working Draft 13 December 2011.* 2011, `http://www.w3.org/TR/xslt-xquery-serialization-30`.

[Wad90] Philip Wadler. *Comprehending Monads.* In *Proc. LFP*, 1990, pp. 61–78.

[WB87] Michael Wolfe and Utpal Banerjee. *Data Dependence and its Application to Parallel Processing.* International Journal of Parallel Programming, 16:137–178, 1987. 10.1007/BF01379099.

[Weg87] Peter Wegner. *Dimensions of Object-based Language Design.* SIGPLAN Not., 22(12):168–182, December 1987.

[Whi09] Tom White. *Hadoop: The Definitive Guide*, O'Reilly Media, 2009.

[Won98] Limsoon Wong. *Kleisli, a Functional Query System.* Journal of Functional Programming, 10, 1998.

[Woo12] Peter T. Wood. *Query Languages for Graph Databases.* SIGMOD Record, 41(1):50–60, April 2012.

[XQI] XQilla XQuery Processor. `http://xqilla.sourceforge.net`.

Lebenslauf

Persönliche Daten

Name	Sebastian Bächle
Geburtsdatum	11. November 1981
Geburtsort	Zweibrücken
Familienstand	verheiratet

Schulbildung

1988 -1992	Grundschule Kirrberg, Homburg (Saar)
1992 - 2001	Saarpfalz Gymnasium Homburg, Homburg (Saar)
06/2001	Abitur

Zivildienst

08/2001 - 06/2002	Rettungssanitäter, Malteser Hilfsdienst, Homburg (Saar)

Studium

10/2002 - 08/2007	Angewandte Informatik, TU Kaiserslautern
08/2007	Erlangen des akademischen Grades *Dipl.-Inf.*

Berufstätigkeit

09/2007 - 08/2012	Wissenschaftlicher Mitarbeiter, AG Datenbanken und Informationssysteme, Fachbereich Informatik, TU Kaiserslautern
seit 10/2012	Softwareentwickler, SAP AG, Walldorf