Processing XML in Database Systems
Schmidt, A.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 4

Querying XML Data

In this chapter\(^1\), we discuss issues around opening up the contents of XML documents to users by means of query languages. We start out by looking at how XML documents can be bulkloaded efficiently and show that we again can take advantage of the tree structure of documents to gain speed. We then look at how relational query algebras can be extended to support regular path expressions; a solution that makes full use of Monet XML will be presented. Then, we study a new query operator, the \textit{meet}, that allows users to query XML documents even if they are not familiar with the tag structure in the database. Eventually, we discuss an extension to the relational algebra and SQL language which is not only useful for querying XML but also in a broader context.

4.1 Bulkloading XML

The popularity of XML as a exchange and storage format brings about large amounts of documents to be stored, maintained and analysed – a challenge that traditionally has been tackled with Database Management Systems. To open up the content of XML documents to analysis with declarative query languages, efficient bulk loading techniques are a prerequisite. Database technology has traditionally been offering support for these tasks but yet falls short of providing efficient automation techniques for the challenges that large collections of XML data raise. In this section, we discuss bulk-loading techniques for Monet XML as a storage back-end; since many applications rely on relational databases, which are designed towards large data volumes, our results carry over to a larger context. Furthermore, we also present a bulk deletion technique, put it into an application context and look at a case study when to use edit-scripts to navigate through and make changes to a document tree by chasing references and when to use bulk deletion, a technique we can expect to work better when we want to update large data volumes. We then conclude this section with a performance study.

4.1.1 Introduction

Still, there is a lack of large scale native XML query engines that go beyond search engine functionality so that the massive amounts of XML data produced by today’s applications escape attempts of disclosure for analysis and maintenance. While it is

\(^1\)Parts of this chapter were published in [SWK99, WSK99, SKW01, SK02, WSV+02, WGLGS02]
certainly possible to convert XML data to other formats for which solutions exists, from a software engineering point of view it would be preferable to go for 'all XML' solutions. A viable approach to achieving this goal is to adapt relational database technology to store and maintain XML documents as proposed in the previous chapter or, for example, in [DFS99, FK99] and nicely demonstrated in [STH+99, KM00b]. The advantage of this approach is that the XML repository inherits all the power of mature relational technology like indexes, transaction management etc. As a first step towards this goal researchers and industry have been extending query languages like SQL with XML specific constructs [SKS*01] that make it possible to deploy XML-based data management in mission critical scenarios. However these solutions still assume a traditional data model and make it more difficult than necessary to use XML in all layers of a software system.

Traditionally, database technology has been offering support for processing large amounts of data. Whereas there has already been considerable research into query languages and logical data models for XML data – early references are [BDHS96, AQM*97] –, there have only been few proposals to meet the challenge of scaling XML databases up to production levels as achieved by relational engines and, thus, to gain acceptance among practitioners. Naturally, XML warehouses, i.e., collections of XML documents gathered from remote sources, inherit the power of relational warehouses [Rou97] but they also face the same challenges; for example, update and consistency problems of materialised replicated and aggregated views over source data need to solved. The first step of populating a warehouse based, for example, on Monet XML is to find a framework that builds on well-understood relational database technology and enables efficient management of large XML repositories. To get the most of relational database systems, we should do away with pointer-chasing tree traversing operations and replace them with set-oriented operations whenever possible.

Unfortunately, many applications are designed to generate updates in the form of edit scripts; detailed descriptions of these scripts are published in [CGM97] and [CRGMW96]. They have been long known in text databases and are similar in behaviour to DOM traversals, of which they can be viewed an abstraction; edit scripts clearly disadvantage relational technology due to their excessive use of pointer-chasing algorithms which generate more or less random seek operations on the database instance. We investigate the use of these scripts and propose alternative strategies for cases when they perform poorly. In the performance evaluation, we find that for our sample queries the use of edit-scripts is only sensible if they update a rather small fraction of the database only; once a certain threshold is exceeded, the replacement of a complete database segment is preferable. We discuss this threshold and try to quantify the trade-off for our example document database.

4.1.2 Application Scenario

The application scenario which motivates our research consists of a set of XML data sources that monitor multimedia data sources and analyse their content; they feed the protocols of the analyses into a central data warehouse. The XML data sources are feature detectors as developed in the Digital Media Warehouse (DMW) project [WSv*02]; the following subsection will give an overview of Feature Grammars, the declarative specification language used to define the behaviour of the feature detector. The warehouse now provides the following services to gather and maintain the output of the detectors:
4.1 Bulkloading XML

Figure 4.1: Application Scenario

![Diagram](/cdrom/images1/23493.jpeg)

**Figure 4.1:** Application Scenario

Figure 4.2: Example document

```xml
<image key="18934" source="/cdrom/images1/23493.jpeg">
  <date>999010530</date>
  <colours>
    <histogram>0.399 0.277 0.344</histogram>
    <saturation>0.390</saturation>
    <version>0.8</version>
  </colours>
</image>
```

**Figure 4.2:** Example document

(1) insertion of a document, *i.e*., a data source transmits a single protocol of an analysis to the warehouse;

(2) insertion of sets of documents which are the result of bulk analyses executed by the data sources;

(3) deletion of documents and sets of documents happens when a document has become invalid or stale, for example because duplicate analyses or erroneous insertion, which happen frequently, need to be corrected; and finally,

(4) execution of edit scripts that are transmitted from the sources and either systematically correct errors in already inserted documents or update a single document; for example, *a posteriori* normalisation of feature values is required frequently.

While (1) can be reduced to (2) since both require the same database actions and hence is not treated separately here, there is an obvious trade-off between a combination of (2) and (3) and the use of edit-scripts (4): it is possible to update the warehouse with deletions of parts of the stored document and re-insertions of an externally generated documents which contain the updates. More precisely, the question we deal with in this section is: When is it cheaper to delete invalid data and re-insert a new consistent version than to use an edit script to 'patch' the warehouse?

Figure 4.2 shows an XML fragment which is taken from the application scenario described above; for orientation, Figure 4.3 displays the corresponding tree (arrows
again indicate XML attribute relationships, straight lines XML element relationships).
The semantics of the document are as follows: the image taken from the file located at /cdrom/images1/23493.jpeg is assigned a key and contains the date of analysis, in this case given according to UNIX system time in seconds since 1st January 1970 and information on the colour distribution in the image. Enclosed in <histogram> tags is the distribution of red, green and blue values; furthermore, the colour saturation and the version of the detector are given.

4.1.3 Excursion: Feature Grammars

This subsection serves as an aside to provide more context to the application scenario used throughout this section and the framework in which the research was carried out. The example in Figure 4.2 is a simplified version of documents produced and in use in the Feature Detector Engine of the Acoi project [KNW98, SWK99], which is the subject of this overview. Detailed descriptions of various aspects of the system can be found in [WSv02].

Acoi is a framework that enables the declarative specification of multimedia search engines. Features Grammars are the linguistic tool to define the behaviour and interplay of the various system components, which are displayed in Figure 4.4. They are used throughout the three levels of the system: on the conceptual level they export a domain-specific description of the contents of the database. Users can consult it to find out whether the contents of the database match their interests and identify portions which they might want to query. The logical level consists of the definition of both how the system interacts with the user when he issues a query and how the system goes about building the index, for example, by following links on a web page or working its way through a collection of images or MIDI files. The physical level, to which the contents of this thesis apply as well, eventually contains a database instance of the descriptions generated by the specification on the logical level.

How these descriptions are generated is outlined in the sequel. Figure 4.5 shows a Feature Grammar (again simplified to suit our presentation) that describes images; the basic idea is now to use the parse tree that is implicitly or explicitly constructed when a Feature Grammar is evaluated against an image as a description of the document’s content and to store it for later querying. The structure of a Feature Grammar resembles Yacc [Joh78] grammars: a declaration section is followed by a rule section, where the
right-hand sides of rules are allowed to use regular expressions; in this sense they are related to Regular Right-Part Grammars [LaL77]. The grammar reads as follows: the symbol from which parsing is started is called image. Five detectors are used in the grammar: colours delivers colour information like the aforementioned histogram of the Red, Green, Blue values in the image and the saturation. To execute the detector a MATLAB [HL97] routine is called; more types of detectors like decision rules or a back-propagation neural network are possible. In the case of the colours detector only analyses by version 0.8 or more recent versions of the detector are considered for queries; older versions stored inside the database either require that a new analysis is initiated by the Feature Detector Scheduler or the item is not included in the set of possible answer candidates at query time. The remaining declarations read as follows: the data type url is imported from the www package and the bitmap type from the image package. The terminal source is of type url, similarly saturation and skin are of type float and bitmap, respectively. The unit which comprises the components described in this paragraph is called Feature Detector Engine (see Figure 4.4).

The focal point of the concept of detector is that they are allowed to write to the input tape from which they read their input data as well as access during the parsing process those parts of the parse tree that have already been constructed. In the example,
Figure 4.5: A Feature Grammar corresponding to our example

when the source symbol is encountered, raw bitmap data are downloaded from the
given URI and written to the input tape where they are consumed by the colours
detector and the detectors which classify the image into graphics, photos or portraits,
which consist of skin and face.

If the required detectors are at hand, Feature Grammars allow a straightforward
integration of all system components so that it is straightforward to construct a search
engine for a specific domain. The system components in Figure 4.4 that we have not
mentioned so far enable incremental maintenance of the warehouse (Feature Detector
Scheduler) and provide a query interface (Feature Query Engine). The physical level,
denoted meta-data, stores a representation of the parse trees persistently. Note that the
extensibility of XML goes nicely together with the dynamic extensibility of Feature
Grammars.

4.1.4 Populating the XML Warehouse

As pointed out in Chapter 2, there are two standardised ways of accessing XML doc-
uments. On the one hand, the low-level event-based API, called SAX [Meg01], where
an XML document is tokenised into start tags, end tags, character data etc. and user-
supplied functions are called on encountering each type of token. The advantage of
the SAX parsers is they only require minimal resources to work efficiently. On the
other hand, the high-level DOM interface provides a standard interface to parse trees
of complete documents. In terms of resources, the memory consumption of DOM
trees is much higher, linear in the size of the document with usually a rather large
constant for administrative data structures; thus, it may happen that large documents
Figure 4.6: Path sequences in the example document (tokens which only consist of whitespace have been omitted)

4.1 Bulkloading XML

```xml
1  <image key="18934"
2      source="/cdrom/images1/23493.jpeg">
3  <image><date>
4  <image><date>"999010530"
5  <image><date></date>
6  <image><colors>
7  <image><colors><histogram>"0.399 0.277 0.344"
8  <image><colors></histogram>
9  <image><colors><saturation>
10 <image><colors><saturation>"0.390"
11 <image><colors></saturation>
12 <image><colors><version>
13 <image><colors><version>"0.8"
14 <image><colors></version>
15 <image></colors>
16 </image>
```

exceed the size of available main memory. It is therefore desirable to use a parsing method that builds on SAX parsing to enable processing of documents whose size is greater than the available main memory. The bulk load method presented in this section has only slightly higher memory requirements than SAX – \(O(\text{height of document})\) vs. constant memory – but still keeps track of all the contextual information it needs to construct a Monet XML instance. This is achieved by materialising only those parts of the parse tree which are relevant in the current context: using the definitions of the last section, for a node with OID \(o\) only those nodes that lie on the path from the root to \(o\) are kept in memory. Later we present an optimised bulk load method that is even faster however at the cost of slightly higher memory consumption, namely \(O(\text{number of paths in document})\).

Since Monet XML stores complete paths, the bulk load routine has to track those paths. This is done with a stack: whenever we encounter a start tag we push the tag name on the stack and generate insertion code for the tag, its rank, and the association list of attributes. If the stack contains just one element we have just encountered the document root. When we encounter an end tag we simply pop the stack. We need not check for the well-formedness constraint that the end tag encountered carries the same tag that is the top element of the stack since the XML parser itself is supposed to reject such a document before notifying the user of the tag. For presentation purposes we don’t use the callback mechanism that most SAX parsers use as an interface since control flow is then hidden inside a third-party piece of software. Rather, we use the more traditional token-based view, which is known from software tools like Lex [LMB92] and Yacc [Joh78] where a lexical analyser chops the input stream up into tokens which are then consumed by a parser.

