TRIGGERS AND RANKED KEYWORD SEARCHES
OVER VIRTUAL XML VIEWS

A Dissertation
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
Feng Shao
August 2007
TRIGGERS AND RANKED KEYWORD SEARCHES OVER VIRTUAL XML VIEWS

Feng Shao, Ph.D.

Cornell University 2007

Current systems that publish XML/relational data using XML views are passive in the sense that they can only respond to user-initiated queries over the XML views. Further, existing systems do not support ranked keyword searches over virtual XML views, which is important for exploring and retrieving information from large views. In this dissertation, we propose an XML view system whereby users can place active triggers on virtual (unmaterialized) XML views, and can efficiently evaluate keyword search queries over such views. In this architecture, we present scalable and efficient techniques for processing triggers over nested views by leveraging existing support for SQL triggers over flat relations in commercial relational databases. When evaluating the keyword search queries, our approach exploits indices present on the base data and thereby avoids computing large parts of the view that are not relevant to the query results. Another feature of the algorithm is that it supports top-k results for queries over the virtual view, and the resulting rank order is the same as if the view was materialized. We have implemented our proposed techniques in the context of the Quark XML middleware system. Our performance results indicate that our proposed techniques are a feasible approach to supporting triggers and ranked keyword searches over virtual XML views.
BIOGRAPHICAL SKETCH

Feng Shao was born January 3, 1978 in China. He was raised by his loving parents, Hengda Shao and Jinying Lu, in their hometown of Huangshan, Anhui province. Feng Shao graduated from University of Science and Technology of China in 2001 with a B.S. in Computer Science. Since July 2001, he has been a PhD student at Cornell University in the department of Computer Science.
ACKNOWLEDGEMENTS

The completion of this dissertation would not have been possible without the support and assistance of some valuable people to whom I wish to extend recognition and thanks. Foremost I would like to thank my academic advisor, Jayavel Shanmugasundaram, for his guidance and encouragement. This work could not have been done without his advice on both database research and the development of large-scale software systems.

I also want to thank the other members of my Special Committee, Professor Jon Kleinberg and Professor Levent Orman, for their valuable advice on research in general and on specific problems related to this dissertation.

The work presented in this dissertation is implemented and evaluated in the context of the Quark XML database system. While there have been many students contributing to Quark, I would like to especially thank Anand Bhaskar, Antal Novak, Chavdar Botev, Lin Guo, Fan Yang for being such wonderful collaborators.

I am fortunate and proud to be part of the database group at Cornell. Much of my understanding of the database is due to my interactions and discussions with faculty and students in the group. I also want to thank Sihem Amer-Yahia at Yahoo! Research for her help on the keyword search problem.

I have met many friends during the five year life at Cornell. Many thanks to Jun Chen, Mingsheng Hong, Hongzhou Liu, Zhuyin Ren, and Yong Yao; my every-day life would not have been as enjoyable as it has been without them.

Finally I would like to extend my warmest thank to my family. They helped me to overcome the moments of doubts with endless faith, love and support in many ways. I am also grateful to my expected son who gave the final push to complete this work. :)
# TABLE OF CONTENTS

1 Introduction  
1.1 Motivation .................................................. 1  
1.2 Triggers over virtual XML views .............................. 3  
1.3 Ranked keyword search queries over virtual XML views 4  
1.4 Organization of chapters ................................... 6  

2 Background  
2.1 XML Documents and Queries ................................ 7  
2.2 Nested Relational Algebra and XML ......................... 9  

3 Triggers over Nested XML Views  
3.1 Background ................................................. 14  
3.1.1 Trigger specification language over nested data .......... 18  
3.1.2 SQL triggers ........................................... 19  
3.2 Semantics & System Architecture .......................... 20  
3.2.1 Semantics of triggers on nested views .................. 20  
3.2.2 System architecture .................................... 26  
3.2.3 Trigger parsing and event pushdown ..................... 27  
3.3 Affected-Row Graph Generation ............................ 30  
3.3.1 Technical challenges in producing $G_{\text{affected}}$ ........ 31  
3.3.2 Proposed algorithm - **CreateAKGraph & CreateARGraph** 33  
3.3.3 Additional examples of the algorithms ................ 39  
3.3.4 **CreateAROpt**: optimizations for CreateARGraph .... 44  
3.3.5 Adding Condition and Action ........................... 52  
3.4 Trigger Grouping and Pushdown ............................. 52  
3.4.1 Trigger grouping ....................................... 52  
3.4.2 Trigger pushdown optimizations ........................ 54  
3.5 Experimental Evaluation .................................... 56  
3.5.1 Experimental setup ..................................... 58  
3.5.2 Performance results .................................... 61  
3.6 Correctness of **CreateAKGraph** .......................... 66  
3.7 Correctness of **CreateAROpt** ............................. 79  

4 Ranked Keyword Search over Virtual XML Views  
4.1 Background .................................................. 94  
4.1.1 Background on XML storage & indexing .................. 99  
4.1.2 Scoring .................................................. 102  
4.1.3 Problem definition ...................................... 103  
4.2 System Overview ............................................ 104  
4.2.1 System architecture .................................... 104  
4.3 QPT Generation Module ...................................... 107
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Deriving canonical keys for operators.</td>
<td>22</td>
</tr>
<tr>
<td>3.2</td>
<td>Operator-specific rules used in event pushdown.</td>
<td>29</td>
</tr>
<tr>
<td>3.3</td>
<td>Experimental parameters.</td>
<td>58</td>
</tr>
<tr>
<td>3.4</td>
<td>Evaluated approaches.</td>
<td>59</td>
</tr>
<tr>
<td>4.1</td>
<td>Experimental parameters.</td>
<td>127</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

1.1 XML views of relational/semi-structured data. 2
2.1 An XML document. 8
2.2 Example of nested relations. 9
2.3 Example database. 10
2.4 Nested relational algebra operators. 10
2.5 The operator graph for the catalog view. 11
2.6 Nested catalog view. 12
2.7 XML catalog view. 12
3.1 XML View System Architecture. 15
3.2 System architecture. 25
3.3 GetSrcEvents: given an operator, o, and a desired event on that operator, e, returns the set of table-level events which can cause e. 28
3.4 G_params: Producing parameters to Action. 30
3.5 Algorithm for producing affected keys. 35
3.6 Algorithm for producing affected keys: SetDiff and Union. 36
3.7 CreateAKGraph: Step 1. 37
3.8 CreateAKGraph: Nest operator. 38
3.9 The complete $G_{par}$ graph. 38
3.10 An additional example for createAKGraph. 40
3.11 Parallel graph produced by createAKGraph. 40
3.12 Simplified graph produced by CreateAKGraph. 41
3.13 CreateAKGraph on Union. 42
3.14 Union augmented with $index$ columns. 42
3.15 Algorithm for producing $G_{affected}$. 44
3.16 The final $G_{params}$ graph. 45
3.17 Converting select to join. 52
3.18 Correlated $G_{grouped}$ graph. 52
3.19 The generated SQL trigger. 57
3.20 Varying the number of triggers. 59
3.21 Varying hierarchy depth. 60
3.22 Varying # of fired triggers. 63
3.23 Varying # updated elements. 63
3.24 Varying # leaf tuples/element. 64
3.25 Varying the data size. 64
3.26 The DBLP structure. 65
3.27 Varying number of triggers for DBLP. 66
3.28 Illustration of valid transition tables. 67
4.1 An XML view associating books & reviews. 95
4.2 An XML document with Dewey Ids. 99
4.3 XML path indices. 100
Chapter 1

Introduction

1.1 Motivation

One of the central concepts in databases is the concept of views. In the relational model, a view is a derived table defined in terms of base (stored) relations. A view thus defines a function from a set of base tables to a derived table; this function is typically recomputed every time the view is referenced. A view can be materialized by storing the tuples of the view in the database to provide fast access. The view mechanism has many advantages including logical data independence, access control and data restructuring.

Recently as XML emerged as a dominant standard for information exchange on the Internet, there has been a lot of interest in XML views. Similar to relational views, an XML view is a derived XML fragment defined in terms of an XML query on XML documents. XML views are particularly interesting in the environment of the Internet. On one hand, XML has become the de-facto standard for information exchange over the Internet, e.g., XML is adopted in many web service standards [112]. On the other hand, a large fraction of data continued to be stored in relational databases, and even when the base data are stored as XML documents, users may wish to create more intuitive application-specific views. As a result, there has been a lot of interest in XML view-related problems including incremental maintenance of materialized views, publishing relational data using XML views, and querying XML views of relational data, and numerous algorithms and techniques have been proposed [27, 52, 57, 61, 65, 80, 82, 100, 101, 106, 116].

Emerging Internet applications, however, pose new requirements on XML views. We focus on two such new requirements in this dissertation. First, current systems that
Figure 1.1: XML views of relational/semi-structured data.

Support XML views are passive in the sense that they only support user-initiated queries over views; hence, users cannot use triggers to effectively monitor views for interesting changes. Second, current systems do not support exploratory queries over XML views. For example, users cannot issue ranked keyword search queries over the views. The goal of my research is to build an XML view architecture that addresses these limitations. The proposed system architecture is shown in Figure 1.1. In the next two sections, I will describe each of the new features in this architecture in detail.
1.2 Triggers over virtual XML views

As mentioned earlier, current systems that support XML views of relational data [21, 52, 77, 100, 106, 116] are passive in the sense that they can only support user-initiated queries over the views. For example, consider a catalog RDBMS consisting of a product table and a vendor table, as shown in Figure 1.1. In this database, users can define an application-specific catalog XML view created by nesting all vendors for a product under the corresponding product element with the restriction that only products sold by at least two vendors appear in the view.

Now consider a supplier that exposes its product catalog information as a web service using our example catalog view. Current systems only allow buyers to explicitly initiate a request to query the catalog for products of interest. In contrast, an active system allows users to specify triggers over XML views. Thus, a buyer can set a trigger to be notified whenever a new product is introduced, or when a product of interest goes out of stock, without having to repeatedly query the nested view to detect these changes.

At a high level, there are two approaches to supporting triggers over nested XML views. The first approach is to materialize the entire XML view and implement triggers over this view. However, this approach suffers from the overhead of replicating and incrementally maintaining the materialized XML on every relational update affecting the view, even though users may only be interested in relatively rare events.

In this dissertation, we propose an alternative approach of implementing triggers over nested XML views of relational data, which is by translating triggers over nested views into SQL triggers over flat base tables. The primary benefits of this approach are that: (a) it avoids having to materialize the nested view, (b) it does not require a sophisticated data management system for nested data, and (c) it works with and fully leverages existing relational technology. We have experimentally evaluated the proposed
techniques in the context of Quark XML database system, and the performance results indicate that the proposed architecture is a feasible approach to support triggers over virtual XML views.

1.3 Ranked keyword search queries over virtual XML views

The second proposed feature in Figure 1.1 is to support exploratory keyword search queries over virtual XML views. The problem of ranked keyword search queries has been studied in the context of traditional information retrieval systems [12, 103]. However, the existing techniques rely heavily on a fundamental assumption that the set of documents being searched is materialized, while there is a rich class of semi-structured search applications for which it is undesirable or impractical to materialize documents.

For example, consider a large online web retailer that uses our example view to allow users to view the products. Since different users may have different interests, the retailer may wish to provide a personalized view of the content to its users (such as products tailored for different ages, languages, and budgets), and allow users to search such views. In such cases, it may not be feasible to materialize all user views because there are many users and their content is often overlapping, which could lead to data duplication and its associated space-overhead. In contrast, a more scalable strategy is to define virtual views for different users of the system, and allow users to search over their virtual views.

Further, in our example, the product data and the vendor data may belong to independent web services, and the web retailer can only exploit an information integration application that queries over both services and creates the view that aggregates vendors under products. Note that such a view is often virtual (unmaterialized) for various reasons: (a) the aggregator may not have the resources to materialize all the data, (b) if
the view is materialized, the contents of the view may be out-of-date with respect to
the base data, or maintaining the view in the face of updates may be expensive, and/or
(c) the data sources may not wish to provide the entire data set to the aggregator, but
may only provide a sub-set of the data in response to a query. While current systems
(e.g., [24, 39, 52]) allow users to query virtual views using query languages such as
XQuery, they do not support ranked keyword search queries over such views.

The above applications raise an interesting challenge: how do we efficiently evaluate
keyword search queries over virtual XML views? One simple approach is to materialize
the entire view at query evaluation time and then evaluate the keyword search query over
the materialized view. However, this approach has obvious disadvantages. First, the cost
of materializing the entire view at runtime can be prohibitive, especially since only a few
documents in the view may contain the query keywords. Further, users issuing keyword
search queries are typically interested in only the top few results, and materializing the
entire view to produce the top few results is likely to be expensive.

In this dissertation, we propose an alternative approach, which is by using regular
indices, including inverted list and XML path indices, that are present on the base data
to efficiently evaluate keyword search over views. The indices are used to efficiently
identify the portion of the base data that is relevant to the current keyword search query
so that only the top ranked results of the view are actually materialized and presented to
the user. We have implemented and experimentally evaluated the proposed techniques
and the performance results indicate that our approach is at least ten time faster than
alternatives.
1.4 Organization of chapters

Chapter 2 provides the necessary background knowledge for triggers and keyword search queries over virtual XML views. Chapter 3 describes in details the proposed techniques for supporting triggers over virtual XML views. Chapter 4 is then devoted to the problem of efficiently evaluating ranked keyword search queries over XML views. Chapter 5 reviews the past research that is related to trigger processing and ranked keyword search queries. Chapter 6 summarizes the entire research and the findings of previous chapters, along with the suggestions for future directions of research.
Chapter 2

Background

In this chapter, we describe some background on XML documents and queries, and also describe background on nested relational algebra that is used to represent and manipulate nested XML views over relational data.

2.1 XML Documents and Queries

An XML document consists of nested XML elements starting with the root element. Each element can have attributes and values, in addition to nested subelements. Figure 2.1 shows an example XML document representing books with nested reviews. Each \texttt{〈book〉} element has \texttt{〈title〉} and \texttt{〈review〉} subelements nested under it. The \texttt{〈book〉} element also has the isbn attribute whose value is “111-11-1111”. For ease of exposition, we treat attributes as though they are sub-elements. While XML elements can also have references to other elements (IDREFs), they are treated and queried as values in XML; hence we do not model this relationship explicitly for the purposes of this paper. In addition, to capture the text content of elements, we use the predicate: \texttt{contains}(u, k) returns true iff the element \texttt{u} directly or indirectly contains the keyword \texttt{k} (note that \texttt{k} can appear in the tag name or text content of \texttt{u} or any of its descendant elements).

An XML document can thus be defined as a directed tree \( D = (V, E, r, Tag, Val) \), where \( V \) is the set of XML elements, \( E \) is the set of edges such that \((u, v) \in E\) iff \( v \) is a nested subelement of \( u \), \( r \) is the root element, \( Tag : V \rightarrow String \) is a function that maps each element to its tag name, and \( Val : V \rightarrow AtomicValue \) is a function that maps elements to their atomic values (if an element does not have an atomic value, the corresponding value is null). We say that an element \( w \) is a descendant of element \( u \) if
there is a path from $u$ to $w$ in $E$.

An XML database instance $D$ can be modeled as a set of XML documents. An XML query $Q$ can be viewed as a mapping from a database instance $D$ to a sequence of XML documents/elements (which represents the output of the query). More formally, if $UD$ is the universe of XML database instances and $S$ is the universe of sequences of XML documents/elements, then $Q : UD \rightarrow S$. Thus, we use the notation $Q(D)$ to denote the result of evaluating the query $Q$ over the database instance $D$. A query $Q$ is typically specified using an XML query language such as XQuery [1]. An XML view is simply represented as an XML Query. We thus use the term view and query interchangeable for the rest of the chapter. Further, we use the following notation for reasoning about sequences of elements. Given a sequence of elements $s$, $e \in s$ is true iff the element $e$ is present in the sequence $s$. Given two sequences $s_1$ and $s_2$, $\text{subseq}(s_1, s_2)$ is true iff $s_1$ is a sub-sequence of $s_2$ (i.e., $s_2$ contains all the elements of $s_1$ in the same relative order).
The nested relational algebra \([99, 110, 77]\) works on non-first normal form (NFNF) or nested relations, where the column value of a row can itself be a relation. In this paper, we consider a relation to be a *bag* of rows because we want to consider duplicate rows as in SQL. An example of a nested relation is shown in Figure 2.2(b). In this example, the nested relation contains three columns: A, B and E. The type of column B is itself a nested relation, which has two columns C and D. A, C, D and E are atomic columns (i.e., not nested relations). For instance, the first row in the relation has the value a1 for column A, the set \(\{(c1,d1), (c1,d2)\}\) for column B and the value e1 for column E.

The nested relational algebra works on nested relations and contains all the traditional relational operators (**Select** \(\sigma\), **Project** \(\pi\), **Join** \(\bowtie\) and **Union** \(\cup\), and **SetDiff** \(-\)) and also introduces two new operator – **Nest** \(\nu\) and **Unnest** \(\mu\) – for dealing with nested relations. Figure 2.4 summarizes the operators in the nested relational algebra.

The **Nest** operator takes an input relation and a list of columns to be nested (nesting columns), and creates an output relation that contains exactly one row for each distinct combination of non-nesting columns. The output relation also contains a new column (the nesting output) that contains a nested set of all combinations of nesting column val-
ues for a given combination of non-nesting column values. The \textbf{Nest} operator over an input relation $r$ with the list of nesting columns $NCL$ and the name of the nesting output NO is denoted by: $\nu_{NO=NCL}(r)$. For example, consider the relation $r$ in Figure 2.2(a).

The nested algebra expression $\nu_{B=(C,D)}(r)$ produces the nested relation $r'$ shown in Figure 2.2(b). While the traditional nested algebra does not support arbitrary aggregate functions, we can augment the \textbf{Nest} with the specification of an aggregate function over a nested relation. For example, the nested algebra expression $\nu_{B=(C,D),cnt=count(B)}(r)$ produces a relation similar to $r'$, but which has an additional attribute $cnt$ whose value for a given row is the cardinality of the nested relation $B$ for that row.

The \textbf{Unnest} is the converse of the \textbf{Nest} and flattens a nested relation based on an unnesting column. The unnesting column can only contain set values. The \textbf{Unnest} operator over an input relation $r$ with an unnesting column UC is denoted by: $\mu_{UC}(r)$. For example, the nested algebra expression $\mu_{B}(r')$ produces the relation $r$ (Figure Figure 2.2(a)). Note that an \textbf{Unnest} is the exact converse of a \textbf{Nest} only if the \textbf{Nest} does not contain aggregate functions (such as $count$). If the \textbf{Nest} does contain an aggregate
function, the **Unnest** will produce an extra column corresponding to the aggregate value, which will be replicated for each unnested row (along with the values of the other non-unnesting columns). If the aggregate column value is not desired in the output of the **Unnest** it has to be explicitly projected out before or after the **Unnest**.

As another example, consider a relational database shown in Figure 2.3 (the database contains products and vendors for each product; primary keys are capitalized). Now suppose this database is exposed as a (virtual) nested view in which vendors are nested under products, with the restriction that only products sold by *at least two* vendors appear in the view. The corresponding nested view definition is shown below:

\[
\begin{align*}
\nu_{\text{products}} &= (\text{pname}, \text{vendor}) \left( \pi_{\text{pname}, \text{vendor}} \left( \sigma_{\text{cnt} \geq 2} \left( \nu_{\text{vendor}} = (\text{vid}, \text{pid}, \text{price}) \left( \pi_{\text{pid}} \left( \text{product} \Join \nu_{\text{vendor}} \right) \right) \right) \right) \right) \\
\nu_{\text{vendor}} &= (\text{vid}, \text{pid}, \text{price}) \left( \text{cnt} = \text{count}(\text{vendors}) \left( \pi_{\text{pid}} \left( \text{product} \Join \nu_{\text{vendor}} \right) \right) \right)
\end{align*}
\]

The result of materializing this nested view is shown in Figure 2.6.

A nested relational algebra expression can also be viewed as a tree or graph of algebra operators, or an operator graph. Figure 2.5 shows the operator graph for the view
definition expression above. Operators (boxes) 1 and 2 produce the rows in the product and vendor tables, respectively. Box 3 joins each vendor with the product it sells, and Box 4 then nests the rows by product name; the nest() function nests all the vendor rows into a set, while the count function counts the number of vendors per row. Box 5 then selects only the rows with cnt ≥ 2. Finally box 6 nests all the rows together to produce the nested relation with a single products column. Note that the projection operators are not explicitly shown in the graph, but are implicitly captured by the columns propagated by the operators.

Note there is a fairly direct mapping between nested relations and XML. For instance, the nested relation in Figure 2.6 directly corresponds to the XML document in Figure 2.7, except that the XML view contains additional tags which can be easily added in a post-processing phase [107]. Similarly, as shown in previous work, there is a direct mapping between nested relational algebra views and XML views [44, 10].

There are, however, some differences between XML views and nested relational views. First, XML views often have descendant and wildcard axes, which are not present in nested relational views. However, this is not a big issue in the domain of relational publishing (the focus of this paper) because the relational schema is known a-priori and
can be used to expand the descendant and wildcard axes to child axes with specific tag names. Second, XML documents can contain a mix of structured and text data, while nested relational views are more structured. However, since we are focusing on publishing structured relational data, the resulting XML is unlikely to have a mix of structured and text data.
Chapter 3

Triggers over Nested XML Views

Triggers have been shown to be a powerful technology to implement push-based data processing in relational databases [40, 41], in the sense that the database server takes the active role of pushing data to users when events that match trigger specifications occur. Consequently, users do not have to repeatedly query the database for interesting changes (which is the case in the pull-based database systems).