A simplified insertion algorithm is displayed in Figure 4.8; note that quite a few details are missing such as treatment of entities, namespaces, attributes, processing instructions etc. to simplify the presentation. The remaining code simply generates OIDs for every tag whenever a tag which is not an end tag is encountered; to remem-
ber parent-child relationships these OIDs are also stored on the stack. This way, we are able to record all path instances in the documents without having to maintain a syntax tree in main memory – an advantage that lets us process very large amounts of documents in relatively little main memory. Run against the example document, the insertion algorithm produces the following sequence of insert statements; insert(R, t) is the function that adds a tuple to a database instance; R is a reference to a relation and t is a tuple of the appropriate type:

1. \( \text{insert(sys, } \langle o_1, \text{image} \rangle \) 
2. \( \text{insert(R(image[key]), } \langle o_1, \text{"18934"} \rangle \) 
3. \( \text{insert(R(image[source]), } \langle o_1, \text{"/cdrom/images1/23493.jpeg"} \rangle \) 
4. \( \text{insert(R(image/date), } \langle o_1, o_2 \rangle \) 
5. \( \text{insert(R(image/date/pndata), } \langle o_2, \text{"999010530"} \rangle \) 
6. \( \ldots \) 

Note that this sequence of insert statements requires us to hash the *complete* path to a relation name for every statement. By exploiting the hierarchical structure of the schema tree we can do much better by organising the path summary as a *schema tree* instead of using a simple stack. This way we can map paths to relations much more efficiently. Each node in the schema tree represents a database relation and contains a tag name and reference to the relation. The schema tree of the example document is displayed in Figure 4.7. Recall that Figure 4.6 shows the path sequences generated by combining the SAX events of the parser and a stack. We now can do away with much of the hashing by storing the part of the hierarchical context, which is invariant for every path, in the schema tree: when we encounter a start tag, we look at the sons of the current context. There are now cases: (1) we find a son that represents the tag,
type stack_element =
   oid : oid;
   tag : string;
end

function relation(stack s) : Relation
  return relation representing concatenation of all tags on stack
end

function bulkload(stack s, int r) : void
  let t = read_token()
  switch type(t)
    case start_token:
      if stack_depth = 1 then
        register_root(r)
        push (s, stack_element_(new_oid, t.tag))
      end
      else
        push (s, stack_element_(new_oid, t.tag))
        insert (relation(s), second(s).oid, first(s).oid)
      end
    end
    case end_token:
      pop (s)
    end
    case character_data:
      push (s, stack_element_(new_oid, cdata))
      insert (relation(s), second(s).oid, first(s).oid)
      insert (relation(s), first(s).oid, text(t))
      pop (s)
    end
  end
end

Figure 4.8: Simplified bulkload algorithm
or, (2) there is no son that represents the tag. In the first case, we simply push the son on the stack, thus making it the current context, and store the OIDs in the relation that is associated with the son. If we don't find a child node that represents the tag, then the path does not yet exist in the database. In this case, we create a node and the respective relation and continue processing with the newly created node as in (1). If we encounter an end tag we 'pop' the stack twice, i.e., pop both the start and corresponding end tag. The performance analysis at the end of this paper quantifies the improvement this simple trick brings about.

We note that we can easily extend the bulkload procedure to record extents of elements as introduced by [ZND'01], i.e., the textual position of a start tag and its corresponding end tag. This is done most easily by storing the end positions in a separate relation; less space consuming would be to insert the start tags when their corresponding end tag is reached and to keep the necessary information on the stack. We can also use the extent mechanism to implement a multi-attribute schema for documents which come along with a DTD by reserving slots for every 1:1 parent-child relationship specified in the DTD and flushing tuples once the end of their extent is reached. This idea combines the algorithms presented in [STH'99] and the notion of path summaries, which are not present in that work.

4.1.5 Maintaining the Database

Once data reside in a database, maintenance of these data becomes an important issue. In our scenario, we distinguish between two different maintenance tasks: First, the update of existing data via edit-scripts for propagating changes of source data to the warehouse, and, second, the deletion and insertion of complete versions of documents which may have become stale or need to be added to the warehouse.

The concept of edit scripts to update hierarchically structured data is both intuitive and easy to implement on modern database systems; it is defined in [CRGMW96, CGM97]: the scripts comprise four basic operations which in practise are combined with expressions in XML query languages such as XPath or XQuery to navigate to the nodes that are to be updated:

1. $\text{insert}(n, f, k)$ add a leaf $n$ as $k$th son to node $f$,
2. $\text{delete}(n)$ remove a leaf $n$,
3. $\text{update}(n, v)$ change a value of node $n$ to $v$,
4. $\text{move}(n, m, k)$ a node $n$ into the position of the $k$th son of $m$.

We also view these operations as representatives for traversals that are defined in the DOM standard [W3C98a]. Note that [CRGMW96] do not assume the presence of object identifiers; in our case, these identifiers are provided by either the database or the source data (or both) so that we can make use of this feature at no cost. Following our example, an edit script could insert additional subtrees that describe textures in the images or delete items that appear twice in the database. Typically, an edit script first pins the location of nodes to be changed; this is done by navigating through the syntax tree as object identifiers in the database are often not accessible to other applications. Once the location is found, the script then applies update, delete, and insert operations. Conceptually, an edit script may do two kinds of changes: systematic and local changes. Systematic changes may become necessary if a faulty application produced
razor \((OID \ roots)\)
\[
o_1, \ldots, o_n := \text{offspring\_relations}(roots)
\]
\[
\text{foreach } o \in \{o_1, \ldots, o_n\} \text{ do}
\]
\[
\text{if } (\text{offspring}(o) \neq \emptyset) \text{ do}
\]
\[
\text{razor}(o)
\]
\[
\text{end}
\]
\[
\text{end}
\]
\[
\text{delete}(o_1, \ldots, o_n, roots)
\]
\[
\text{end}
\]

Figure 4.9: Bulk deletion algorithm

data with errors that are spread over the whole database; in this case, the edit script traverses large parts of the syntax graph and applies changes. In the relational context of our work, this may be an expensive restructuring process. On the other hand, if changes are only local, the script just visits a small number of nodes and patches them. This should not be a resource-intensive problem, neither in relational, object nor native systems.

Since this is not the place to discuss edit scripts in depth, we hence refer the reader to the above citations. However, we demonstrate their use with an example similar to that used in the performance discussion. Consider again Figure 4.2. A systematic change would, for example, require us to change all dates from Unix system time, \textit{i.e.}, seconds since January 1 1970, to a more human readable format. The way we go about creating the appropriate edit script is the following: We look up all associations which assign a value to an attribute \textit{date}. Then, for all these nodes, we calculate the new date and replace the old one. Techniques for constructing automata that do the traversal can be found, \textit{e.g.}, in [NS99]. Once such an automaton finds a node \(n\) that needs to be updated, it executes an \textit{update}(n, \textit{new date format}) statement. On the physical data model Monet XML this is translated into a command that replaces the value of the respective association.

The point that is important for us is that edit scripts traverse parts of the XML syntax graph and manipulate individual nodes. This is in stark contrast to the second method mentioned above, bulk deletion and re-insertion, where we delete a complete segment of the database and re-insert a corrected version. In the example scenario, this means that an individual detector re-sends the corrected version of a previously submitted analysis instead of a patching edit script. Generally, the underlying assumption is that the aforementioned data sources provide the capability of sending both the edit-script and a complete updated document; however, this assumption holds for many practical applications as well as for our example: a detector may either send an edit script or re-transmit a corrected version of the complete document. Additionally, all data items have a unique identifier which then can be used as an orientation to replace the automaton that guides the edit script by algebraic joins which were been shown to have a more efficient execution model [SKWW00a].

The algebraic algorithm that deletes a complete database segment with root \(r\) is displayed in Figure 4.9. Note that this algorithm is efficient because it visits every node once in a breadth-first search like manner, imitating a single scan over the relevant parts of the document (for simplicity, we left out the deletion of rank information). The complementary question, how to translate an edit script into algebraic insert and delete
operations, is rather straightforward: the trick is to dump those parts of the database that are to be inserted or updated into relations and then add those to the database.

Still, we need to discuss when it is advantageous to use bulk deletion combined with re-insertion and when to use edit scripts. The next section looks quantitatively at when to go for what.

### 4.1.6 Quantitative Analysis

This section presents performance impressions of a data warehouse containing actual features more elaborate but similar to the ones used in the example. We assume that the data warehouse uses Monet XML as the physical storage model; thus, our results may need slight modifications if applied to systems that use other data models. However, we believe that relational database management systems should behave in a similar manner as our implementation on top of the relational Monet database kernel [BK99].

Figure 4.10 displays the relationship between database size and insertion speed. The figure contrasts the naïve approach with the optimisation using the schema tree. As one might expect the insertion into an empty database is faster than into an already densely populated one if no intelligent caching is used. As the database gets larger, insertion speed converges to a ratio of about 390 KB/sec, where the constraining cost factor is indeed excessive string hashing. If schema trees are used, bulk load speed more than triples, showing the potential of this technique. Note that neither bulk load method blocks the database; both operate interactively and do not interfere with the transaction system. In terms of overall performance, it is interesting to remember that path caching makes insertion performance independent of the depth of the tree whereas in the case of the naïve approach performance degrades the deeper the tree is. Additionally it is worth noting that only a small portion of the overall insertion time is spent on actually parsing the document, namely 3%; the remaining time consists of database internal operations only.

Bulk deletion is assessed in Figure 4.11. The algorithm presented in Section 4.1.5 is run against the database created during the previous experiment. For each run, segments of around 55 MB are deleted. Note that the insertion performance in Figure 4.10
includes converting the textual representation of a document to executable database statements and, thus, random memory accesses (which can be alleviated with schema tree-base path caching), whereas deletion can be done as sequential scans.

Eventually, with respect to when to choose which technique, the two lines in Figure 4.12 show that once more than approximately 220 entries are changed by the edit script, one should consider reverting to bulk operations for performance reasons. The threshold of 220 entries is surprisingly low; however, one should keep in mind that relational databases are not optimised for pointer-chasing operations. We also remark that the threshold also depends on the characteristics of the XML document, especially on the ratio between text and markup. Nevertheless, it does not vary greatly for different types of documents.

4.2 A Query Algebra

In this section, we discuss how documents stored in Monet XML can be queried with the relational algebra such as discussed in [AHV95] and put in the Monet context in [BK99]. While basic queries can be expressed by assigning the usual SQL bag semantics to the association tables that were generated by the Monet XML bulkload procedure described in the previous section, probably the most salient feature of XML query languages, regular path expressions, is not expressible this way. To overcome this lack of expressiveness we present a preprocessing technique which translates regular path expressions into the plain relation algebra given the path summary of a database instance and thus makes it possible to use existing query optimisers and execution engines on XML queries without extensions. By building on the relational algebra, we primarily aim at bulk retrieval but other types of queries are supported as well. Another design advantage is that we inherit the simplicity and minimality of the relational algebra along with its rule set for query transformation.

![Figure 4.11: Bulk deletion](image-url)
4.2.1 Features of XML Query Languages

Although this thesis is not the place to discuss in detail the requirements and features of XML query languages, it is still useful to look at what makes them different from SQL [Ame86], which is the standard interface to query processors based on the relational algebra. Then we analyse the requirements to identify how the algebra we present can be extended to act as a fully functional query language.

[CFMR01a] define some general requirements of XML query languages. These requirements reflect the tendency to extend the role of query languages beyond what they have been in past settings. For example, relational databases are queried through SQL or Query By Example (QBE) interfaces, both of which are made for human users. The role of XML as a machine readable data interchange format also necessitates a machine-readable version of the query language. Therefore, it makes sense to define more than one syntax for the logical query model to support both machine-read and human-readable formats. Furthermore, to deserve the designation query language an XML query language has to be declarative; this means that it should describe queries on the logical level rather than by algebraically enforcing a particular strategy of evaluating the query.

On the technical level, a query language should be independent of protocols so that queries do not depend on the physical infrastructure of the World Wide Web or the database server. Since so much effort has been made to provide and standardise lowest common denominators like URIs, on which XML and its infrastructure depend, it is not desirable for the query language to depend on more than these lowest common denominators.

Since one important use of query languages is the role of embedded languages in program code, an XML query language should provide a set of standard error conditions like exceptions to signal to the host application that expressions cannot be processed. This could be due to syntactical or logical errors in the expression or the unavailability or failure of external resources such as network or external functions.

Additionally, an XML query language should be extensible in the sense that it is
open for additional functionality that goes beyond querying. Updates and transactions are critical for many applications but still not part of any standard. Since some XML data models can define infinite document instances, fixed point computations are useful for these cases; however a query language is only required to be defined for finite instances.

Although XQuery seems to be the current frame of reference in XML querying, a number of alternative approaches are available and may catch on in the future. Therefore, we do not pretend to present an algebra that is capable of serving as a basis for an implementation of XQuery. Rather we try to focus on the features that separate XML query languages from relational query languages and show how to implement the former with a basic relational algebra and the additional information provided by the Monet XML mapping. This means that the algebra which we present in the next subsection has two important properties: First, it is closed under composition, \textit{i.e.}, the result of a sub-query can be bound to a variable in the enclosing query. Second, it is set-oriented, \textit{i.e.}, it aims at processing of large data volumes.

### 4.2.2 Overview of the Algebra

This subsection gives an overview of the query algebra mainly by explaining its syntax. Figure 4.14 gives a listing of the grammar productions. The algebra is a simple extension of basic relational algebras [AHV95] with function application, intersection and path expressions.

The semantics of most of the operations are standard and straightforward. Like many algebras in the database world, the functions are either singleton or binary. The selection operator $\sigma_p(R)$ filters out those tuples in a relation $R$ for which the predicate $p$ does not hold. The projection operator $\pi_A(R)$ only keeps the attributes contained in the set $A$ from the tuples in the relation $R$. The binary operators $\cup, \cap$ and $\bowtie_p$ are the well-known union, intersection and equi-join operators, where $p$ again is a predicate; $\text{map}_f(R)$ applies the side-effect free function $f$ to all tuples in $R$. It will mainly be used to convert tuples to XML notation and to cast the type of attributes. The salient feature of the algebra is the production $\text{pathexpr}$, which can be interpreted as follows: while
the other operators work in the document tree in a horizontal manner, path expressions help to query the tree vertically, i.e., along the tag hierarchies. Note that the set of operations we presented is not minimal. This means that it is possible to express certain operators by combinations of other operators. Also note that the algebra does not allow expressions to be substituted by database relations; this is done automatically by the query compiler. The only way to navigate through the hierarchies of documents is by means of path expressions.

Example 10. Consider the example document of the previous section in Figure 4.2, whose schema tree is displayed in Figure 4.7. Suppose we want to extract all histograms and the key of the corresponding image. The following expression could be used to do just that:

\[
\text{image}([\text{key}] \bowtie_{\text{hd} = \text{hd}} (\text{colour}/\text{histogram}/cdata[\text{string}]))
\]

This translates to the plain algebra in a straightforward manner:

\[
\begin{align*}
\text{R1} & \bowtie_{\text{tl} = \text{hd}} (\text{R2} \bowtie_{\text{hd} = \text{hd}} \text{R6} \bowtie_{\text{tl} = \text{hd}} \text{R7} \bowtie_{\text{tl} = \text{hd}} \text{R8})
\end{align*}
\]

The structure of the plain algebra expression resembles that of the original query. For orientation, note the following way of reading the original query: \textit{In the database, navigate to all nodes which carry an image tag. There join the key attribute with the string found at the end of the paths along the tags colour and histogram.} We only need two different kinds of join attributes in the query. If we follow a hierarchical path, we join parent-child relations on the OIDs that refer to each other, this is denoted by $\bowtie_{\text{tl} = \text{hd}}$ following Monet speak. If we want to combine objects with a common ancestor we use the (semi-)join $\bowtie_{\text{hd} = \text{hd}}$.

We now turn our attention to regular path expressions and how they can be replaced with operators from the plain relational algebra.

4.2.3 Compilation of Regular Path Expressions

While Regular Path Expressions (RPEs) are one of the most powerful features of XML query languages, there is still research going on how to evaluate them efficiently in
general settings. We now present a two-step evaluation scheme that (1) enables efficient execution of a restricted class of RPEs in the relational algebra and (2) opens up this class of RPEs to query optimisation.

Once a query front-end produces an algebrised version of the input query, query execution consists of the steps outlined in Figure 4.15: during preprocessing all wildcards in regular path expressions are eliminated and replaced with join and union operations; the query can then be handed on to a conventional query optimiser and execution engine. In the preprocessing step, we eliminate wildcards in regular path expressions by keeping track of the current context of a query node and replacing a wildcard with the paths that match the wildcard in the current context. The following algorithm eliminates all wildcards in path expressions from the input tree.

**Example 11.** This example query is an extension of the previous one. Now we are not only interested in the colour histogram but also in all string data below the colour node:

```
image([key] => hd (colour//cdata[string]))
```

Using the algorithm in Figure 4.16 this translates to:

```
R1 => tl=hd (R2 => hd=hd (R6 => tl=hd R7 => tl=hd R8
            R6 => tl=hd R9 => tl=hd R10
            R6 => tl=hd R11 => tl=hd R12))
```

In this example, the regular path expression `colour//cdata[string]` extends to the three paths

1. `colour/histogram/cdata[string]`,
2. `colour/saturation/cdata[string]`, and
3. `colour/version/cdata[string]`.

Note that once a path expression evaluates to more than one path, all its child path expressions possibly also evaluate to more than one path.

During the translation, new union statements are only introduced if not all paths are fully specified or otherwise non-unique. After the elimination of path expressions, the query is optimised for efficient execution. Since our algebra is essentially a relational algebra extended with path expressions, relational optimisation techniques can be applied. Note also that the presence of path expressions makes query optimisation feasible for a larger class of queries than a plain translation of path expressions into

![Figure 4.15: Phases of query compilation](image)
procedure eliminate (context c, expr e) : expr
    if e is pathexpr then
        replace e with union of all matching relations according to c
        the new context cr is concatenation of c and e
    end
    else
        the new context cr is concatenation of c and e
    end
    ∀ child e1 of e : eliminate e1 cr
end

Figure 4.16: Algorithm to eliminate regular path expressions schematically

FOR/LET Clauses
    Ordered list of tuples
    of bound variables

WHERE Clause
    Pruned list of tuples
    of bound variables

RETURN Clause
    Instance of XML
    Query data model

Figure 4.17: Data flow in XQuery (adapted from [CFR+01])

joins, which would be functionally equivalent. This is because large numbers of joins
tend to enlarge the number of optimisation and reordering opportunities beyond what
current optimisers are capable of handling.

Figure 4.17 shows the data flow in an XQuery processor as anticipated by the de-
signers of the language. In such an engine, path expression come in two flavours: just
as we found it useful to introduce strong and weak associations, we also distinguish
between strong and weak path expressions. Whereas strong path expressions are eval-
uated with join semantics, weak path expressions bear outer join semantics. For the
purpose of this algebra, path expressions in the first phase of query execution are all
strong, whereas those in the second part are all weak. Note that this restriction does
not reduce the expressiveness of the algebra.
4.3 The Meet Operator

The preceding section presented queries in algebraic form; in general, formulating such queries requires a fair amount of familiarity with the underlying database structure. However, due to the ubiquity and popularity of XML, users often are in the following situation: they want to query XML documents which contain potentially interesting information but they are unaware of the markup structure that is used. For example, it is easy to guess the contents of an XML bibliography file whereas the markup depends on the methodological, cultural and personal background of the author(s). Nonetheless, it is this hierarchical structure that forms the basis of XML query languages.

In this section, we equip users with a powerful tool, the *meet operator*, which exploits the tree structure of XML documents and lets them query databases with whose content they are familiar, but without requiring knowledge of tags and hierarchies. The approach is based on computing the lowest common ancestor of nodes in the XML syntax tree: e.g., given two strings, we are looking for nodes whose offspring contains these two strings. The novelty of this approach is that the result type is unknown at query formulation time and dependent on the database instance. If the two strings are an author's name and a year, mainly publications of the author in this year are returned. If the two strings are numbers the result mostly consists of publications that have the numbers as year or page numbers. Because the result type of a query is not specified by the user we refer to the lowest common ancestor as *nearest concept*. For illustration we also present a running example taken from the bibliography domain, and demonstrate that the operator can be implemented efficiently.

4.3.1 Introduction

As laid out in the Introduction to this thesis, XML takes the idea of markup further than HTML: it is not used for visual representation of data, but for encoding semantics in documents which makes not only a document's content particles but also the tags and the hierarchies they form an interesting target for query languages. In contrast to other hierarchical data models (see [AHV95]) like the complex data models or the object-oriented models, XML is an incarnation of the semistructured paradigm, which means that the database schema that results from the mapping of a document to a database instance tends to be large and irregular. It may not be immediately clear which parts of the database obey which part of the schema. All this hinders *ad hoc* users and non domain experts in posing meaningful queries, as state-of-the-art query languages likely do not fully capture the loose schema of many XML data and require the specification of schema templates.

Basically all XML query languages and especially those that we introduced in this thesis support some flavour of schema wildcards and, thus, relieve the user of the burden of having to specify the complete paths to the data. Good and early examples for this strategy are XML-QL [DFF*98], XQL [RLS98] and Quilt [CRF00a]. The commonest way to accomplish this is to allow the specification of sets of paths with path expressions that are evaluated against the actual database. However there are cases when path expressions do not provide the power necessary to retrieve the intended results. Consider the following situation taken from the area of bibliographic databases: A user wants to know what 'Ben Bit' edited or published in '1999', *i.e.*, find the relevant publication record(s) in an XML bibliography, but hasn't got any knowledge of the schema of the XML file sketched in Figure 4.18. Therefore the user may try the following query:
Querying XML Data

FOR $t$ IN document("database.xml")
WHERE $t$ CONTAINS 'Bit'
AND $t$ CONTAINS '1999'
RETURN <result>
    $t$
</result>

The query binds $t$ to all nodes in the syntax tree in Figure 4.18 and filters those nodes whose offspring contains as character data the string 'Bit' and '1999'. Evaluated against the example document shown in Figure 4.18 the answer looks like (the OIDS of the nodes in the database that correspond to the result record are given in brackets):

<answer>
    <result>
        <article key='BB99'> ... <article>
    </result>
    <result>
        <institute> ... </institute>
    </result>
    <result>
        <bibliography> ... </bibliography>
    </result>
</answer>

Although the answer contains the desired result, it suffers from a serious drawback: we are only interested in a subset of the answers the database generates. Some not so interesting answer elements are implied by the path from the first node that is bound to $t$, to the root node: they are ancestor nodes of this first node (e.g., the institute and the first bibliography elements in the answer set are implied by the article element). Even worse, in larger databases the computation might cause a combinatorial explosion of the result size.
4.3 The Meet Operator

One solution to the problem is to refine the query. In general, this involves a fair amount of domain knowledge that cannot be expected of ad hoc users. Thus it would be desirable to give users the opportunity to query the database without requiring domain knowledge. To achieve this we define a special operator, the meet operator, which gives the user more control over the results generated by such queries. For two nodes in the syntax tree \( o_1 \) and \( o_2 \) the meet operator \( \text{meet}(o_1, o_2) \) simply returns the lowest common ancestor of nodes \( o_1 \) and \( o_2 \), which we call the nearest concept of \( o_1 \) and \( o_2 \) to indicate that the type, i.e., element name or tag, of the result is not specified by the user but data-dependent on the stored document trees. Informally, this node implies all other possible answers. By suitably extending this operator to work on sets of nodes and adding it as a declarative construct to our query language we give the user an opportunity for explorative querying even if he or she has only little or no knowledge of the database schema and content. As [Abi97, GM99] point out, there is always the notion of a schema in semi-structured or XML databases, but it may be large, unknown or implicit and therefore opaque to the user.

While the semantics of the operator for two objects are intuitive, it is less clear what happens when there are more than two nodes. This is the case if it is applied to the result of a full-text search. If we apply the original motivation to such an input we will end up with a combinatorial explosion of the result size. Therefore we will also present a generalisation of the operator that is tailored towards large amounts of nodes: it delivers both intuitive results and has an efficient execution model.

The rest of this section is structured as follows: in 4.3.2 we formalise the notion of meet for various inputs and also present algorithms. Subsection 4.3.3 expands on these ideas. Then we assess the performance of the algorithms presented and conclude with some general remarks.

4.3.2 Nearest Concept Search

We now formalise the semantics of the meet operator in terms of the data model of the previous section. We start from the simple case of finding the meet, denoted \( \text{meet}_p \), of a pair of nodes to the more sophisticated case of applying the meet to a set of objects such as the results of a full-text search.

The Meet-Operator

To simplify the discussion, we abstract from the example query given in the introduction for the time being and limit ourselves to the basic question: Given two nodes \( o_1 \) and \( o_2 \) in the syntax tree, how can we calculate \( \text{meet}_p(o_1, o_2) \). Later, we come back to the initial question and extend on it.

We now formalise and generalise the ideas sketched in the introductory example. First, we borrow some notation to denote offspring relationships in the schema and in the database instance.