Emerging Internet applications, such as XML view systems [21, 52, 100, 77, 106, 116], require triggers as well so that users are able to monitor events of their interests. However, as mentioned in the introduction, existing XML view systems are passive in the sense that they can only support user-initiated queries over the views. For instance, consider the XML view system architecture in Figure 3.1 in which the base relational data are published using nested XML views. Current systems using this architecture only allow buyers to explicitly initiate a request to query the catalog for products of interest. In contrast, an active system allows users to specify triggers over nested views. Thus, a buyer can set a trigger to be notified whenever a new product is introduced, or when a product of interest goes out of stock, without having to repeatedly query the nested view to detect these changes.

At a high level, there are two approaches to supporting triggers over nested views. The first approach is to materialize the entire nested view and implement triggers over this view. However, this approach suffers from the overhead of replicating and incrementally maintaining the materialized nested view on every relational update that affects the view, even though users may only be interested in relatively rare events. Another practical downside of this approach is that it requires a full-function database that sup-
ports incremental view updates and triggers over nested views, which is not commonly available. For instance, none of the XML (or XML-enhanced relational) databases that we are aware of support triggers over incrementally maintained nested views, while virtually all major commercial relational databases have sophisticated support for SQL triggers over (flat) relational tables [72, 87, 93].

To address the above issues, in this dissertation, we propose an alternative approach of implementing triggers over nested views of relational data, which is by translating triggers over nested views into SQL triggers over flat base tables. The primary benefits of this approach are that: (a) it avoids having to materialize the nested view, (b) it does not require a sophisticated data management system for nested data, and (c) it works with and fully leverages existing relational technology.

Figure 3.1: XML View System Architecture
The main technical contribution is a systematic way to translate triggers over nested views of relational data into SQL triggers over flat base tables. This translation is fairly challenging because triggers can be specified over complex nested views with nested predicates, while classical SQL triggers can only be specified over flat relational tables. Consequently, even identifying the parts of an nested view that could have changed due to a (possibly deeply nested) SQL update is a non-trivial task, as is the problem of computing the old and new values of an updated fragment of the view.

On the surface, this problem of identifying changes in a view may appear to be similar to the problem of incremental view maintenance. However, there are several subtle issues that require the development of new techniques. First, most incremental view maintenance techniques assume that the old (before update) values of data items are materialized, and focus on computing the new value based on the old value. In contrast, since our goal is to work over virtual views, we need to selectively compute the old (before update) and new (after update) values for a particular trigger based on an underlying update. Second, nested views give rise to an important and interesting class of predicates which we call nested predicates (essentially, these are predicates over nested structures, as will be explained in more detail later); existing view maintenance techniques designed primarily for flat relational tables do not apply to such predicates. Third, triggers over views need an efficient way to check whether the old values and new values, which may be deeply nested, are indeed different so that users do not receive spurious notifications (for example, if a view computes a function such as max, the old and new values may be the same even if the base data has been updated, so long as the maximum value does not change). Another issue to consider is that current commercial relational databases are not very scalable with respect to the number of SQL triggers, while we expect a large number of triggers to be specified over nested views exposed as
web services.

The proposed solution address the above challenges. Specifically, the three main contributions are: (1) a system architecture for supporting triggers over nested views of relational data (Section 3.2), (2) an algorithm for identifying and computing changes in an nested view based on possibly deeply nested relational updates (Section 3.3), and (3) the definition and use of a general class of views called injective views for which we can efficiently check whether the old values and new values are different, without having to explicitly compare these (possibly deeply nested) values (Section 3.3). It is also shown that how prior work on scalable trigger processing [34, 67] can be adapted for the nested view problem, and investigate the effects of using incrementally maintained relational materialized views for evaluating triggers over nested views (Section 3.4).

As mentioned in Section 2.2, there is a direct mapping between nested relational algebra views and XML views. Therefore in this chapter we use nested relational algebra to represent and manipulate nested XML views. One benefit is that the proposed techniques are naturally applicable to nested relational views.

The proposed techniques are implemented in the context of the Quark XML middleware system. One of the original goals of Quark (like SilkRoute [52] and XPERANTO [106]) was to support queries over nested XML views of relational data. By integrating with Quark, we were able to leverage many of the techniques originally developed for querying nested XML views, and adapt them to the trigger problem. This suggests that our techniques can be easily integrated into systems that already support queries over nested XML views of relational data (including relational databases with built-in XML publishing support). Our performance results using our prototype (Section 3.5) show that our proposed techniques provide an efficient and scalable way to support triggers over nested views of relational data, so long as the triggers are structurally similar.
gers that are not structurally similar, trigger grouping techniques are not effective and suffer from the same scalability problems as in publish/subscribe systems.

3.1 Background

In this section, we describe some background on trigger specification languages and SQL triggers – these concepts are used in our trigger processing system architecture.

3.1.1 Trigger specification language over nested data

Bonifati et al. [19] propose a trigger specification language for triggers over XML documents. We adapt this language for triggers over nested relational views by mapping XML path expressions to nested relational path expressions [114], i.e., XML path expressions of the form catalog/product/name gets mapped to nested relational path expressions of the form catalog.product.name, where catalog, product and name are names of columns in the nested relational model. Other than this syntactic modification, we retain the semantics of triggers proposed by Bonifati et al. [19].

The trigger specification language has the following syntax:

CREATE TRIGGER Name AFTER Event
ON Target WHERE Condition DO Action

A trigger has a unique Name. The Event specifies the operation that activates the trigger, and can be either UPDATE, INSERT, or DELETE. Target is an nested relational path expression that specifies the portion of the nested view to be monitored for the event. Condition is a Boolean expression which determines whether the trigger is to be fired for each row produced by evaluating the Target expression. If the condition is satisfied for a row, an Action is performed; in our system, the action is a call to an
An example trigger over the view in Figure 2.6 is shown below.

```sql
CREATE TRIGGER Notify AFTER Update
ON catalog.products
WHERE OLD_ROW.pname = 'CRT 15'
DO notifySmith(NEW_ROW.pname, NEW_ROW.vendor)
```

On any update to a product whose name was “CRT 15” (before the update), the trigger invokes an external function `notifySmith()` with the new values of attributes of that product row.

We note that the above semantics corresponds to row-level triggers in SQL trigger terminology – the trigger is fired for each row that is affected by the triggering statement.

### 3.1.2 SQL triggers

SQL triggers [40, 41] are fired when an event (INSERT, UPDATE, or DELETE) occurs on a specific relational table. When a SQL trigger is activated, it has access to the pre- and post-update versions of the affected rows through transition tables. We use the notation $\nabla table$ to denote the transition table that contains the updated rows before an update, and $\Delta table$ to denote the transition table that contains the updated rows after an update ($\nabla table$ is empty for INSERT triggers, and $\Delta table$ is empty for DELETE triggers). For example, if P1 goes on sale at Amazon, then the transition tables might look like:

<table>
<thead>
<tr>
<th>$\nabla vendor$</th>
<th>$\Delta vendor$</th>
</tr>
</thead>
<tbody>
<tr>
<td>vid</td>
<td>pid</td>
</tr>
<tr>
<td>Amazon</td>
<td>P1</td>
</tr>
</tbody>
</table>
3.2 Semantics & System Architecture

We now formalize the semantics of triggers on nested views, and then present our system architecture.

3.2.1 Semantics of triggers on nested views

In order to define the semantics of triggers on views, we need a precise definition of when a row in a view is said to be updated, inserted, or deleted. This in turn requires us to define the identity of a row in the view (so that we can talk about which specific row is updated, inserted or deleted). Note that the issue of identity is not as problematic for triggers over base relational data because each row has a well-defined notion of physical identity based on its record id (rid). In contrast, rows in unmaterialized views are virtual and do not have a standard notion of identity based on record ids.

We now present an intuitive definition of the identity based on the semantic structure of the view, i.e., in terms of the view’s algebra expression. The main idea is to use the notion of keys of nested relational operators to define the identity of rows.

**Definition 3.2.1 (Keys of Operators).** Given an operator $o$ in a graph $G$, a key of $o$ is a minimal set of (existing or derivable) columns of $o$ whose values uniquely identify each output row produced by $o$.

As an illustration, a key of the Table operator in box 1 in Figure 2.5 is the $spid$ column (which is the product table’s primary key). A key of the Nest operator in box 4 is the $spname$ column (since the Nest operator produces an output row for each unique $spname$). Finally Select in box 5 has the same key column as its input operator ($spname$).

Since an operator can have more than one key, we use the term canonical key to
denote a unique key for the operator. Specifically, for a table operator, we use its primary
key. The canonical keys for the other operators can be defined in terms of the canonical
keys of its input operators. Table 3.1 defines the canonical key for each operator (except
for Unnest) in terms of the canonical keys of its input operator(s). The canonical key
for Select and Project operators is the same as the canonical key as their input operator.
Note that we use bag (not set) semantics for operators; thus the output of the project
operator has the same number of rows as its inputs, and can hence propagate the input
keys. The canonical key of the Join operator is the concatenation of the canonical keys
of its input operator (since each row produced by the join contains columns from both
its inputs). The canonical key for the Union operator is the union of the canonical key
columns of its inputs plus a position column; the position column has a unique value
(i) corresponding to each input operator and is thus used to distinguish duplicate rows
from different inputs. The canonical key for the SetDiff operator is the canonical key of
its left input. Finally, the canonical key for the Nest operator is the set of non-nesting
columns since exactly one row is produced for each distinct combinations of values of
the non-nesting columns. We note that the above rules ensure that the canonical key for
an operator is unique.

We use the notation \( \text{ck}_o \) to denote the set of canonical key columns for an operator
\( o \). \( \bar{C}(o) \) is used to denote the set of all columns produced by operator \( o \). For a row \( r \)
produced by an operator \( o \) and a set of columns \( C \subseteq \bar{C}(o) \), we use \( v_C(r) \) to denote the
“sub-row” of \( r \) that contains only the values of the columns in \( C \). Thus, \( v_{\text{ck}_o}(r) \) denotes
the row that contains the values of \( r \) corresponding to the canonical key columns of \( o \).
We denote the top operator of an graph \( G \), which produces the final result of the graph,
as \( o_G \). For two rows \( r_1 \) and \( r_2 \), we also use the notation \( r_1 = r_2 \) if \( r_1 \) and \( r_2 \) have equal
values for all columns, and \( r_1 \neq r_2 \) otherwise.
Table 3.1: Deriving canonical keys for operators.