**Definition 12.** We denote the list of object identifiers encountered on the path from the document root to a node \( o \) in the syntax tree with \( \text{context}(o) \).

For example, in Figure 4.18 \( \text{context}(o_4) = [o_1, o_2, o_3, o_4] \). Additionally, we can now define the relationship \( \preceq \) between contexts.

**Definition 13.** We now write \( \text{context}(o_1) \preceq \text{context}(o_2) \) if \( \text{context}(o_2) \) is a prefix of \( \text{context}(o_1) \) (including the equality of \( \text{context}(o_1) \) and \( \text{context}(o_2) \)). Analogously, we
write \( \text{path}(o_1) \leq \text{path}(o_2) \) if \( \text{path}(o_2) \) is a prefix of \( \text{path}(o_1) \) (again including the equality of \( \text{path}(o_1) \) and \( \text{path}(o_2) \)), where \( \text{path} \) is defined as in Chapter 3.

The basic difference between \( \text{context}(o) \) and \( \text{path}(o) \) is that the latter only provides schema information (the tag sequences) whereas the former includes parts of the actual database instance; another dissimilarity is that in a given association \( (o, \cdot) \), \( \text{path}(o) \) comes for free by looking at the name of the relation; on the other hand, deriving \( \text{context}(o) \) in general requires joins to be computed. We now use \( \text{context} \) to interrelate any two objects in a document tree:

**Definition 14.** Let \( o_1, o_2 \) and \( o_3 \) be objects in an XML syntax tree. Then \( o_3 = \text{meet}(o_1, o_2) \) iff

1. \( \text{context}(o_1) \leq \text{context}(o_3) \),
2. \( \text{context}(o_2) \leq \text{context}(o_3) \) and
3. \( \exists o_4 : \text{context}(o_4) \leq \text{context}(o_2) \land \text{context}(o_1) \leq \text{context}(o_4) \land \text{context}(o_2) \leq \text{context}(o_4) \).

Note that \( \text{meet} \) does not depend on the order of its arguments. Eventually, we identify the following semantics with the \( \text{meet} \): The nearest concept of objects \( o_1 \) and \( o_2 \) is \( \text{path}(\text{meet}(o_1, o_2)) \).

**Example 15.** We now have a look at some examples to demonstrate how to use and interpret the \( \text{meet} \) operator.

1. Suppose we do a full-text search for "Ben" and "Bit" on the example document. The resulting associations are \( a_1 = A(o_6, "Ben") \) and \( a_2 = B(o_8, "Bit") \) (we abbreviate the relation names with \( A \) and \( B \); the full names are easily recovered by looking them up in Figure 4.18). After calculating \( \text{meet}(a_1, a_2) = o_4 \) we find that the two associations constitute an author's name.

2. Furthermore, a full-text search for "Bob" and "Byte" returns the associations \( a_1 = A(o_{15}, "Bob Byte") \) and \( a_2 = A(o_{15}, "Bob Byte") \). In this case, we find that trivially \( \text{meet}(a_1, a_2) = o_{15} \), which is the very character node that the full-text search returned. Fortunately, the hierarchical information included in the Monet XML model immediately exhibits that the character data node is a son of an author node.

3. When searching for "Bit" and "1999" the full-text search returns the associations \( a_1 = A(o_8, "Bit") \), \( a_2 = B(o_{12}, "1999") \) and \( a_3 = B(o_{17}, "1999") \). Similarly, \( \text{meet}(a_1, a_2) = o_3 \) reveals that Mr "Bit" published an article in "1999"; however, \( \text{meet}(a_1, \text{meet}(a_2, a_3)) = o_2 \) only reveals that the three associations are located in the bibliography of an institute. We therefore will discuss variants of the \( \text{meet} \) operator to produce more intuitive results and filter out trivial or counter-intuitive ones.

We now consider a variety of interpretations that can be applied to query results obtained with the \( \text{meet} \) operator in general. These possible views make the \( \text{meet} \) a useful construct in many different application domains. The following enumeration deals with two argument objects only, but the reasoning extends to a larger set of objects as well. Note that we assume \( o = \text{meet}(o_1, o_2) \), i.e., that \( o \) is the result of a query \( \text{meet}(o_1, o_2) \) and therefore that \( \text{context}(o) \) is a prefix of both \( \text{context}(o_1) \) and \( \text{context}(o_2) \):
4.3 The Meet Operator

function meetP (oid o1, oid o2): oid
if o1 = o2 then return o1
else
  case
  path(o1) ≤ path(o2): return meetP(parent(o1), o2)
  path(o2) ≤ path(o1): return meetP(o1, parent(o2))
  default: return meetP(parent(o1), parent(o2))
end
end

Figure 4.19: Function meetP for a pair of OIDs

1. context(meetP(o1, o2)) is the longest common prefix of the two contexts of o1 and o2, context(o1) and context(o2).

2. context(o1) − context(o) and context(o2) − context(o) describe the context of o1 and o2 with respect to o. Depending on the overall schema, this may describe a part of o or is a relationship or a sequence thereof. (For two paths p1 and p2, p1 prefix of p2, p2 − p1 denotes the elements of p2 that are not included in p1.)

3. context(o1) − context(o) as well as context(o2) − context(o) describe the different contexts we see while traversing from o1 to o2 or vice versa. Trivially, this is also the shortest path from o1 to o2.

4. We can also interpret the path(meet(o1, o2)) as the smallest enclosing context of the input objects.

5. Finally, meetP(o1, o2) is the first node on context(o1) and context(o2) that contains both o1 and o2, i.e., the nearest concept of both nodes.

Computation

In this section we present the basic algorithms to compute meetP and two useful generalisations. Note that all algorithms in this section take advantage of the physical properties of our data model Monet XML. The prefix order among the paths is used to steer the search for the lowest common ancestor so that superfluous look-ups are avoided.

The algorithm displayed in Figure 4.19 computes meetP(o1, o2) for two objects and will be used as a building block for more general cases. The function parent(o) returns the parent association of the node or association o, basically a hash look-up. A remark on the case clause: we avoid unnecessary look-ups by comparing path(o1) and path(o2) to find the meet of these two objects as fast as possible and to steer the search direction of the algorithm. As pointed out in Chapters 3 and 4 this information is provided with only little additional cost at bulk load time.

The previous algorithm operated on two object identifiers. The next step we take is to generalise meetP to work with sets of OIDs O1 and O2 where all associations in O1 are of the same type, i.e., there is a path p in the path summary that ∀o ∈ O1 : path(o) = p. With this set-up, we may generalise the previous algorithm to
procedure \text{meet}_S \ (OID \ O_1, OID \ O_2) : OID \\
\text{for} \ i = 1 \text{ to } 2 \\
\quad r := \{o | \bigwedge_{j=1,...,n} O_i(o, o_j)\} \\
\quad \text{add to result } r \\
\quad O_i := O_i - r \\
\text{end} \\
\text{if } O_1 = \emptyset \text{ or } O_2 = \emptyset \text{ then return } \\
\quad I := \emptyset \\
\text{if } I \neq \emptyset \text{ then add to result } I \\
\quad O_1 := O_1 - I \\
\quad O_2 := O_2 - I \\
\text{case} \\
\quad \text{path}(O_1) \leq \text{path}(O_2) : \text{add to result } \text{meet}_S(\text{parent}(O_1), O_2) \\
\quad \text{path}(O_2) \leq \text{path}(O_1) : \text{add to result } \text{meet}_S(O_1, \text{parent}(O_2)) \\
\quad \text{default} : \text{add to result } \text{meet}_S(\text{parent}(O_1), \text{parent}(O_2)) \\
\text{end} \\
\text{end}

Figure 4.20: Procedure \text{meet}_S for two sets of OIDs

what is displayed in Figure 4.20. This time, the function \text{parent}(O_1, O_2) is a shortcut for \text{join}(O_1, O_2), a binary join on associations \(A_1(o_1, o_2)\) and \(A_2(o_2, o_3)\) so that \(\text{join}(A_1(o_1, o_2), A_2(o_2, o_3)) = A(o_1, o_3)\) (the inner columns are projected out, leaving a binary relation – association in our terminology; see [BK99] for an overview of the Monet algebra, which underlies our algorithm). Note that we avoid a combinatoric explosion of the result size as \text{meet}_S computes minimal meets, i.e., as soon as the first meet of \(o_1, o_2, \ldots \in O_1 \cup O_2\) is found subsequent meets are not considered anymore because the elements are removed from the input sets. This generalises the minimality criterion (3) of Definition 14 to sets of objects while still being invariant of the input order. Also note that we slightly extended the definition of meet: we now call a node meet if it is the lowest common ancestor of at least two other nodes. A salient feature of this and the following algorithm is that they make heavy use of the relational operations of the underlying database engine. In the analysis we will see they indeed perform favourably.

We now present the most general algorithm of this paper: it calculates the meet of an arbitrary input set of nodes grouped into relations \(r_1, \ldots, r_n\) according to the type of association they represent. This approach proves useful when we want to combine the results of full-text queries, which may be distributed over a large number of relations, i.e., we extract from the results of the full-text query starting points from where the user can start displaying and browsing the database. The algorithm is displayed in Figure 4.21.

In contrast to the previous algorithm, we cannot simply exploit the function \(\leq\) to compare the paths to steer the search, because then the algorithm would become dependent on the input order, as the algorithm does not know which subtrees of the document instance are being searched at a particular moment. Therefore, we rather roll up the tree-shaped schema from the bottom by iteratively contracting the offspring of nodes whose only offspring are leaves until we reach the root or the empty set. This way, all nodes that are meets of other nodes are minimal by construction; they are output and
procedure meet ([r₁, ..., rₙ]) : OID
if (n = 0) or (n = 1 and |r₁| = 1) then return
for i = 1 to n
  r := {o | \( j = 1, ..., o \in r, r \subseteq r \)}
  rᵢ := rᵢ ∩ r
  add to result r
end
let n be a relation all of whose children are leaves
w. l. o. g. let r₁, ..., rₙ (n ≥ l ≥ 1) be the children of n
(r₁, ..., rₙ) := (parent(r₁), ..., parent(rₙ))
p := r₁
meets := ∅
for i = 2 to l
  hits := p ∩ rᵢ
  rᵢ := semijoin(rᵢ, hits)
  meets := meets ∪ hits
  p := p ∪ (rᵢ - hits)
end
add to result hits
remove empty rᵢ
meet(r₁, ..., rₙ)
end

Figure 4.21: Procedure meet for arbitrary sets of objects

not considered anymore, thus, avoiding a combinatorial explosion of the result set and dependence on the input order.

Coming back to the example query, we see that after reformulating the query with the meet operator the cardinality of the answer set reduces (from now on, we interpret the meet operator as an aggregation operation):

MEET (FOR $t₁ IN document("database.xml"), $t₂ IN document("database.xml")
  WHERE $t₁/text()=’Bit’
  AND $t₂/text()=’1999’
  RETURN $t₁, $t₂)

Evaluated against the example document we now obtain the following result, a true subset of what the solution presented in the introduction with regular path expressions returned:

<answer>
  <result> article </result> (o₃)
</answer>

The generated answer now resembles our initial intuition and with some domain knowledge gained by looking at an appropriate visualisation of the answer the user can interpret the result as follows: Mr. Bit wrote an article in 1999.
Since the XML standard provides IDs and IDREFs, documents may make use of references and potentially break the tree structure defined by the element relationships. The algorithms we presented only cover element relationships as we believe that they often carry very natural semantics and because the design of the meet algorithms remains clear and intuitive while execution times enable interactive querying. If we interpret the meet operator as some variant of nearest neighbour search, we might find generalisations on graph structures that prove useful in certain application domains. However, the fact that we then have to take care of circular structures may add significant complexity to our algorithms.

Finally, we remark that the meet operator is not expressible in the relational algebra: We need stratified datalog [AHV95] to calculate it since the computation involves both a fixed point and a minimality criterion.

### 4.3.3 Extensions and Applications

In large databases our algorithms may still deliver too many unintuitive results. In this section we propose variations of the meet operator to gain more control over what the operator returns. In particular, we propose to extend the meet operator with two parameters: (1) a maximum distance that says how many edges may lie between two input objects, and (2) restrictions of the type of results, i.e., if \( o \) is a result candidate we restrict \( path(o) \) to be outside a certain set of paths \( R \); if \( path(o) \in R \) we discard \( o \):

\[
meet_R(r_1, \ldots, r_n) = \{ o \mid o \in meet(r_1, \ldots, r_n) \text{ and } path(o) \notin R \}
\]

For example, by setting \( R \) to \{bibliography\} we can filter out uninteresting matches, i.e., where the meet corresponds to the document root, in large bibliographies. This variant is also used in the case study in Section 4.3.4.

Another interesting application of the operator is distance calculation: since, in the meet algorithm, each join between two relations can also interpreted as a moving one edge closer to the root, i.e., a traversal of length one, the number of joins executed while calculating \( meet_P(o_1, o_2) \) for two nodes \( o_1 \) and \( o_2 \) corresponds to the number of edges on the shortest path from \( o_1 \) to \( o_2 \). So we can define:

\[
d(o_1, o_2) = \text{number of joins when calculating } meet_P(o_1, o_2).
\]

Building on this we can define another restricted version that is occasionally useful to block undesired matches:

\[
k-meet_P(o_1, o_2) = \begin{cases} \bot & \text{if } d(o_1, o_2) > k, \\ meet_P(o_1, o_2) & \text{otherwise}, \end{cases}
\]

The number of joins is also a simple yet both effective and intuitive heuristic for establishing a ranking between the result OIDs: the smaller \( d(o_1, o_2) \) the closer the elements are together in the document tree. In many applications, this can be used as a measure of how related the nodes are.

We believe that it is worthwhile to apply additional heuristics like distances in the source file or even more complicated information retrieval techniques to improve the ranking of the answer set. In particular, thesauri are a promising tool to help a user find interesting results, especially to broaden a search that returned too few answers.

Additionally, we mention a convenient application of the meet operator: staying in the bibliography domain, we may want to know whether a certain bibliographical item
that we found in one bibliography also lives in another bibliography; however, we have no idea how the relevant information is marked up. So a good approach is to combine the meet operator with full text search similar to the introductory example and use the results as a starting point for displaying and browsing.

### 4.3.4 Performance

We now assess some performance characteristics of two versions the meet operator: \textit{meet}_{p} and \textit{meet}. We will see that the costs of these operators are negligible if they are used in combination with a relatively selective full text search and that the set-oriented version of the operator scales well, \textit{i.e.}, linearly, with respect to the cardinality of the input sets.

We implemented the meet operator like most of the algorithms presented in the first four chapters of this thesis on top of the Monet XML module within the Monet database server. The measurements were carried out on a Silicon Graphics 1400 Server with 1 GB main memory, running at 550 MHz. Two XML sources were used: a file of about 200 MB with descriptions of multimedia data items, extracted by feature detectors [SWK99], and the DBLP bibliography, which is available on the Internet [Ley99]. For the first experiment the total main memory requirements of the database server were about 120 MB, the second experiment could be run in 100 MB. Note that only a fraction of the main memory was needed to compute the meet; most of it was necessary for our main memory DBMS to load relations and perform operations on them.

Figure 4.22 shows the run-time behaviour of a typical query such as the one presented in the introduction; however the underlying database is a file of descriptions of multi-media data items. In the plot, we normalised the duration of the full-text search to an average value as its execution varies greatly in relation to the little time the computation of the meet consumes. The figure shows two things: First, the execution time is dominated by the full-text search, which takes 1207 ms as opposed to the 2 ms the computation of the meet of objects with distance two. Secondly, the meet scales well with respect to distance of the objects. Therefore it can serve as a sensible and valu-
able add-on to an already existing search engine for semi-structured or XML data that comes at little cost.

Case Study

We now take a look at a meet query run against the XML version of the on-line DBLP bibliography [Ley99]. We prepared the bibliography by bulk loading it into Monet XML as described earlier in this chapter. We now want to list all publications in the Proceedings of the IEEE International Conference on Data Engineering (ICDE) of a certain year. To achieve this, we do a full-text search for the strings "ICDE" and the year and calculate the meets of the results according to algorithm meet<sub>R</sub> with the document root excluded from the set of possible results. To demonstrate that the algorithms scales we iteratively extend the search interval from 1999 back to 1984 (note that there was no ICDE in 1985, hence the small step at about 1100 on the x-axis), which gives us control over the size of the result set. The results resemble to a large degree our intuition and consist mostly of the ICDE publications of the respective year (there were just two false positives). The graph in Figure 4.23 shows the time elapsed for calculating the meet, e.g., for a result set of 1000 publications the computation takes about three seconds (the time the full-text search takes is not included in this figure). Note that the input sets are fairly large: they contain all associations whose string component contains the year, i.e., all publications in the bibliography between 1984 and 1999 are involved which is the bulk of the publications recorded in the bibliography. This demonstrates that the algorithm scales well to large datasets and is suitable for interactive querying.

We finally remark that the performance behaviour of the meet may differ on different underlying physical data models: not all XML-to-database mappings preserve as much information as the Monet model. However, we expect queries with small result sets to perform favourably on many relational models since they are usually optimised for the kind of join operations that dominate performance.
4.4 User-Defined Query Equivalences for SQL

This section takes the idea of using relational engines for querying XML data one step further. Since there are different physical representations for the same XML document — all good for different scenarios — it may well be sensible to store a document in more than one format and leave the choice to the query optimiser on which representation to execute a query. This section presents a mechanism to exploit domain knowledge about the physical properties of the data lay-out and inform a query optimiser about alternative strategies without having to extend it with new rules and operators. The resulting scheme is not only sensible for XML processing but also for a number of other applications such as Data Warehousing and Geographic Information Systems (GIS). Hence we draw the examples in this section from a large variety of application areas.

State-of-the-art query optimisers consider a rich set of alternative execution plans for a query, and pick one of them based on its estimated execution cost. The alternatives are generated using algebraic equivalence rules implemented in the optimiser. The rules follow from algebraic properties of relational operators and, additionally, often exploit constraints defined in the database schema. In many applications, however, the database instance obeys constraints that cannot be effectively declared using current constraint mechanisms. Yet the application writer knows about them and maintains them; hence, they could be exploited by the query optimiser during the generation of alternatives. For example, in spatial applications, a query that selects rows based on the predicate \( \text{PolygonOverlap}(T.p, \text{<target>}) \) may be speeded up significantly by introducing an extra join with a table \( T_B \) containing bounding-box information for polygons \( T.p \). However, the extra join may also reduce performance if on the other hand the query contains other predicates that are easy to evaluate: for example, if another predicate tests for equality on a key of \( T \), then the extra join is likely to introduce additional overhead. Current query optimisers are unable to make the above cost-based decision because this query equivalence is unknown to them.

In the rest of this section, we present a small but powerful extension to the SQL language or relational algebra that allows application writers to declare two query expressions to be equivalent. This mechanism allows the generation of alternatives in the query optimiser to be freely extended by applications. We describe the implementation of this extension in Microsoft SQL Server to show how to fully exploit user-defined equivalences. Our framework seamlessly integrates with the cost-based selection mechanism and thus guarantees that the optimal plan from the extended search space is chosen.

4.4.1 Introduction

Equivalences of relational expressions are basic to the design and operation of query optimisers. Earlier research on optimisers showed that join reordering and pushing down selects (i.e., exploiting algebraic properties of join and select) could bring order-of-magnitude performance improvements over naive execution of queries [SAD*79] and effectively made it possible to use high-level declarative query languages such as SQL in practise. Since then, other useful equivalences have been identified and opened up new ways to execute and optimise queries. There is a vast amount of literature on equivalences for relational expressions, dealing with operators such as semijoin, outerjoin, and group-by; for treatises of this kind, see [BC81], [CS94, YL95] or, more recently written, [GLR97, RLL*01] to name just a few. In each case, the equivalent ex-
pressions again can lead to order-of-magnitude performance improvements for specific queries and data distributions. Modern query optimisers know how to exploit the constraints declared in the schema definition and how to reap benefit algebraic rewriting properties of the underlying primitive operations; so even more equivalent alternative execution plans can be explored and a possibly more favourable plan be picked among them. Typically, 'check' constraints are used to, for example, restrict domain values of attributes, define column uniqueness, and declare foreign-key relationships. During optimisation, they can be used to detect empty subexpressions, simplify and reorder grouping operations and eliminate unnecessary joins as described in [SSM96, YL95].

In addition to the standard repertoire of the optimiser, application designers often are aware of and maintain equivalences that cannot be effectively declared to a DBMS using current constraint mechanisms. For example, consider an OLAP engine that uses a relational DBMS to store results of partial aggregations from sales data, a standard scenario in Data Warehousing. The source data are stored in a table `detail(amount,customer,date,product,store,...)`; derived are the tables sketched in Figure 4.24. Each customer is assigned a rating based on his total purchases. In the following, we list tables which are needed by users and which the OLAP engine might choose to materialise:

1. Total sales per customer:

   ```
   tableC(customer,sales,customerrating)=
   select   customer,sum(amount),
           customerrating(sum(amount))
   from     detail
   group by customer
   ```

2. Total sales per customer rating, product, store and sales:

   ```
   tableCrPS(customerrating,product,store,sales)=
   select   customerrating,product,store,sum(amount)
   from     detail d join tableC tC
   on     d.customer = tC.customer
   group by customerrating,product,store
   ```
3. Total sales per customer rating, product and date:

\[
\text{tableCrPD(customerrating,product,date,sales)} = \\
\text{select customerrating,product,date,sum(amount)} \\
\text{from detail} \text{ join tableC} \\
\text{on tc.d.customer = tc.customer} \\
\text{group by customerrating,product,date}
\]

If a user wants to retrieve the total sales per product and customer rating, either of the summary tables, tableCrPS or tableCrPD, could be used to answer the query, as sketched in Figure 4.24. But which strategy should be chosen depends on the actual characteristics of the database instance, namely the data distribution and the indexes available.

One approach is to make the dependencies explicit to the database system by materialising them with views. With a proper re-formulation of the input query, which should use the summary table, the query optimiser could decide which strategy by using the built-in cost models. However, it is in general not desirable to rely on the availability of the detail table since many application scenarios cannot guarantee that the database is kept on-line and up-to-date at all times. But this would be necessary because users need it to formulate queries and because the DBMS uses it to maintain the aggregations in the summary tables. Practically, it is compulsory to materialise tableC since otherwise stacked group-by operations for construction and maintenance of tableCrPS and tableCrPD are required; unfortunately, these types of views are generally not supported by mainstream database systems. To sum up, we are forced to store the part of the database which is framed in Figure 4.24. But these restrictions are rather artificial as they are imposed by the lack of functionality of the underlying storage system and since they increase the storage and processing overhead of the OLAP system because data are stored which are in fact not needed by the application.

If the OLAP engine chooses, as an alternative strategy, not to store the detail table, not enough information is available to communicate the data and query dependencies to the DBMS. Now suppose a situation which is typical for OLAP scenarios: we have to send multiple equivalent queries to the DBMS. We could go about this by using a heuristic to choose between the alternatives; if we use a sophisticated heuristic we should keep track of useful statistical characteristics of the data and index information. Unfortunately, this duplicates functionality which is already implemented in the query optimiser. Alternatively, we might submit some or all of the alternatives to the DBMS, request not the execution of the queries but rather cost estimation and eventually have the query with the lowest cost executed. Again, we would duplicate functionality for this approach that is already implemented in the optimiser of the DBMS, namely plan selection and query reordering, the very core functionality of an optimiser. Unfortunately, the very fact that the queries are equivalent is not available to the optimiser.

We propose now the use of a new SQL construct, CHOOSE, to declare equivalence between expressions. Syntactically, CHOOSE is similar to UNION or UNION ALL; semantically, its result is execution of any one its branches. The intended implementation of this operator is to have the cost-based optimiser choose the alternative with the least expected cost. In the implementation, which we describe in Section 3.4.5, the user-defined equivalence is seamlessly integrated into the generation of alternatives, along with all readily available equivalences built-in in the optimiser as outlined above. Cost-based selection is then applied on the extended search space.
For our OLAP example above, the total sales per product and customer ranking can be defined by the following view:

define viewCrP(customerrating, product, sales) as
select        customerrating, product, sum(sales)
from tableCrPS
groupBy customerrating
CHOOS E
select        customerrating, product, sum(sales)
from tableCrPD
groupBy customerrating

Such a view is especially useful in larger queries when it is convenient to leave it to the query processor to select the proper (partial) execution plan. For example, the following query requests the total sales for each customer rating and for a particular product:

select    customerrating, sum(sales)
from      viewCrPD
where     product='product#1026'

The query processor should now push down the selection through the alternatives and then choose the best alternatives according to which indexes are available. It would require a huge unnecessary development and probably also research effort to engineer this functionality outside the DBMS – let alone that we would only duplicate functionality already available inside the DBMS. By allowing the propagation of equivalences of expressions from the application to the query optimiser, a useful primitive is added for application-specific extensibility of optimisers which seamlessly integrates with the usual optimisation infrastructure.