<table>
<thead>
<tr>
<th>Operator type</th>
<th>Input (operator, key) pairs</th>
<th>How to derive output key (Key_O)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select, Project</td>
<td>(I, Key_I)</td>
<td>/* Propagate the key of our input operator. */ Key_O ← Key_I</td>
</tr>
<tr>
<td>Join</td>
<td>(I_1, Key_{I_1}), ⋯, (I_n, Key_{I_n})</td>
<td>/* New key is the concatenation of input keys. */ Key_O ← Key_{I_1} ∪ ⋯ ∪ Key_{I_n}</td>
</tr>
<tr>
<td>Union</td>
<td>(I_1, Key_{I_1}), ⋯, (I_n, Key_{I_n})</td>
<td>∀i ∈ I, let pos_i be the new column added to I_i which always produces the value i, let PosKey_{I_i} be Key_{I_i} ∪ {pos_i}, let pos_O be the column in O which unions all pos_i, and let M : C_I → C_O be the mapping from the columns of input operators to columns of O. Key_O ← ∪<em>{i ∈ I} (∪</em>{e ∈ PosKey_{I_i}} M(e))</td>
</tr>
<tr>
<td>SetDiff</td>
<td>(I_{left}, Key_{I_{left}}), (I_{right}, Key_{I_{right}})</td>
<td>/* New key is the key of the left input. */ Key_O ← Key_{I_{left}}</td>
</tr>
<tr>
<td>Nest</td>
<td>(I, Key_I)</td>
<td>Key_O ← the non-nesting columns of O.</td>
</tr>
<tr>
<td>Table</td>
<td>—</td>
<td>Key_O ← the primary key of O.</td>
</tr>
</tbody>
</table>

In order to define updates, inserts, and deletes on a view, we first formalize the notation for a database transition, which is the result of UPDATES, INSERTS, and/or DELETES on relational tables. We do so in terms of the database state, where the database is in a state D before the transition, and a different state D' after the transition; we write the transition itself as D ↦ D’. When considering the effect of UPDATES, INSERTS, and/or DELETES to a single table T (as is the case when a SQL trigger on T is fired), we denote the transition as D T ↦ D’. The result of evaluating operator o in state D is written R(o, D).

We now define updates, inserts and deletes on views.
Definition 3.2.2 (View Trigger Updates). A row $r$ is said to be updated in view $G$ by relational transition $D \xrightarrow{*} D'$ iff $r \in R(o_G, D)$, and $\exists r'(r' \in R(o_G, D') \wedge v_{ck_G}(r) = v_{ck_G}(r') \land r \neq r')$.

Definition 3.2.3 (View Trigger Inserts). A row $r$ is said to be inserted in view $G$ by relational transition $D \xrightarrow{*} D'$ iff $r \in R(o_G, D')$ and $\neg \exists r'(r' \in R(o_G, D) \wedge v_{ck_G}(r) = v_{ck_G}(r'))$.

Definition 3.2.4 (View Trigger Deletes). A row $r$ is said to be deleted in view $G$ by relational transition $D \xrightarrow{*} D'$ iff $r \in R(o_G, D)$ and $\neg \exists r'(r' \in R(o_G, D') \wedge v_{ck_G}(r) = v_{ck_G}(r'))$.

Since we use canonical keys to uniquely identify rows, if the value of the canonical key of a row $r$ is updated, we do not consider $r$ as being updated; $r$ is considered to be deleted and a new row $r'$ with the new key value is inserted.

Given the above definition of events, we use the semantics of the triggers described in Section 3.1.1. We note that our events are well-defined only for operators with canonical keys. We thus need to define a class of views for which triggers are well-defined.

Definition 3.2.5 (Trigger-Specifiable Views). A view with graph $G$ is trigger-specifiable iff every operator in $G$ has a canonical key.

We require every operator (not just the top operator) in the view to have a canonical key because the user can specify a trigger on a deeply nested row (and not just a top level row). We now prove that a view is trigger-specifiable if all its Table operators have canonical (i.e., primary) keys.

Theorem 3.2.6. A view of relational data, $G$, is trigger-specifiable if all the table operators in $G$ have canonical keys.
Proof. (Sketch)

We need to prove that every operator in $G$ has a canonical key. In Table 3.1, we define the canonical keys for every type of operator, except for \textbf{Unnest}, in terms of its input operator(s). Thus, if $G$ does not contain any \textbf{Unnest} operators, then we can simply derive the canonical key for each operator $o$ by applying the definitions in Table 3.1.

If $G$ \textit{does} contain \textbf{Unnest} operators, then it can be rewritten to an equivalent graph $G'$ that does not contain any \textbf{Unnest} operators using the sound and complete view composition rules proposed in [106]. This transformation is possible because $G$ is an nested view of relational data and the underlying relational data contains no inherent nesting. Hence, an \textbf{Unnest} operator can only unnest a nesting structure created in the view itself. We can thus assume without loss of generality that $G$ does not contain any \textbf{Unnest} operators.

Since all operators in $G$ have canonical keys, the view is trigger-specifiable by Definition 3.2.5.

Thus, arbitrarily complex views can have triggers specified on them, so long as the underlying relational tables have primary keys (which is the common case).

We note that there is a subtle issue that arises if the canonical key does not appear in the nested view (for example, if the $\textit{pname}$ column is not propagated beyond box 4 in Figure 2.5). In this case, the user may not be able to detect whether a view row has been inserted, updated or deleted because the canonical key is not visible to the user, even though the system can perform the correct action using the propagated canonical keys. Note, however, that this is not a new problem introduced by views, but also exists for base relational data. For example, consider a base table that contains duplicate rows (which is allowed by commercial relational systems). Although the duplicate rows contain the same column values (visible to the user), they have distinct record ids (rids)
that the system can use to uniquely identify the rows. Now consider a user who opens an updateable cursor on the table, and updates exactly one of the duplicate rows (whichever row is returned first by the cursor). The system knows which of the two duplicate rows is updated, and will thus fire the trigger for that row. However, this action is non-deterministic from the user’s point of view. This case is similar to the case where the canonical keys do not appear in the nested view and should be interpreted similarly.

For the rest of this paper, we only consider trigger-specifiable views. Since every operator in a trigger-specifiable view has a canonical key, the associated relations are sets (with no duplicate rows); thus, we use bag and set semantics interchangeably henceforth.
3.2.2 System architecture

Our system architecture is shown in Figure 3.2. Users can create triggers (using the syntax in Section 3.1.1) on trigger-specifiable views. The Target, Condition and parameters of the Action of the trigger are converted into their respective operator graphs (recall that Target, Condition and parameters of the Action are all nested relational algebra expressions). The trigger Event and the Target graph are then analyzed by the Event Pushdown module to determine the minimal set of base relations on which inserts, updates, or deletes could cause the trigger to be fired. For each of these tables, the Affected-Row Graph Generator constructs an operator graph which, when evaluated, produces the OLD_ROW and NEW_ROW values for each affected row. This graph is fed into the Trigger Grouping module, which groups similar triggers together for improved scalability. The Trigger Pushdown module takes the grouped trigger graph, pushes down selection conditions, and produces a set of SQL triggers, one for each relational event.

When activated, an SQL trigger issues a single SQL query to retrieve the relational data required for the actions of the specified triggers. A constant-space Nester [107] then converts these results into nested relations. The Trigger Activation module then activates the appropriate triggers and passes in the nested relational rows as parameters to their actions.

One limitation of our current implementation is that trigger(s) over views are fired for each SQL INSERT, UPDATE, or DELETE statement, rather than for each SQL transaction (which could contain more than one statement). However, this is not a limitation of our approach itself, but due to the fact that most commercial databases do not support SQL triggers at the transaction level; they only support SQL triggers at the granularity of a statement within a transaction. We note, however, that our approach is general enough to support transaction level triggers if the underlying relational databases exposes trans-
3.2.3 Trigger parsing and event pushdown

The first step in our architecture is to convert the trigger Target, Condition and Action expressions into operator graphs. In addition, we apply view composition rules similar to [106] on the Target expression. This has the effect of removing Unnest operators (since the base data is relational and has no inherent nesting), and identifies the specific part of the view to be monitored by the trigger. For example, the trigger in Section 3.1.1 monitors the part \( \text{catalog.product} \), which corresponds to the nested relational algebra expression \( \mu_{\text{product}}(\text{catalog}) \). On composing this expression with the catalog view, it produces the graph in Figure 2.5A. Note that this graph only produces products and not the entire catalog (since the trigger only monitors particular products). Also note that this graph produces all products because the Condition, which is specified on the product name in our example, is not considered in this phase.

The next step is to determine which events on which relational tables can cause the event specified in the trigger. This is similar to the problem of identifying events on the base tables that can affect materialized views [27] and violate constraints [26]. We adopt a similar approach to identifying relevant events on base tables: for each type of operator (Join, Nest, etc.), and for each of the three event types, there is a set of possible input events that can cause that output event (see Table 3.2). Put another way, there is a set of rules \( E_I \rightarrow E_O \), where operator \( I \) is an input to operator \( O \), such that the event \( E_O \) can occur if \( E_I \) occurs. Thus, starting at the top operator of the Target graph, we can determine the set of events on all of its input operators which can cause the Event specified in the trigger. Applying these rules recursively, as shown in Figure 3.3, we eventually reach the base Table operators, at which point we have the set of all base-
Figure 3.3: **GetSrcEvents**: given an operator, \( o \), and a desired event on that operator, \( e \), returns the set of table-level events which can cause \( e \).

table events \( I_B \) such that \( I_B \rightarrow Event \).

```sql
CREATE TRIGGER Notify AFTER Update
ON catalog.product
WHERE OLD_ROW.pname = 'CRT 15'
DO notifySmith(NEW_ROW.pname, NEW_ROW.vendor)
```

In our trigger example (repeated above), we are interested in UPDATE on all columns of Box 5 in Figure 2.5. According to Table 3.2 this translates to UPDATE on all columns of Nest in Box 4 . For Box 4, \$vendor\ is the result of the nest() function, which groups multiple rows with the same \$pname\; thus, any INSERT, DELETE and UPDATE to columns \$vid, $pid and $price\ of box 3 could cause \$vendor\ in its corresponding row in box 4 to be updated. Therefore UPDATE on \$vendor\ on Box 4 translates to INSERT, DELETE and UPDATE on \$vid, $pid and $price\ of box 3. By repeating this process using the rules in Table 3.2, we can finally derive that an UPDATE on Box 5 can be caused either by an INSERT, UPDATE or DELETE on the product table or the vendor table.
Table 3.2: Operator-specific rules used in event pushdown.