We expect CHOOSE to be most useful for applications which support a complex domain or implement an additional level of semantics on top of the DBMS. The application would set up a number of tables, constraints, and triggers in the relational DBMS, and present an application-specific model to the user. Then it synthesises SQL from users’ queries and updates. Some simple examples from this class of application are the following:

- An OLAP engine supports a model of ‘cubes’ with detail data and dimensions. It may choose what partial results to pre-compute and store in the database. Each user query then can be mapped to different pre-aggregations, as shown in our earlier example. Details on an OLAP engine can be found in [Ise00].

- A Geographical Information System presents geometric data and operations to users. For implementation on a relational DBMS, it may use extensibility mechanisms to define domain-specific predicates such as PolygonOverlap. It may create additional tables with information synthesised from geometric objects such as bounding boxes, grid overlap, or Z-curve intersections to try to speed up query time for some queries. In general, the decision to use these additional tables (effectively an extra join in the query), should be done in a cost-based way. The CHOOSE operator can be used to declare the equivalence and let the query optimiser choose an efficient plan. For details on creating these auxiliary tables, and the precise mapping to SQL we refer to [FFS00].
Semistructured data such as XML can be mapped to relational tables in different ways. Each mapping has its own benefits and disadvantages in terms of query performance. Maintaining some redundant structures allows for efficient processing of a larger class of queries than any single mapping can, if the CHOOSE operator is available. Some of the many XML mappings are presented in [STH'99, SKWW00a]. In Section 4.4.4 we present a detailed example of the use of CHOOSE on XML querying.

### 4.4.2 The CHOOSE Operator

To integrate the CHOOSE operator into an existing query engine we now look at how to extend SQL [ISO99] with the necessary syntactic constructs. This turns out to be easy since we can use the same grammar production as well-known set operations like UNION or INTERSECT. Using the ANSI terminology of the SQL standard, the addition of a single production suffices:

\[
\text{\langle non-join query expression\rangle ::=}
\]

\[
\text{\langle non-join query term\rangle | \langle query expression body\rangle \text{ CHOOSE \langle query term\rangle}}
\]

There are a number of implicit semantic constraints that the abstract syntax tree of a SQL query has to abide by but that cannot be checked on the grammatical level. In the case of CHOOSE the following constraints have to be checked to fit the operator into the existing language framework of SQL:

1. CHOOSE has lower precedence than any other operator including set operators.
2. All inputs have to be UNION-compatible, which means that the number and type of columns of the tables involved must match.
3. Implicit conversion of types to achieve compatibility is applied when necessary.
4. The column names of the result table are taken from the first input.
5. CHOOSE may be used in view definitions.

These requirements coincide with those for the set operations UNION, INTERSECT, and EXCEPT. Therefore, the integration of CHOOSE into the language processing component of existing systems requires only moderate modifications and extensions to the infrastructure already in place. From an algebraic point of view, CHOOSE can be viewed as a binary operator which non-deterministically evaluates to one of its arguments and for which the laws of commutativity, associativity and distributivity hold (in the sequel, the \(E_i\) denote arbitrary relational algebra expressions):

1. **Commutativity:**
   
   \[\text{CHOOSE}(E_1, E_2) \leftrightarrow \text{CHOOSE}(E_2, E_1)\]

2. **Associativity:**
   
   \[\text{CHOOSE}(E_1, \text{CHOOSE}(E_2, E_3)) \leftrightarrow \text{CHOOSE}(\text{CHOOSE}(E_1, E_2), E_3)\]

3. **Distributivity:** other operators distribute over CHOOSE
   
   \[f(\text{CHOOSE}(E_1, E_2), E_3) \leftrightarrow \text{CHOOSE}(f(E_1, E_3), f(E_2, E_3))\]
Making use of these rules, we can rewrite any query containing CHOOSE as a non-deterministic choice of \(n\) queries which do not contain CHOOSE, i.e., for each expression \(E\) containing \(k\) CHOOSE operators there exists an equivalent expression of the form \(\text{CHOOSE(\text{CHOOSE}(\ddots, \text{CHOOSE}(E_1, E_2) \cdots, E_{n-1}), E_n))}\) such that no \(E_i\) contains CHOOSE. Furthermore, the number of expressions \(n\) is bound by \(k + 1 \leq n \leq 2^k\). To see why this is, consider a query with \(k\) CHOOSE operators. Using the commutativity, associativity and distributivity rules, the CHOOSE operators can be factored out and the query be transformed to the above expression. Note, that the number of CHOOSE operators might significantly increase compared to \(k\) since repeated application of the distributivity law duplicates CHOOSEs in the subexpression \(E_3\). As to the bounds of \(n\) we apply simple combinatorial counting arguments: Every CHOOSE can evaluate to either of its two arguments. Hence, there are at most \(2^k\) possible configurations. For the lower bound, consider the case where all CHOOSE are subsequently applied. Consequently, all CHOOSE but the first add one configuration which in turn establishes the lower bound. The following example illustrates the transformation process:

**Example 16.** The expression \(\text{CHOOSE}(A, B) \bowtie \text{CHOOSE}(C, D)\) can be transformed in the following way:

- \(\text{CHOOSE}(A, B) \bowtie \text{CHOOSE}(C, D)\)
- \(\text{CHOOSE}(A \bowtie \text{CHOOSE}(C, D), B \bowtie \text{CHOOSE}(C, D))\) (distributivity)
- \(\text{CHOOSE}(\text{CHOOSE}(C, D) \bowtie A), \text{CHOOSE}(C, D) \bowtie B)\) (commutativity)
- \(\text{CHOOSE}(\text{CHOOSE}(C \bowtie A, D \bowtie A), \text{CHOOSE}(C \bowtie B, D \bowtie B))\) (distributivity)
- \(\text{CHOOSE}(\text{CHOOSE}(C \bowtie A, D \bowtie A), C \bowtie B), D \bowtie B)\) (associativity)

Here the maximum bound \(n = 2^k\) is reached.

### 4.4.3 Algebraic Optimisation of User-defined Query Equivalences

Although the CHOOSE operator is in the first place a syntax extension which does not depend on any particularities of the query processor, the architecture of the query optimiser has significant influence on the integration of this operator into the infrastructure already in place. The most straightforward approach to optimising a query with CHOOSE is to re-formulate it as up to \(2^k\) different queries without CHOOSE by enumerating all alternatives as specified by the user. The resulting queries are then optimised separately and the best plan is then chosen for execution. Evidently, the time spent in optimisation may be increased by an enormous factor rendering this approach infeasible even for queries of only moderate complexity. We therefore need a way that makes better use of the available resources. Thus, in this section we discuss how to integrate CHOOSE into query optimisation without significant overhead or interference with the existing infrastructure. The key to achieving this goal is algebraic optimisation. In an algebraic optimiser framework, queries are represented by operators of an extended relational algebra. All optimisation is carried out on the basis of inference rules which describe how a single operator or group of operators can be transformed into a semantically equivalent operator or operator group. The solution we develop here is based on the well-known Cascades optimiser framework [Gra95] used in commercial products like Microsoft SQL Server [Del00]. To understand this approach, it is necessary to survey the architecture and its key components briefly.
Preliminaries

A *query plan* is an *n*-ary tree whose nodes correspond to operators of an extended relational algebra. The tree structure determines the order in which the operators are executed. As a consequence, each operator represents a sub-goal of the plan, that is, the partial query plan which is rooted in it.

During the optimisation process a *cost function* is used to estimate the costs of a (partial) query plan, usually an estimate for the relative or absolute time needed to execute this plan. The optimiser considers partial plans – i.e., generates alternative but semantically equivalent plans and, using the cost function, determines the costs of the alternative plans. If a new alternative is a candidate for an optimal sub-plan it is stored in a lookup table for further reference.

The MEMO Structure

The core data structure of the optimiser is the above mentioned lookup table known as *MEMO Structure*. We will describe the framework in only as much depth as is necessary to understand the implications for the use of \textsc{Choose} and refer the interested reader to [GM93, Gra95] and [WGL00]. The MEMO Structure holds all partial plans which are considered during the optimisation process. Since every operator corresponds to the sub-goal of the plan rooted by it, we can identify an operator with a partial plan. The MEMO is organised in groups. Each group contains all operators, and thus implicitly the partial plans, which implement equivalent sub-goals, *i.e.*, the plans which carry the same logical properties.

Copy-In Phase

Initially, the SQL query is algebraised, which means that an initial canonical evaluation plan is produced. In general, the initial plan is sub-optimal. When setting up the optimiser, the initial plan is copied into the MEMO structure. Every operator of the plan is assigned to a new group. Note that for the sake of presentation simplicity, we omit the cases where a plan contains several common sub-expressions. The references to the children of an operator are substituted by references to the respective groups. Hence, an operator, once copied into the MEMO, no longer refers to individual operators as children but all alternative sub-plans represented by its children’s groups. The group which accommodates the root of the initial plan is distinguished and we will refer to it as the *root group*. In Figure 4.25, the copy-in of a plan involving two joins is shown. The table on the right indicates the MEMO where numbers used in conjunction with operators refer to groups, *e.g.*, Group 3 contains the join of Groups 1 and 2, which
means any operator of Groups 1 and, respectively, 2 (and the subtrees rooted in them) can be used in this join.

**Optimisation**

During the actual optimisation process, transformation rules are applied to the operators in the MEMO. A transformation may match a sub-plan of arbitrary size rooted in the operator. The outcome of an application can be (1) a single operator which goes to the same group as the original, or (2) a sub-plan which spans several groups. New groups are created whenever necessary. Great care has to be taken to ensure that the transformations span the entire search space, i.e., repeated application of available transformations ensures that all plans of the search space are considered. Figure 4.26 shows a snapshot of the MEMO after various join orders have been explored.

Each operator in the MEMO is assigned a cost value which not only reflects the individual operator's cost but comprises the costs of the least costly sub-plan rooted in this operator. This minimality of costs is guaranteed by looking up the lowest cost value for each group and using this value when computing the costs for the parent groups. The exploration is steered by certain strategies which try to apply the most promising transformations first but essentially enumerate the entire search space. By investigating the most promising alternatives first, upper bounds on the overall costs of the plan can be obtained and used to prune the search space.

**Copy-Out Phase**

The principle to record the least costly operator per group, which is used for determining the cost value of subsequent operators, guarantees that the operator with lowest costs assigned in the root group represents the best plan found so far. During the copy-out phase, we simply traverse the MEMO structure and start with the least costly operator of the root group following the references to its children groups. This way, we obtain the most cost effective plan and then return it.

**Query Plans with CHOOSE**

The CHOOSE operator abstracts the fundamental property of the MEMO to groups of operators, which means that roots of equivalent query trees belong to the same group. Consider an initial plan as shown in Figure 4.27. During the algebraisation of the plan, the compatibility of the individual branches of the CHOOSE is checked by the SQL language processor. All operators are copied into the MEMO as outlined above.
4.4 User-Defined Query Equivalences for SQL

![Diagram of a query plan with CHOOSCE and corresponding initial MEMO structure.]

**Figure 4.27:** Input plan containing CHOOSCE and corresponding initial MEMO structure.

<table>
<thead>
<tr>
<th>Group</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root</td>
<td>3 (\bowtie) 4, 4 (\bowtie) 3, 5 (\bowtie) 2, 2 (\bowtie) 5, 6 (\bowtie) 1, 1 (\bowtie) 6</td>
</tr>
<tr>
<td>6</td>
<td>4 (\bowtie) 2, 2 (\bowtie) 4</td>
</tr>
<tr>
<td>5</td>
<td>1 (\bowtie) 4, 4 (\bowtie) 1</td>
</tr>
<tr>
<td>4</td>
<td>Scan C</td>
</tr>
<tr>
<td>3</td>
<td>1 (\bowtie) 2, 2 (\bowtie) 1, Scan D</td>
</tr>
<tr>
<td>2</td>
<td>Scan B</td>
</tr>
<tr>
<td>1</td>
<td>Scan A</td>
</tr>
</tbody>
</table>

**Figure 4.28:** Populated MEMO for plan with CHOOSCE operator after optimisation.

Instead of copying the CHOOSCE operator into the MEMO, its children are assigned to the same group (Group 3 in Figure 4.27). The subsequent join – copied into the Root group which referenced the CHOOSCE in the initial plan – now references the groups of the children of CHOOSCE. Note that the initial MEMO is very similar to the one displayed in Figure 4.25.

After the copy-in phase, all further optimisation is unaffected by the usage of CHOOSCE operators, which have now disappeared. The alternative partial query plans originating from the CHOOSCE are indistinguishable from equivalent operators the optimiser generated by applying inference rules. Note that since the CHOOSCE operator gets eliminated before the actual optimisation begins, no extensions to the optimiser logic are necessary. To underline the seamlessness of integrating the CHOOSCE into this kind of algebraic framework, we also present the populated MEMO after different join orders have been explored, analogously to above. Figure 4.28 shows hardly any difference to the previous one except that there’s one additional operator in Group 3.

Moreover, our approach also integrates with the cost-based selection mechanism of the optimiser framework which guarantees to choose the optimal plan according to the cost function used.

### 4.4.4 A Case Study with XML

As indicated in the introduction to this section, we anticipate a number of scenarios where overall query performance can benefit greatly from user-defined equivalences. Besides the aforementioned scenario of managing spatial data, applications which query XML documents stored in a relational database management system can
draw benefit from the CHOOSE operator as will be shown in the sequel.

With the advent of XML and a whole service infrastructure based on it as a communication format, the need for effective ways of storing and querying XML have become evident as has been shown in the previous chapters and pointed out, for example, in [SSK'01] and other works we already mentioned. Using relational database technology as a foundation has obvious advantages: relational database technology has been developed over an extensive period of about thirty years addressing all critical issues in data management. Particularly query optimisation and execution matured with respect to ever increasing performance requirements. Hence, it seems rather natural that implementing an XML store by using and extending a relational database engine has been investigated in a series of research prototypes. It became clear that extending a relational database system with XML functionality is not a straightforward task: a number of pitfalls, mainly due to the mismatch between structured and semi-structured data, became apparent.

The mapping of semi-structured data to relational concepts is pivotal to the problem. Expressing schemas where large numbers of attributes are merely optional may lead to grotesque artifacts like very sparsely populated tables; the situation may be even exacerbated if an a priori unknown schema has to be accounted for. A multitude of mapping techniques addressing various derivatives of these problems has been developed over the past few years. Mapping techniques range from fully generic graph models, which can cope with any kind of XML document, to schemes which rely on schema information given as a DTD or even XSD and which try to exploit this extra information as best they can to derive a compact storage format. Besides that, rigid mappings which do not allow for any flexibility but rather describe a data transformation mechanism can be found in most commercial database systems [Del00]. Eventually, it turns out that no mapping is consistently superior to all alternative mappings with respect to query performance. The choice of the appropriate mapping is at first sight application specific: XML data which is actually relational data but published as XML would best be stored using the original underlying relational schema; for a general purpose XML store, on the other hand, only generic mappings can be deployed. However, most application domains are neither of exactly the one or the other kind – rather, they encompass both aspects and feature structured and unstructured parts even in the same physical document. Although different data layouts in terms of relational schemas are feasible in these situations, and the decision when to choose which mapping cannot be made in a straightforward manner.

Though the mappings reconcile structured and semi-structured data with respect to storing of data, in terms of query performance a new challenge is encountered. All mappings exhibit certain strengths and weaknesses, and thus result in suitable to unacceptable query performance even within one application domain using one input document.

As a consequence, the suitability of a mapping cannot be decided statically on a per-application or on a per-data instance; rather, a decision is to be made with respect to a particular query set, too. In many situations application developers find themselves in the situation where multiple mappings, employed simultaneously, could help overcome performance problems of individual mappings. In the following, we show how the CHOOSE operator can be used in this very situation to combine any number of mappings and let the query optimiser decide when to use which mapping.
4.4 User-Defined Query Equivalences for SQL

...
Application Scenario

The framework within which we evaluate the applicability of CHOOSE is modelled after a typical e-business application: an on-line product catalogue. For clarity's sake, we use a relatively simple version of a real document: the central entity, the *item*, has a small number of attributes associated with it which account for the structured part of the document; the accompanying *Manual* with its textual components is the unstructured part. In actual application scenarios the number of elements related to an item might be much larger; however, no essential aspects relevant to our purposes are added by the additional elements.

Figure 4.30 shows a DTD which describes our document. The *item*’s sub-elements are singletons though not all mandatory, *i.e.*, there is a 1:1 relationship between them and the item; date and price may not be present in all items. The Manual consists of section, subsection, and paragraph elements which have 1:n relationships with their ancestors. Figure 4.29 shows a snippet of the document.

In our experiments, we studied two well-known approaches for mapping an XML document to relational tables and compared them to a combination of both that makes use of the CHOOSE operator. We only describe the techniques at a level of detail which is essential to understand our experiments. For a detailed discussion of the methods see, *e.g.*, [ZND+01], or consult the overview given in the previous chapter.

The first mapping is an adaptation of [STH+99] and relies on schema information about the document in the form of a DTD. The idea behind this technique, which we refer to as *inlining* in the following, is to identify structured components in the schema which are then mapped to tables in the database. To that end, we construct a graph whose nodes correspond to the elements specified in the DTD. Edges denote the sub-element relationships and are annotated with the type of the relationship (1:1 or 1:n). We fragment the graph along the 1:n relationships and map each fragment, which then only consists of 1:1 relationships to a table. In our example, all attributes other than Manual and its sub-elements can be inlined into an *item* table. Elements with 1:n relationships to their parents are assigned to separate tables. The links to their ancestors are established by referential integrity constraints on the id columns. Ignoring updates for the time being, the id column can be used as order column by encoding the rank into the OID as described in the previous chapter. Since all sub-elements of Manual may have arbitrary incidence, Sections, Subsections, and Paragraphs will be assigned separate tables. Note that it is possible to inline 1:n relationships also if the maximum value for n is known. However, inlining them is only useful if n is always close to its maximum in order to avoid sparsely populated columns. The resulting schema for this mapping is:

```
Items  : Itemld, Id, Title, Date, Price
Manual : Manualld, ItemId
Section : Sectionld, Manualld, Text
Subsection : Subsectionld, Sectionld, Text
Paragraph : Paragraphld, Subsectionld, Text
```

The second mapping is known as the *node table approach*. All elements are uniformly mapped to one single table with the following schema:

```
NodeTable : Startld, Endld, Parentld, Type, Value
```

*Startld* and *Endld* provide an id and encode the extent of an element, *i.e.*, determine the positions of start and end tags in the source document. *Parentld* encodes the
tree structure of the document while Type determines whether a tuple represents an XML element, an XML attribute, or character data. The values of elements, attributes, and character data is stored in the Value column.

Queries

Two typical and straightforward queries users might want to run against the above document are queries to access the attributes or to view the manual of a given item. In XQuery, these queries can be written as:

Query 1.

FOR $item IN document("catalog.xml")/item
RETURN
  <iteminfo>
    <date> $item/date/text() </date>,
    <title> $item/title/text() </title>,
    <price> $item/price/text() </price>
  </iteminfo>

Query 1 extracts the data-centric part of item records and assembles date, title into price. Query 2 simply recalls the entire manual of an item with id '712207'.

Query 2.

FOR $item IN document("catalog.xml")//item
WHERE $item/id/text()="712207"
RETURN
  <itemmanual>
    <manual> $item/manual </manual>
  </itemmanual>

Queries of both these types, besides being already quite useful as stand-alone queries, have a certain building block character since they describe the evaluation of path expressions [CFMR01b]. As a third query, we combined the previous two to simulate scenarios where assembling of both shallow and deep structures is necessary. In order to keep the example simple we just used the two previous queries and concatenated them using different document instances.

4.4.5 Query Performance

We translated the above XQuery expressions to SQL. For Query 1 the SQL counterpart looks like this:

```
SELECT XMLAGG(*)
FROM items
```

when using the inlining schema, and like this

```
SELECT XMLAGG(*)
FROM nodetable item, nodetable date,
    nodetable title, nodetable price
```
WHERE item.type = 'item' AND date.type = 'date'
AND title.type = 'title' AND price.type = 'price'
AND date.pid = item.id
AND title.pid = item.id
AND price.pid = item.id
ORDER BY item.id

for the node table approach.

The aggregate function XMLAGG(*) converts the output of the query to internal XML, a representation the query processor can more efficiently handle than plain, marked-up text. In commercial products, this aggregate usually requires a somewhat more elaborate notation. For brevity, we go with the above short-hand, as it does not add to our discussion. Moreover, the aggregate needs to be adapted for each of the mappings.

We reformulated Query 2 as

```
SELECT XMLAGG(*)
FROM items, manual m, section s, subsection sb, paragraph p
WHERE items.id = m.itemid
AND m.id = s.manualid
AND s.id = sb.sectionid
AND sb.id = p.subsectionid
AND items.id = 712207
ORDER BY items.id, m.id, s.id, sb.id, p.id
```

for the inlining mapping. For the node table, Query 2 looks like this:

```
SELECT XMLAGG(*)
FROM nodetable item, nodetable t
WHERE item.type = 'item' AND item.id = 712207
AND t.id > item.id AND t.id < item.id
AND t.type IN
    ('manual','section','subsection','paragraph')
ORDER BY t.id
```

For both queries the SQL text for the CHOOSE combination of both mappings is

```
<SQL for inlining>  CHOOSE  <SQL for node table>
```

We synthetically generated an XML document using a modified version of the software developed in the XML Benchmark Project [SKF*00], which is also introduced in the following section. On average, a manual contained 10 sections, 10 subsections, and 10 paragraphs. Each item is slightly larger than 1MB. In total, the database contained 500 items making for a total amount of data of 600MB. For both mappings we created indexes as appropriate.

Figure 4.31 shows a comparison of the query times for all three approaches. All times are given relative to the fastest of the three.

**Query 1.** As the inlining model performs only a simple table scan it easily trounces the node table approach which requires three joins to execute the query. With a larger number of attributes the performance gap would increase. Combined with the CHOOSE, the inlining is identified as the more cost effective. The overhead in optimisation when comparing it to inlining is negligible.
4.4 User-Defined Query Equivalences for SQL

100

10

1

Query 1 Query 2 Query 3

Figure 4.31: Query performance of individual mappings and mappings combined with CHOOSE

Query 2. With this query, we find the trend of the previous one reversed: inlining takes about forty times longer than in the node table approach. Here, inlining requires four joins to assemble the whole manual; in the case of a node table, the query boils down to a simple range query which benefits from an index. Again, using CHOOSE chooses the cheaper of the two, the node table in this case.

Query 3. This query underlines the advantages of CHOOSE when used deep in a query – as opposed to the previous queries, here, CHOOSE does not combine complete queries but sub-queries of a larger query. As a result, the optimised CHOOSE plan contains both an inlining and a node table part, combining the best performance of both approaches. We see a difference between CHOOSE and the runner-up of almost a factor 5.

Given the building block character of queries, one can expect the many non-trivial XML queries to be of the type of Query 3 in practise. It should be noted that the case study is not meant to promote some way of storing XML in relational databases. XML processing is only used as an illustrative and relevant example to motivate the advantages of user-defined equivalences with CHOOSE in advanced query processing.

4.4.6 Further Application Scenarios

As we have seen, the CHOOSE operator is very powerful in semi-structured or XML databases. In effect, the CHOOSE operator enables a new type of data independence, different from both physical and logical data independence. It permits multiple, possibly redundant representations of information, and selects the most promising one. This is different from physical data independence as the alternative representations are not indexes but tables in their own right, and it is different from logical data independence as it does not express a change in format or perspective but expresses a choice among alternative execution plans.

In this section, we consider some other application domains in which the CHOOSE operator adds considerable expressive power, optimisation opportunities, and execu-
tion performance. We will consider domains of growing importance for relational databases: on-line analytical processing (OLAP), temporal databases, spatial databases and caching.

**OLAP**

Many OLAP tools rely on relational database systems as backends for data storage to achieve availability, scalability, etc. These tools achieve interactive performance by storing not only detail data but also aggregations, often multiple aggregation levels in multiple dimensions, typically each level in a different table. When processing an OLAP request, the tool generates an appropriate SQL query against one of the tables, usually the smallest one by record count or byte count. However, query performance of relational systems depends more on appropriate indexes than on the size of the stored source data. Unfortunately, OLAP products either must re-implement the query rewrite capabilities and execution cost functions of each database system they support and, even worse, multiple releases of each product or they must rely on extremely simplified assumptions, e.g., that access cost is proportional to size. They usually do the latter, because the former is not commercially viable.