<table>
<thead>
<tr>
<th>Operator type</th>
<th>Output event</th>
<th>Input (operator, event) pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Select, Project</strong></td>
<td><strong>DELETE</strong></td>
<td>DELETE(I) (I is the input operator); UPDATE(I, C_σ) where C_σ are the columns used in the selection condition</td>
</tr>
<tr>
<td></td>
<td><strong>INSERT</strong></td>
<td>INSERT(I); UPDATE(I, C_σ)</td>
</tr>
<tr>
<td></td>
<td><strong>UPDATE</strong></td>
<td>UPDATE(I, C)</td>
</tr>
<tr>
<td><strong>Join</strong></td>
<td><strong>DELETE</strong></td>
<td>DELETE(I) for any input I; UPDATE(I, C_I), where C_I are the columns of operator I.</td>
</tr>
<tr>
<td></td>
<td><strong>INSERT</strong></td>
<td>INSERT(I) for any input I; UPDATE(I, C_I)</td>
</tr>
<tr>
<td></td>
<td><strong>UPDATE</strong></td>
<td>UPDATE(I, C_I)</td>
</tr>
<tr>
<td><strong>Nest</strong></td>
<td><strong>DELETE</strong></td>
<td>DELETE(I); UPDATE(I, G), where G is the set of non-nesting columns</td>
</tr>
<tr>
<td></td>
<td><strong>INSERT</strong></td>
<td>INSERT(I); UPDATE(I, G)</td>
</tr>
<tr>
<td></td>
<td><strong>UPDATE</strong></td>
<td>UPDATE(I, C); INSERT(I) unless C ⊆ G; DELETE(I) unless C ⊆ G</td>
</tr>
<tr>
<td><strong>Union</strong></td>
<td><strong>DELETE</strong></td>
<td>DELETE(I) for any input operator I; UPDATE(I, C_I) for any input operator I (Note that DELETE(O) could be caused by an UPDATE where a previously unique row becomes a duplicate.)</td>
</tr>
<tr>
<td></td>
<td><strong>INSERT</strong></td>
<td>INSERT(I) for any input operator I; UPDATE(I, C_I) for any input operator I (analogously to DELETE(O))</td>
</tr>
<tr>
<td></td>
<td><strong>UPDATE</strong></td>
<td>UPDATE(I, C_I) for any input operator I</td>
</tr>
<tr>
<td><strong>SetDiff</strong></td>
<td><strong>DELETE</strong></td>
<td>DELETE(I_{left}); INSERT(I_{right}); UPDATE(I_{left}, C) for any column C; UPDATE(I_{right}, C) for any column C;</td>
</tr>
<tr>
<td></td>
<td><strong>INSERT</strong></td>
<td>INSERT(I_{left}); DELETE(I_{right}); UPDATE(I_{left}, C) for any column C; UPDATE(I_{right}, C) for any column C;</td>
</tr>
<tr>
<td></td>
<td><strong>UPDATE</strong></td>
<td>UPDATE(I_{left}, C)</td>
</tr>
</tbody>
</table>
3.3 Affected-Row Graph Generation

One of the main technical contributions of this paper is the Affected-Row Graph Generator (see Figure 3.2), which produces operator graphs that compute the input parameters for the trigger action. Specifically, the module takes as input the graphs for the Target, Condition, and parameters for the Action, and also the set of relational table-event pairs identified by the Event Pushdown module. For each of these table-event pairs, it produces an graph which computes the transformation from the relational transition tables to the parameters for the trigger action.

Our high-level approach is to produce a single graph, \( G_{\text{params}} \), which consists of three parts as shown in Figure 3.4. First, \( G_{\text{affected}} \) produces an \((\text{OLD\_ROW}, \text{NEW\_ROW})\) row for each affected row of the view. Next, \( G_{\text{cond}} \), the graph corresponding to the Condition, filters out any rows that do not satisfy the condition. Finally, \( G_{\text{action}} \) computes the nested algebra expressions given as parameters to the Action. The main technical contribution of this section is an algorithm to produce \( G_{\text{affected}} \).
3.3.1 Technical challenges in producing $G_{\text{affected}}$

On the surface, the problem of producing $G_{\text{affected}}$ may appear similar to the incremental view maintenance problem (where the goal is to compute changes to a materialized view based on updates to the base data). However, there are three new challenges that arise in our context, which require the development of new techniques.

First, as mentioned in the introduction, one of our design goals is to not materialize the nested view. We avoid materialization because (a) it would require a sophisticated middleware database that can support incremental view updates and triggers over nested views, and (b) it would require the view to be updated for every relevant relational update even though user triggers may have very selective predicates$^1$. In contrast, most incremental view maintenance algorithms (e.g., [3, 17, 27, 47, 48, 57, 80, 82, 101]) assume that the view is materialized, and use the materialized old value of a row to compute its new value. We thus need to devise techniques that can selectively compute the relevant new values directly from the base data.

Second, in producing $G_{\text{affected}}$, we need to compute new and old values after an update. In contrast, for materialized views, we only need to compute the new value because the old value already exists (since it is materialized). Thus, even materialized view techniques that can selectively compute new values without using materialized old values (e.g., [94, 97]) are not applicable because they cannot compute (old value, new value) pairs for all INSERT, DELETE and UPDATE events. This problem is especially acute for INSERT/DELETE events because they introduce specific restrictions on whether the old/new values can appear in the view before/after an update (Definition 3.2.3).

$^1$Note that if we chose to materialize the view, all rows in the view (even those that do not satisfy any trigger selection predicate) would have to be incrementally maintained, because any row could become the old value of an updated item that does satisfy a trigger predicate.
Third, triggers over nested views need an efficient way to check whether the old and new values, which may be deeply nested, are in fact different. Such a problem may arise when the view uses a function such as max, where the old and new value may remain the same even when the base data is updated. Note that this problem is especially acute for nested views because explicitly checking whether two deeply nested structures are different can be very expensive.

Finally, the fourth (and perhaps most important) challenge arises due to nested predicates. For instance, in Figure 2.5, we have a Nest operator along with a selection predicate on an aggregate value. While prior work on view maintenance for object-oriented [57, 82, 101], nested relational [80] and semi-structured [3, 17, 47, 48] databases support nesting, these do not work with nested predicates. To understand why, consider the following example where a transaction inserts a row into the vendor table. The transition table is:

<table>
<thead>
<tr>
<th>vid</th>
<th>pid</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon</td>
<td>P2</td>
<td>500.00</td>
</tr>
</tbody>
</table>

Intuitively, for the nested view in Figure 2.5, the above insert corresponds to an update of the “LCD 19” product (since a new vendor is added to this product). However, it turns out that the change computation technique (also referred to as the propagate phase [94]) commonly used for view maintenance will not detect this update. Specifically, most view maintenance algorithms compute changes to a view by replacing an updated table in the view definition with its corresponding transition table. In our example, this corresponds to replacing the vendor table in Figure 2.5 with the $\Delta$vendor table, and evaluating the resulting query to compute the changes to the view. However, since $\Delta$vendor has a single row, boxes 2 and 3 will each produce a single row, and the selection predicate in box 5 will return no rows since $cnt = 1$. Hence, no changes will be detected!
As the reader has probably observed, the above problem arises because we are trying to compute changes for nested predicate views using only rows from the transition table. This results in inaccurate aggregate values and hence misses some relevant updates (it can also introduce spurious updates in other cases). We thus need to devise techniques for correctly computing changes for views with nested predicates. We note that [3] does present a technique for computing changes to views with existential predicates and a single level of nesting (existential predicates can be seen as a very specific form of a selection over an aggregation), but we are not aware of any prior technique that can handle complex query predicates at arbitrary levels of nesting.

3.3.2 Proposed algorithm - CreateAKGraph & CreateARGraph

We now present our algorithm for producing $G_{affected}$. The algorithm first detects the keys of the rows affected by an update (affected keys) and then use the affected keys to compute the actual values. Our main contributions are (a) a technique for correctly determining affected keys even when the view has arbitrary nested predicates, and (b) a technique for using the affected keys to generate $(OLD\_ROW, NEW\_ROW)$ pairs that satisfy the definition of trigger events, without using any materialized data. In the next section, we also show how we can avoid explicitly checking whether $OLD\_ROW$ and $NEW\_ROW$ are different for a certain class of views.

In what follows, we use the following notation. $G$ is the Target graph; $T$ is the post-update version of the table in question (recall that this algorithm is invoked once for each table-event pair); $T_{old}$ is the pre-update version of this table; $G_{old}$ is a graph identical to $G$ with the sole exception that $T$ is replaced by $T_{old}$. While most relational database systems do not expose the $T_{old}$ table directly, it can easily be constructed using a query of the form [41]:
(SELECT * FROM T) EXCEPT (SELECT * FROM ∆T)
UNION (SELECT * FROM ▽T)

CreateAKGraph: finding affected keys

In Figure 3.5, we present our algorithm for determining the affected keys.\(^2\) The algorithm takes as input an operator \(O\) (the top operator in the Target graph \(G\)), an operator \(O_{old}\) (the top operator in \(G_{old}\)), and a base table \(T\). It returns the top operator, \(O'\), of an operator graph and a set of key columns of \(O', K\), which satisfy the following two properties: (1) \(O \Join_K O'\), when evaluated, produces the subset of \(O'\)’s output rows that are inserted or updated by the relational update captured by \(\Delta T\) and \(\nabla T\), and (2) \(O_{old} \Join_K O'\), when evaluated, produces the subset of \(O_{old}\)’s output rows that are deleted or updated by the relational update captured by \(\Delta T\) and \(\nabla T\).

In order to determine the keys of \(G\) affected by \(\Delta T\) and \(\nabla T\), we traverse \(G\) and \(G_{old}\) in depth-first order and build up a parallel graph \(G_{par}\). At each step, we maintain the following invariant: for each operator \(o\) in \(G\), \(o_{old}\) in \(G_{old}\) and the corresponding operator \(o_{par}\) in \(G_{par}\) (1) joining \(o\) and \(o_{par}\) on the key of \(o_{par}\) will produce the rows in the result of \(o\) that were inserted or updated by \(\Delta T\), and (2) joining \(o_{old}\) and \(o_{par}\) on the key of \(o_{par}\) will produce the rows in the result of \(o_{old}\) that were deleted or updated by \(\Delta T\) and \(\nabla T\). Thus, if \(o\) is the top operator of \(G\), then the corresponding \(o_{par}\) operator provides a way to identify the rows in the result of \(G\) affected by the relational update.

We now walk through the algorithm using the Target graph in Figure 2.5 for the case of an UPDATE on vendor (the other cases are similar). At the leaf level, a Union operator will first be created (lines 4-8) which computes the union of Table(\(\Delta\)vendor) and Table(\(\nabla\)vendor) (boxes 1\(_{par}\), 2\(_{par}\) and 3\(_{par}\) in Figure 3.7). Clearly the invariant holds at this point: joining 2\(_{par}\) with Table(vendor) on the $vid$ column (the key of 2\(_{par}\)) will

\(^2\)Note that this algorithm does not explicitly consider Unnest operators, which are eliminated by view composition (see Sec. 3.2.3 and [106]).
1: \textbf{CreateAKGraph} (O, O_{old}, T) : (Operator, Key)
\{O is an operator in G; O_{old} is the corresponding operator in G_{old}; T is a table name.\}
2: i ← input operators to O, I_{old} ← input operators to O_{old}
3: if \text{O.type} = \text{Table} then
4: \hspace{1em} if O\_tableName = T then
5: \hspace{2em} PK ← primary key of table T
6: \hspace{2em} O' ← \text{Union}(\text{Project}_{K}(\Delta T), \text{Project}_{K}(\gamma T))
7: \hspace{2em} K ← PK
8: \hspace{1em} else (O', K) ← (Nil, Nil)
9: \hspace{1em} else if \text{O.type} = \text{Nest} then
10: \hspace{2em} (P', K') ← \text{CreateAKGraph}(I, I_{old}, T)
11: \hspace{2em} if I' = Nil then (O', K) ← (Nil, Nil)
12: \hspace{1em} else
13: \hspace{2em} J_{new} ← \text{Join}_{K'}(I', P'), J_{old} ← \text{Join}_{K'}(I_{old}, P)
14: \hspace{2em} U ← \text{Union}(J_{new}, J_{old})
15: \hspace{2em} K ← non-nesting columns of O
16: \hspace{2em} O' ← new Nest on U with non-nesting cols K
17: \hspace{1em} end if
18: \hspace{1em} else if \text{O.type} = \text{Select} or \text{O.type} = \text{Project} then
19: \hspace{2em} (O', K) ← \text{CreateAKGraph}(I, T_{old}, T)
20: \hspace{1em} else if \text{O.type} = \text{Join} then
21: \hspace{2em} (I_{0}', K_0) ← \text{CreateAKGraph}(I_0, I_{old}, T)
22: \hspace{2em} (I_1', K_1) ← \text{CreateAKGraph}(I_1, I_{old}, T)
23: \hspace{2em} if I_0' = Nil \land I_1' = Nil then (O', K) ← (Nil, Nil)
24: \hspace{2em} else if I_0' \neq Nil \land I_1' = Nil then O' ← I_0', K ← K_0
25: \hspace{2em} else if I_1' \neq Nil \land I_0' = Nil then O' ← I_1', K ← K_1
26: \hspace{1em} else
27: \hspace{2em} \{Let \text{JoinPrd} be the join predicates in O\}
28: \hspace{2em} K ← \text{ckO}
29: \hspace{2em} \{Join I_0' with I_0, and then with I_1 to retrieve affected rows\}
30: \hspace{2em} J_{new0} ← \text{Join}_{K_0}(I_0', I_0), J_{old0} ← \text{Project}_{K_0}(\text{Join}_{\text{JoinPrd}}(J_{new0}, I_1))
31: \hspace{2em} \{Join I_0' with I_{old0}, and then with I_1 to retrieve rows affected\}
32: \hspace{2em} J_{new0} ← \text{Join}_{K_0}(I_0', I_{old0}), J_{old0} ← \text{Project}_{K_0}(\text{Join}_{\text{JoinPrd}}(J_{old0}, I_{old}))
33: \hspace{2em} U_0 ← \text{Union}(J_{new0}, J_{old0})
34: \hspace{2em} \{Do the same to I_1' and I_1\}
35: \hspace{2em} J_{new1} ← \text{Join}_{K_1}(I_1', I_1), J_{old1} ← \text{Project}_{K_1}(\text{Join}_{\text{JoinPrd}}(J_{new1}, I_0))
36: \hspace{2em} J_{old1} ← \text{Join}_{K_1}(I_1', I_{old1}), J_{old1} ← \text{Project}_{K_1}(\text{Join}_{\text{JoinPrd}}(J_{old1}, I_{old}))
37: \hspace{2em} U_1 ← \text{Union}(J_{new1}, J_{old1})
38: \hspace{2em} \{Create a Union to propagate all keys\}
39: \hspace{2em} O' ← \text{Union}(U_0, U_1)
40: \hspace{1em} end if
41: \hspace{1em} else if \text{O.type} = \text{Union} then \{Refer to Figure 3.6.\}
42: \hspace{1em} else if \text{O.type} = \text{SetDiff} then \{Refer to Figure 3.6.\}
43: \hspace{1em} end if
44: \{Ensure that O's key is propagated\}
45: \{Add K to O.outputColumns\} \textbf{return} (O', K)

Figure 3.5: Algorithm for producing affected keys.
1: CreateAKGraph (O, O_{old}, T) : (Operator, Key)
   \{ O is an operator in G; O_{old} is the same operator in G_{old}; T is a table name. \}
2: if O.type = Table then
3:   \{ Create a Project on I_0 which adds an additional $position$ column producing 0 \}
4: else if O.type = Union then
5:   (I_0', K_0) ← CreateAKGraph(I_0, I_{0,old}, T)
6:   (I_1', K_1) ← CreateAKGraph(I_1, I_{1,old}, T)
7:   if I_0' = nil ∧ I_1' = nil then \{ Union of all keys \}
8:     \{ Create a Project on I_0 which adds an additional $position$ column producing 0 \}
9:   else if I_0' \neq nil ∧ I_1' = nil then \{ Project \}
10:      \{ Projection of all columns \}
11:     \{ Create a Union to propagate all keys \}
12:     \{ Join I_0'(I_1') and I_0(I_1') on the keys to retrieve all columns \}
13:     \{ Join on the keys to retrieve all columns \}
14:     \{ Join \}
15:     \{ Join \}
16:     \{ Join \}
17:     \{ Join \}
18:     \{ Join \}
19:     \{ Join \}
20:     \{ Join \}
21:     \{ Join \}
22:     \{ Join \}
23:     \{ Join \}
24:     \{ Join \}
25:     \{ Join \}
26:     \{ Join \}
27:     \{ Join \}
28:     \{ Join \}
29:     \{ Join \}
30:     \{ Join \}
31:     \{ Join \}
32:     \{ Join \}
33:     \{ Join \}
34:     \{ Join \}
35:     \{ Join \}
36:     \{ Join \}
37: end if
38: end if
39:
40: \textbf{Figure 3.6:} Algorithm for producing affected keys: \textbf{SetDiff} and \textbf{Union}
Figure 3.7: CreateAKGraph: Step 1.

produce the vendor rows that are inserted or updated (with values after updates), and joining $3_{par}$ with the old Table(vendor) on the $Svid$ column will produce the vendor rows that are deleted or updated (with the values before updates).

Box 3 (the Join operator) simply propagates the $G_{par}$ corresponding to its input (lines 18–19). Thus, the top operator in $G_{par}$ remains the Union operator (box $2_{par}$ in Figure 3.7), and the invariant still holds: joining box 3 with box $2_{par}$ on $Svid$ would produce the product-vendor pairs inserted or updated, and joining box $3_{old}$ with $2_{par}$ would produce the product-vendor pairs deleted or updated.

We then arrive at box 4, a Nest operator. Since a Nest operator aggregates multiple input values, any update to any one of the input values in a group can change the aggregate result for that group. We therefore need a way to create an operator $o_{par}$ in $G_{par}$ that only produces the keys of those groups affected by the update. This is handled in lines 10–17. First, we join the operator below the current Nest (box 3) with its corresponding operator in $G_{par}$ (box $2_{par}$). By the algorithm invariant, we can infer that this
Figure 3.8: CreateAKGraph: Nest operator.

Figure 3.9: The complete $G_{par}$ graph.
new join produces the set of input rows to the Nest operator that are inserted or updated. Similarly, we need to join the operator $3_{old}$ with operator $2_{par}$ to retrieve the rows that are deleted or updated. Then we create a Union operator, box $6_{par}$ to propagate affected rows from both joins. Finally, to identify the keys of all affected groups, we simply need to project distinct values of the non-nesting columns, which we achieve by creating a new Nest operator (box $7_{par}$). In our example, the non-nesting column is $spname$. The $G_{par}$ graph at this point is shown in Figure 3.8.

The final operator is a Select operator which, like box 3, simply propagates the $G_{par}$ corresponding to its input (lines 18-19). The final graph is shown in Figure 3.9.

3.3.3 Additional examples of the algorithms

We illustrate the CreateAKGraph algorithm (Figure 3.5) for the cases of a self-join and a Union. The case for the SetDiff is similar to that of a Union.

CreateAKGraph on Join

We walk through an example and explain how createAKGraph works with a self-join. Consider the Target graph in Figure 3.10, which produces pairs of vid’s that sell the same product. Box 3 performs a self-join on the table vendor on pid and produce relevant vid’s from boxes 1 and 2.

Intuitively to determine the affected keys, we first determine the keys affected due to changes to individual inputs, and then union them to get all affected keys for the Join (lines 27-39). Here we show how the algorithm creates a sub-graph which, when evaluated, produces affected rows due to changes to an individual input, box 1. Figure 3.11 shows the graph after we apply CreateAKGraph. The algorithm begins by invoking CreateAKGraph on input operators 1 and produces a Union operator $2_{par}$ taking the
Figure 3.10: An additional example for \texttt{createAKGraph}.

Figure 3.11: Parallel graph produced by \texttt{createAKGraph}.
To determine the keys of affected rows due to updates to box 1, we begin by creating a Join, box $4_{par}$, which joins boxes 1 and $2_{par}$ on the key columns. By the invariant of CreateAKGraph, box $4_{par}$ produces all the values of the rows inserted or updated in box 1. Next, to construct all the key columns corresponding to the Join operator 3, we create a new Join box $5_{par}$, which joins box $4_{par}$ with box 2 on the same joining condition as in box 3. This join produces rows in box 3 that are affected by the updates represented by $2_{par}$ and box 1. Similarly, we create boxes $6_{par}$ and $7_{par}$, which produces rows in box 3 that are affected by the updated represented by $2_{par}$ and box $1_{old}$. Finally, we create a Union operator, box $8_{par}$, to propagate all affected keys from both inputs.

We can similarly repeat this process to create a sub-graph, which when evaluated, produces affected rows due to updates to box 2.

While the result appears to have many joins, these joins can be simplified using Union of $\triangle vendor$ and $\triangledown vendor$.

To determine the keys of affected rows due to updates to box 1, we begin by creating a Join, box $4_{par}$, which joins boxes 1 and $2_{par}$ on the key columns. By the invariant of CreateAKGraph, box $4_{par}$ produces all the values of the rows inserted or updated in box 1. Next, to construct all the key columns corresponding to the Join operator 3, we create a new Join box $5_{par}$, which joins box $4_{par}$ with box 2 on the same joining condition as in box 3. This join produces rows in box 3 that are affected by the updates represented by $2_{par}$ and box 1. Similarly, we create boxes $6_{par}$ and $7_{par}$, which produces rows in box 3 that are affected by the updated represented by $2_{par}$ and box $1_{old}$. Finally, we create a Union operator, box $8_{par}$, to propagate all affected keys from both inputs.

We can similarly repeat this process to create a sub-graph, which when evaluated, produces affected rows due to updates to box 2.

While the result appears to have many joins, these joins can be simplified using Union of $\triangle vendor$ and $\triangledown vendor$. 