Some cases in which the CHOOSE operator would naturally be useful have already been implemented natively, with a fair amount of effort, in modern database query processors. Those include straightforward aggregation and inner join expressions, supported as materialised or indexed views by database products from all large database vendors. However, only a limited set of views can be materialised or indexed: outer joins, set operations, nested queries, transitive closure, roll-up and cube, as well as recent language additions such as rank and top are not supported yet. Nonetheless, pre-computing or caching will be useful for all these types of queries, and application designers or tools do so in many cases. In those cases, the CHOOSE operator is a natural solution and a perfect fit, because it results in query performance equal to that of a materialised or indexed view.

It can be argued that if equivalence truly exists, the alternative data sources indeed are materialised views of each other or of a third, more detailed data source. This may be true; however, implementing materialised view functionality for all thinkable view definitions will take a long time. Moreover, view materialisation functionality in database management systems will always rely on precise equivalence, whereas in some data analysis situations and statistical studies, approximate equivalence is sufficient and can be asserted only by the data or application architect or even the end user. Thus, we believe a syntactic mechanism such as the CHOOSE operator in a view definition or query will always be required as a complement to materialised view functionality implemented in database management systems today or in the foreseeable future.

OLAP tools often use standard naming conventions for tables representing different aggregation levels. In that case, it is possible to formulate CHOOSE queries that reference tables that currently do not exist. The database query processor could be extended to process any CHOOSE query that includes at least one branch that binds correctly to existing tables, columns, and data, and executes the least expensive among those. Of course, in general, the idea of running requests that reference undefined objects, and even presuming something about their contents and semantics should these objects ever be defined in the future, is rather frightening. Thus, we are not proposing this extension other than for future research, e.g., into securing or locking part of the namespace such that only the OLAP tool could ever create such tables.
Spatial Queries

Many common implementation techniques for spatial queries on relational database systems use bounding boxes at multiple resolutions. When determining which geographic features, for example, overlap with an area of interest, say a river basin, preprocessing steps based on bounding rectangles are used. These preprocessing steps can readily be expressed in SQL and executed efficiently using standard indexing techniques. After this pre-selection, an accurate and often computationally expensive predicate determines the final query result. The pre-selection is usually expressed in SQL as one or more joins with one or more tables containing bounding boxes.

If, however, there are other predicates in the query, and in particular if these other predicates are very restrictive yet relatively inexpensive to evaluate, the entire mechanism of joining with the table of bounding boxes may well be ineffective yet quite expensive. However, a query optimizer cannot automatically drop those joins, and will include them in the execution plan. With the CHOOSE operator queries can be written so that additional predicates become optional and are subject to the optimizer's cost-based selection mechanism.

Distributed Queries, Replica, and Caches

Nowadays, when specialised appliance servers are networked to provide Web services, data are often cached on demand at various locations and database servers with distributed query processing capabilities are often used as local caches. When queries are processed in such caches, the required data collection or table may or may not be present. Today, referenced data must be present or a query cannot be compiled and executed. We proposed to employ the CHOOSE operator in a further variant: Let it be sufficient that any one of the branches can be compiled, and let the query optimiser choose among those branches. For example, the two branches of a CHOOSE operator might refer to a local copy and a remote copy of the same data. If both copies exist and are accessible, the local copy is likely to be less expensive and will therefore be chosen. If the local copy does not exist and only the remote copy can be queried, the optimiser will correctly process the query. If a local copy exists but the remote server is temporarily disconnected, the optimiser will also process the query correctly.

If the query processor is able to estimate costs accurately even for distributed queries, it is also possible to choose among multiple alternative remote sources. Source data might be unavailable due to many reasons, e.g., temporary deletion as in the case of caches, off-line database maintenance, network issues such as router failure or deliberate disconnect during a security breach, planned or unplanned server downtime, etc. Again, by formulating a query or a view using the CHOOSE operator, the application or data architect can specify query equivalence.

Without further extensions, the optimiser can process only queries when fewer than all copies are available, but not updates. In general, updates of views with a CHOOSE operator can be processed correctly by updating all branches with the same delta, i.e., the same insertion and deletion sets. If one of the branches cannot be updated, e.g., because it is too complex due to a many-to-many join, the update request of the entire CHOOSE expression must be rejected.

Limitations of the CHOOSE Operator

Before we conclude this chapter, we have a look at the limitations that the use of the CHOOSE operator has. Semantic query optimisation techniques to which the operator
belongs, often violate the principle of physical data independence, i.e., they tend to exploit information that is not expressible in the framework of the relational data model. Hence, special care has to be taken when data are migrated to a different schema or queries rewritten; it is very easy in these cases to overlook and disregard dependencies, which eventually leads to wrong query results. However, remember that most SQL queries are not written by humans but rather generated by Graphics User Interfaces, Computer-Aided Software Engineering (CASE) tools or middleware applications – in short, in a closed world. In these scenarios, it should be possible to integrate dependency evolution into the design process and hence to control the dependencies as systems develop.

Quite obviously, views employing a CHOOSE operator can be updated, presuming they do not contain any of the other traditional constructs that prevent updates. An update simply updates all branches of the view, very similar to optimised multi-index update plans that update one index at a time and to execution plans for cascading referential constraints that propagate updates to one table at a time. Perhaps more interestingly, the semantics of CHOOSE operators can be extended to permit branches that cannot be executed at all, e.g., because referenced tables do not exist. It is not clear, however, how these two extensions can be combined in a safe way. Thus, the use of CHOOSE can also affect the transactional behaviour of a DBMS.

4.5 Conclusion

This chapter covered four different topics in XML querying: we started off by looking at how we can bulkload source documents efficiently into a DBMS and presented a way of querying them with the relational algebra combined with a preprocessing step. Whereas a query algebra requires domain knowledge for query formulation, the meet operator allows even novice users without domain knowledge to work with document databases. Eventually the CHOOSE operator was presented as a tool for advanced query optimisation by exploiting domain knowledge.

In particular, the meet operator lets users query XML databases with whose content they are familiar with but whose schema or structure they are unaware of. We have shown that it neatly fits current XML data models and that query languages can be easily extended to incorporate the additional functionality. Furthermore, we demonstrated that the algorithms yield useful results on real world data and also scale well, thus enabling interactive ad hoc querying. The novelty of our work is that the result type of the query is not specified by the user but dependent on the database instance queried. Therefore we referred to meet queries as nearest concept queries.

Promising future research topics in this area include further investigations into expanding the applications of the meet operator with respect to information retrieval techniques; some aspects are already present in this paper like ranking and restrictions. A natural extension of the work presented here would also be to look at how to incorporate views and IDREFs, which may break the tree structure of the database, into the search process. A case study in the bibliography domain demonstrate the usefulness of the meet with a practical example.

With the CHOOSE operator, we introduced a small syntactic extension to the SQL language. The proposed syntax for the CHOOSE operation is based on relational algebra table expressions, similar to other SQL extensions such as outer joins that don't fit as predicates in the WHERE clause. Syntactically, it is very similar to a UNION operator as it, too, combines two union compatible table expressions, i.e., tables with compatible
column sets. Despite its syntactic simplicity, the CHOOSE operator gives tremendous power to the application designer, tool developer or middleware application. In some cases, equivalent effects can be achieved using materialised or indexed views, which are often already implemented in the database system. In other cases, equivalent effects could be achieved if the implemented support for materialised views were more powerful. In many other cases, however, the equivalences known to the application designer or data architect cannot be expressed in SQL, meaning that they cannot be supported by any materialised views. This is the real advantage of our proposal: giving application designers the power to declare what information is equivalent. The power of the proposed extension was illustrated in several application domains, including online analytical processing, spatial databases, and databases for semi-structured data or XML. Experiments with some typical representations of XML data demonstrated the advantage that can be achieved by employing the CHOOSE operator to extend the set of possible query execution plans. In spite of the power and the breadth of application domains, it is truly straightforward to implement the CHOOSE operator in modern, extensible query optimisers. It is a core property of closed algebras that operations can be assembled into arbitrarily complex expressions, because constants and operator results have fundamentally the same type – relations our case. Therefore, algebraic query processing technology is inherently extensible. The described implementation of the CHOOSE extension is but one proof of that. There are a number of extensions beyond the CHOOSE operator hinted at here.

Parts of this chapter were previously published or are under review. The bulkloading techniques were first discussed in [SK02], and the meet operator in [SKW01]. A paper describing the CHOOSE operator is under review [WGLGS02]; parts of this work were done while the author was visiting Microsoft Corporation.

4.6 Bibliographical Remarks

There have been a number of works on evaluating queries on streams of XML documents whose results carry over to bulkloading documents since the latter can be seen as a special kind of query; see [AF00] and [ILW00] for details on this idea. The author is not aware of any other work describing intricacies and optimisation of bulkloading structured documents to date.

The literature on querying XML can be divided into two groups: querying XML views derived from conventional data models such as relational, object-relational or object-based models, and native support for XML query languages. [SSB+00] show how to treat and query relational sources as XML documents; [Sch01] present an interesting native XML mapping based on text arrays to query XML documents.

With respect to the meet operator, there have been a number of attempts to make querying XML documents or semi-structured data easier for users. In [FKM00] the authors enrich XML-QL with keyword search on subtrees of certain tags. The DBMS Lore [MAG+97a] also supports keyword and distance search. The difference to our work is that the result types have to be made explicit in the queries, which is what the meet operator avoids and hence allows simpler query formulation. Furthermore, by restricting the result types, the operator can be used to implement keyword search as a special case. In [GSVGM98] the authors present algorithms for proximity search in graphs; their queries follow a ‘Find objects from $S_1$ Near objects from $S_2$’ pattern where the user has to specify sets $S_1$ and $S_2$; therefore formulating these queries also requires more domain-knowledge than is needed for meet queries. Another view on
our work is that we are trying to exploit the inherent semantics encoded in the tag hier-
archies; a very interesting approach to combining knowledge from outside the database
with internal knowledge is [GW00]. However, this approach is of a different nature and
as such complementary to ours. The algorithmic problem of finding lowest common
ancestors although novel to XML processing as a query primitive has a long history in
databases and code optimisation, see [ASSU81, AHU76]. We also assume that es-
specially relational XML Query processors that support XQL’s before and after predicates
already provide some of the functionality a full implementation of the meet operator
requires.

The CHOOSE operator is only one member in the large family of query optimisation
strategies. Earlier work on DBMS extensibility considered mechanisms to increment-
tally add logic to query optimisers, to deal with new logical operators, implementa-
tion algorithms, and equivalence rules; some important works include[Fre87, GD87,
Loh88] and [HFLP89, GM93, PHH92]; for a very readable description see [Gra95]. In
these works, a typical DBMS extension would include adding support for a new kind
of object, say time series, requiring changes spanning all the way from language sup-
port to potentially special-purpose storage structures. A database implementor would
use extensibility features to prepare a custom-tailored DBMS. In the query optimiser
in particular, equivalences would then be defined through a collection of rules that in-
dicate how to generate alternatives, using data structures provided by the extensibility
framework. The code from those rules gets compiled and linked into the optimiser
module of the DBMS. The work presented here is complementary to the techniques
developed for extensible DBMSs. We assume that the application can map its data to
objects already supported by the DBMS, such as tables, and focus exclusively on ex-
tending the generation of alternatives of the query optimiser. The relevant information
about equivalences is provided per-query, using SQL directly. This allows factoring
out the DBMS the logic to understand application-specific equivalences, while main-
taining cost-based selection of efficient plans in the extended search space. As with
other extensibility features, this enables application writers to implement functionality
on their own, without having to wait for it to be implemented natively in the DBMS. An
area for improvement in DBMS development is constraint definition and its utilisation
by the query processor. Materialised views already are a powerful form of constraint
declaration: the DBMS is supposed to maintain the constraint rather than to enforce
it. In some cases, a more expressive constraint system could eliminate the need for the
CHOOSE operator. However, it is questionable whether it is desirable or feasible to
support very complex application-specific constraints natively.

Another body of work that is both orthogonal and complementary to the CHOOSE
operator is dynamic optimization – improving performance by delaying some decisions
on the execution plan until query runtime, to take into account information that may
not be available at compilation time, such as actual values of query parameters, avail-
able memory, and state of the execution environment in general [CG94, INSS92]. The
CHOOSE operator is only about declaring equivalences, while the process of deciding
which alternative to execute and how remains entirely the taste of the query processor.
The alternatives opened up by CHOOSE need not be resolved statically at query com-
pile time. Specifically, the run-time choose operator proposed in [CG94] can be used to
select between the alternatives, when there is not enough information at compile time
as to which is preferable.