Figure 3.12: Simplified graph produced by CreateAKGraph.
Consider the Target graph in Figure 3.13, where only the productB table is affected by the relational update. The CreateAKGraph algorithm returns box \(2_\triangle\) corresponding to traditional join elimination and query minimization techniques [30, 7, 102, 96]. For example, we know that all the keys produced by box \(2_{par}\) that join with box \(1\) will be contained in box \(1_{par}\) (and similarly for boxes \(2_{par}, 1_{old}\) and \(3_{par}\)). Further, we know that joining box \(1\) with box \(1_{par}\) will produce exactly \(1_{par}\) (and similarly for boxes \(1_{old}\) and \(3_{par}\)). Using such equivalences, we can significantly simplify the resulting graph. The final simplified graph for our running example is shown in Figure 3.12.

CreateAKGraph on Union

Figure 3.13: CreateAKGraph on Union.

Figure 3.14: Union augmented with \$index\ columns.
box 2 (lines 8-10). Using the keys returned by box \(2_{par}\), we need to find the affected keys for box 3. If we simply propagate the keys for \(2_{par}\) as the affected keys for box 3, the joining box 3 with box \(2_{par}\) on the key for \(2_{par}\) may return incorrect affected rows if we have duplicate rows from box 1 with the same key value. Thus, to differentiate rows from the two inputs, the algorithm creates a new operator \(3_{par}\) that creates an additional \(\$position\) column which always produces the value 2. The algorithm also augments the original graph to propagate \(\$position\) columns. The augmented graph is shown in Figure 3.14. In this graph, joining box 3 with box \(3_{par}\) on the key will produce the correct affected rows.

If both inputs to the union operator are updated (due to, say, a self-union), this case is similar to the case of the self-join (lines 27-39), with the additional detail of propagating the \(\$position\) columns.

At this stage, for a given relational table-event pair, we have the Target graphs \((G\) and \(G_{old}\)) and the affected-key graph \((G_{par}\)). Our next goal is to produce the \(G_{affected}\) graph, which generates the \((OLD\_ROW, NEW\_ROW)\) pairs corresponding to the relational update. The algorithm for producing \(G_{affected}\) is given in Figure 3.15.

CreateARGraph: producing affected rows.

We begin by invoking CreateAKGraph on \(G\) and \(G_{old}\) which returns \((O', K)\) (line 2). \(O'\) is then joined with \(o_G\) to get \(NEW\_ROW\) (line 3), because by the invariant of CreateAKGraph, we know that this would produce rows that are inserted or updated. Similarly, \(O'\) is joined with \(o_{G, old}\) to get \(OLD\_ROW\) (line 4). Finally, \(OLD\_ROW\) and \(NEW\_ROW\) are joined on the key columns of \(o_G\). The type of this join depends on the view trigger Event: an UPDATE has both \(OLD\_ROW\) and \(NEW\_ROW\), hence an inner join (line 5-9), while an INSERT (DELETE) has only \(NEW\_ROW\) \((OLD\_ROW)\), hence a left (right) anti join (lines 10-14). In our example, we perform an inner join since we
1: CreateARGraph (E : Event, G : Graph, T : String) :
   {Build up the affected-row graph for △T and ▽T.}
2:   (O', K) ← CreateAKGraph(oG, oG_gold, T)
   {And join it back with G(Gold to produce NEW_ROWOLD ROW}
3:   O_new ← JoinK(O', oG)
4:   O_old ← JoinK(O', oG_old)
   {Finally, the way we produce Gaffected depends on the type of event}
5:   if E = UPDATE then
6:      {Inner join; we want those nodes which are present in both OLD_ROW and NEW_ROW}
7:      {Note the join is on ckOG which is produced by lines 3-4.}
8:      Gaffected ← JoinckOG(O_new, O_old)
9:      If required, Gaffected ← Select(OLD_ROW ≠ NEW_ROW)(Gaffected)
10:    else if E = INSERT then
11:       {Here we only want those nodes which are not present in OLD_ROW}
12:       Gaffected ← LeftAntiJoinckOG(O_new, O_old)
13:    else if E = DELETE then
14:       Gaffected ← RightAntiJoinckOG(O_new, O_old)
15:    end if

Figure 3.15: Algorithm for producing Gaffected.

are monitoring UPDATES. Figure 3.16 shows the final Gaffected graph for our example. For this particular view, we apply an optimization discussed in Section 3.3.4 and avoid explicitly checking whether or not the values of OLD_ROW and NEW_ROW actually differ.

We prove the correctness of the CreateARGraph algorithm in Section 3.6.

3.3.4 CreateAROpt: optimizations for CreateARGraph

In general, the CreateARGraph algorithm needs to explicitly check whether the OLD_ROW and NEW_ROW values differ in the case of UPDATES (Figure 3.15, line 9); this check is to ensure that a view trigger is not fired unnecessarily due to spurious results. There are two reasons why OLD_ROW and NEW_ROW values can actually be the same, even when the relevant underlying relational rows are updated. The first, somewhat trivial, reason is due to a relational update such as:

UPDATE VENDOR SET PRICE = 1 * PRICE
which will result in the relational transitional tables containing as many rows as there are vendor rows, even though none of the vendor rows actually changed in value. This simple case can be fixed by pruning the transitional tables; that is, for each base table $T$, eliminating those rows which are identical in $\Delta T$ and $\nabla T$. In general, however, this pruning is not sufficient to eliminate the possibility of spurious results. For example, a view might only reference some relational column $c$ by an aggregate function (e.g. $\text{max}(c)$), in which case an update to $c$ may or may not result in the corresponding view rows actually changing. Therefore, even if the transitional tables are pruned, $\text{CreateAKGraph}$ might produce keys of rows that were not actually updated; the final inequality check is thus required to filter out these spurious results.

Our system can perform this final inequality check to eliminate spurious results. However, there is a performance concern because doing this check is likely to be ex-
pensive. First, the check cannot be done at the base level (since it deals with nested relations), but must be done as the Nester, which means that a great deal of computation must be done before determining whether or not the update is spurious. Second, if \( OLD\_ROW \) and \( NEW\_ROW \) are large, the comparison can be quite expensive. Finally, the entire \( OLD\_ROW \) must be computed for the inequality check, even if it is not actually referenced in the trigger action.

To address the above shortcomings, we now identify a general class of views called injective views, for which the Nester does not need to explicitly compare \( OLD\_ROW \) and \( NEW\_ROW \) values, while still ensuring that view triggers are not fired spuriously. Many views, including the running example in the paper, are injective views. For other (non-injective) views, we also present a few optimizations that can push the inequality check down to the relational database in certain cases.

**Optimizing Injective Views**

Intuitively, injective views have the property that there is a one-to-one mapping between each top-level row produced by \( o_c \) (the top operator of the view graph) and the set of relational rows used to construct the row. For such views, including the running example in the paper, the selection condition \( OLD\_ROW \neq NEW\_ROW \) can be eliminated from \texttt{CreateARGraph} if we prune the transitional tables. Specifically, all references to \( \triangle T \) and \( \triangledown T \) in the SQL trigger are replaced with \( \triangle T \) and \( \triangle T \), respectively, where \( \triangle T = \triangle T - \triangledown T \) and \( \triangledown T = \triangledown T - \triangle T \).

We now formally define the notion of an injective view. We begin by defining the contributing set of a row. Intuitively, for a row \( r \) produced by an operator \( o \), the contributing set of \( r \) associated with an input operator \( o_i \) of \( o \) is the set of rows produced by \( o_i \) that are used to compute \( r \).

**Definition 3.3.1** (Contributing set). Given an operator \( (o) \), one of its input operators \( o_i \) contributes to the contributing set of the row \( r \) produced by \( o_i \) if \( o_i \) is the input operator of \( o \) that produces \( r \) and \( o_i \) directly or indirectly references \( r \) in its input list. We denote the contributing set of a row \( r \) produced by \( o \) as \( \text{ContribSet}(o, r) \).
(o_i), a database state (D), and a row (r_o ∈ R(o, D)), the contributing set of r_o is:

**Case 1:** o.type = Select

\[ \zeta(r_o, o_i, o, D) = \{ r_i ∈ R(o_i, D) | r_i = r_o \} \]

**Case 2:** o.type = Project

\[ \zeta(r_o, o_i, o, D) = \{ r_i ∈ R(o_i, D) | v_{\tilde{C}(o)} \cap \tilde{C}(o_i)(r_i) = r_o \} \]

**Case 3:** o.type = Join

\[ \zeta(r_o, o_i, o, D) = \{ r_i ∈ R(o_i, D) | r_i = v_{\tilde{C}(o_i)}(r_o) \} \]

**Case 4:** o.type = Union

\[ \zeta(r_o, o_i, o, D) = \{ r_i ∈ R(o_i, D) | r_i = r_o \} \]

**Case 5:** o.type = SetDiff

\[ \zeta(r_o, o_i, o, D) = \begin{cases} 
{ r_i ∈ R(o_i, D) | r_i = r_o }, & \text{if } o_i \text{ is the left input} \\
R(o_i, D), & \text{if } o_i \text{ is the right input}
\end{cases} \]

**Case 6:** o.type = Nest with non-nesting columns nc

\[ \zeta(r_o, o_i, o, D) = \{ r_i ∈ R(o_i, D) | v_{nc}(r_i) = v_{nc}(r_o) \} \]

Intuitively, For a **Project** or **Select**, the contributing set of a row r is the single input row from which r is computed by projection or selection, respectively. For **Join**, the contributing set of a row r with respect to a given input o_i is the single row r_i produced by o_i that joined with an input row from the other input to produce r. For **Union**, the contributing set of a row for a given input o_i is the same row produced by o_i (if o_i did produce such a row; note that either or both inputs could have produced that row). For **SetDiff**, the contribution set of a row r with respect to the left input o_l is the same row produced by o_l; the contributing set with respect to the right input o_r is the entire set of input rows because all of them together determine the existence of r (by not including r in the input). The contributing set of a row r produced by a **Nest** operator is the set of input rows that have the same non-nesting column values as r.
In the following, we denote projection for a set of rows $S$: $\pi_C(S) = \{v_C(t) | t \in S\}$.

Further, when $C$ is a set of columns belonging to multiple operators, then $(C|o)$ denotes the subset of $C$ belonging to operator $o$; i.e. $(C|o) = C \cap \tilde{C}(o)$.

We now define injection for a single operator in terms of the contributing set of each row produced by the operator.

**Definition 3.3.2** (Injection for operators). Given an operator $(o)$ with a set of input operators $(I)$, a set of $o$’s columns $(C_o)$, and a subset of $I$’s columns $(C_I)$: the columns $C_o$ are injective with respect to the columns $C_I$ (denoted as $C_I \rightarrow \rightarrow C_o$) iff:

$$\forall r_1, D_1, r_2, D_2 (r_1 \in R(o, D_1) \land r_2 \in R(o, D_2) \land v_{C_o}(r_1) = v_{C_o}(r_2)) \rightarrow (\forall o_i \in I(\pi_{C_I|o_i})(\zeta(r_1, o_i, o, D_1)) = \pi_{C_I|o_i}(\zeta(r_2, o_i, o, D_2)))).$$

Definition 3.3.2 states that $C_I \rightarrow \rightarrow C_o$ iff whenever two rows produced by $o$ have the same values for columns $C_o$, their contributing sets have the same set of values for columns $C_I$. In other words, if $C_I \rightarrow \rightarrow C_o$, then there is a one-to-one mapping such that for each row $r$ produced by $o$, $v_{C_o}(r)$ maps to a unique set of $C_I$ values produced by the input operator(s) $I$.

For a view graph $G$, we use $OP_G$ to denote the set of operators in $G$. For an operator $o \in OP_G$, we use $G_o$ to denote the sub-graph of $G$ with the top operator $o$.

**Definition 3.3.3** (Transitive injection). An operator $o$ is transitively injective with respect to a table $T$ for a subset of its output columns $C_o$ (denoted $T \rightarrow \rightarrow C_o$) iff one of the following holds:

- Table$(T) \notin OP_{G_o}$
- Table$(T) \in OP_{G_o}$ and
- \( o = \text{Table}(T) \) and \( C_o = C(o) \), or
- \( \exists C_I((C_I \mapsto C_o) \land \forall o_i((o_i \in I) \rightarrow (T \mapsto (C_I|o_i)))) \).

That is, an operator \( o \) is transitively injective for a subset of its output columns, \( C_o \), if and only if there is a one-to-one mapping such that for each row \( r \) produced by the operator \( o \), \( v_{C_o}(r) \) maps to a unique set of rows in \( \text{Table}(T) \).

**Definition 3.3.4 (Injective View).** A view with graph \( G \) is injective with respect to a table \( T \) iff \( T \mapsto C \), where \( C \) is the set of output columns of \( o_G \).

Although the condition for an injective view may seem restrictive, most nested views of relational data are injective with respect to each of their base tables. For example, the original catalog view (Figure 2.5) is injective with respect to both product and vendor.

In the *electronic appendix*, we prove that for injective views, CreateARGraph will not produce spurious result if we remove the inequality check in line 9; we refer to this modified version as CreateAROpt.

**Sufficient Conditions for Injection**

For each operator \( o \) in a graph \( G \), and a set of columns \( C_o \), we can determine whether \( C_i \mapsto C_o \) based on the type of operator \( o \):

- **Project, Select, and Join.** \( C_I \mapsto C_o \) if \( \forall c_i \in C_I, c_i \in C_o \).

- **Nest.** \( C_i \mapsto C_o \) if \( \forall c_i \in C_i \), one of the following holds:
  - \( c_i \in C_o \), or
  - \( \exists c \in C_o \) such that \( c = nest(..., c_i, ...) \).

- **SetDiff.** Assuming \( o_l \) is \( o \)'s left input operator, and \( o_r \) is \( o \)'s right input operator. \( C_I \mapsto C_o \) if,
\[ \forall c_i \in (C_1 | o_t), c_i \in C_o, \text{ and} \]
\[ \forall D_1, D_2, D_1 \xrightarrow{T} D_2 \Rightarrow R(o_r, D_1) = R(o_r, D_2). \]
Intuitively this means the set of rows produced by \( o_r \) remains unchanged for all database transitions involving \( T \).

It is easy to see that the view in Figure 2.5 satisfies the above conditions. Note that the above conditions are sufficient but not necessary for injection.

**Additional optimizations**

**Base table pruning**

The definition we have given for injective views is a little stronger than it needs to be; one specific instance in which it can be relaxed is when a view is injective with respect to a subset of a base table’s columns. For example, if in the example view the vendor table had an additional non-key column (such as \( \text{vendorWebSite} \)) which was not referenced anywhere in the view, the view would no longer be injective; an update to this column could cause a trigger to be fired even though it does not result in an update to the view. Nonetheless, there is still a one-to-one mapping from base table rows to vendor values, so intuitively, we should be able to avoid any additional overhead by simply ignoring updates that affect this extra column. Indeed, many relational database systems support triggers on specific columns of a table.

This intuition can be captured formally as follows: for a given table operator, \( o = \text{Table}(T) \), for any set of columns \( C_o \subseteq \tilde{C}(o) \) such that \( C_o \cap ck_o = ck_o \), it follows from Definition 3.3.2 that \( \tilde{C}(o) \mapsto C_o \). Therefore, we can modify the first case in Definition 3.3.3. Initially, we required that:

\[ o = \text{Table}(T) \text{ and } C_o = \tilde{C}(o), \]

which we can relax to:
o = Table(T) and \( C_o \cap ck_o = ck_o \).

Note that the correctness of the trigger, then, requires that only updates to the columns \( C_o \) be captured. This can easily be achieved when pruning the transition tables: \( \Delta T \) and \( \nabla T \), respectively, can be expressed as follows:

```sql
SELECT C_o FROM \( \Delta T \) EXCEPT
SELECT C_o FROM \( \nabla T \)
```

and:

```sql
SELECT C_o FROM \( \nabla T \) EXCEPT
SELECT C_o FROM \( \Delta T \)
```

**Optimizations for non-injective views**

If a view is not injective, then in general, we need to explicitly check whether \( OLD_ROW \) and \( NEW_ROW \) differ. In certain cases, however, we can still optimize this check by pushing it down to the relational level. One such case arises for non-injective views, which would have been injective except for the presence of a non-injective aggregate function (e.g. \( \min \), \( \max \), \( \text{count} \), etc.) in a Nest operator. In this case, it is not necessary to compare the entire \( OLD_ROW \) and \( NEW_ROW \). Instead, it is only necessary to compare the values of these aggregates. Since this is a comparison of numeric values with no nesting involved, it can be pushed down to the relational engine, thus avoiding the need to perform an expensive view-level comparison.

This is just one of many possible optimizations to avoid performing a nester-level comparison for non-injective views. A direction of future work is to identify the general class of views where the final inequality check can be pushed down to the relational level.
3.3.5 Adding Condition and Action

Finally, as described in the beginning of this section, we need to produce $G_{params}$, the graph that produces parameters for the Action after selecting only the (OLD_ROW, NEW_ROW) pairs that satisfy the trigger condition (recall Figure 3.4). $G_{params}$ is produced by converting the nested relational algebra expressions for Condition and parameters of Action into their respective graph representations (to produce $G_{cond}$ and $G_{action}$, respectively), and stacking these graphs on top of $G_{affected}$. Figure 3.16 shows $G_{params}$ for our running example.

3.4 Trigger Grouping and Pushdown

Given $G_{params}$ for each table-event pair, the final two steps in generating SQL triggers are Trigger Grouping and Trigger Pushdown (Figure 3.2). We describe each in turn.

3.4.1 Trigger grouping

A simple approach to producing SQL triggers is to create one SQL trigger for each $G_{params}$ graph of an top-level trigger. However, this approach is not likely to be very efficient because the number of SQL triggers produced will be at least as many as the number of top-level triggers (which we expect to be large), and current relational
databases are not very scalable with respect to the number of SQL triggers. We therefore explore techniques for grouping structurally similar $G_{params}$ graphs together, and producing a single SQL trigger for each group. We note that our focus is not on developing new techniques for grouping triggers; rather, our focus is on adapting existing techniques [34, 67] to work with nested views.

For the purposes of this paper, we only consider grouping $G_{params}$ graphs that differ in the constant value(s) of a selection condition (this corresponds to grouping structurally similar triggers that only differ in selection constant(s) in the WHERE clause). For instance, we would consider grouping the $G_{params}$ graph in Figure 3.16 with another graph that has a different selection condition in box 14 (which, say, selects “LCD 19” instead). The proposed approach can also be extended for grouping joins [34], but we do not discuss this extension here.

The first step is to create a constants table [67] for each group of structurally similar $G_{params}$ graphs. The constants table has a TrigIDs column, which identifies the triggers which share a particular set of constants, followed by as many columns as there are constants in the triggers. For instance, if in our example, triggers 1 and 2 both share the value CRT 15, while trigger 3 uses LCD 19, the constants table would look like:

<table>
<thead>
<tr>
<th>TrigIDs</th>
<th>Const1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,2</td>
<td>CRT 15</td>
</tr>
<tr>
<td>3</td>
<td>LCD 19</td>
</tr>
</tbody>
</table>

Given the constants table, the standard grouping technique [34, 67] is to directly convert the selection condition with constant(s) into a join with the constants table, as shown in Figure 3.17 for our example. In this way, multiple individual selections are converted into a single join, and are hence more efficient.

However, this direct replacement of a select with a join does not work for complex nested conditions. To see why, suppose the WHERE condition in our example is
modified to be of the form \( \text{count}(\text{Select}_{\text{price}<x}(\text{NEW\_ROW\_vendor})) \geq y \) (i.e., the new row contains at least \( y \) vendors who sell an item for less than \( x \); here \( x \) and \( y \) are different constants for different triggers). In this case, the condition contains a selection (\( \text{price} < x \)) nested under a \( \text{nest}() \) nested under another selection (\( \geq y \)). Simply replacing a selection (such as \( \text{price} < x \)) with a corresponding join would be incorrect because this would change the output cardinality of the operator (due to the join with the \text{constants} table, where multiple triggers could be fired); this would in turn change the \( \text{nest}() \) result, thereby producing wrong results.

To address this issue, we propose a simple yet powerful approach that works for arbitrarily complex nested selections. The basic idea is to use the \text{constants} table to set up a \text{correlation} in the \( G_{\text{params}} \) graph to produce a \( G_{\text{grouped}} \) graph, as shown in Figure 3.18. Conceptually, this means that the \( G_{\text{params}} \) graph is evaluated once for each row in the \text{constants} table (i.e., for each unique set of constants). While this will certainly produce the correct results, it is likely to be inefficient because we still do selections one by one for each unique set of constants. However, the key idea now is to decorrelate this graph using query rewrite techniques developed for SQL [105] and XML [106] queries. Decorrelation converts correlated selections to joins [105, 106] (as we desire) and also preserves the correct semantics of the graph by adding appropriate non-nesting columns to \text{nest} operators so that nested selections are handled correctly.

### 3.4.2 Trigger pushdown optimizations

The final step is to generate a SQL trigger that, when activated, produces the output of the decorrelated \( G_{\text{grouped}} \) graph. In generating this SQL trigger, we leverage techniques developed for publishing relational data as nested XML. Specifically, we apply selection/join pushdown [106] on \( G_{\text{grouped}} \) to generate a single sorted outer union [106] SQL
query whose results can be nested in constant space to produce the nested output; this query becomes the body of the SQL trigger generated.

In addition, we apply an important optimization to avoid directly computing the contents of \( T_{old} \) (the pre-update version of \( T \)), which can be expensive since it is not directly made available by the relational database system. For instance, in our example trigger, since only \( NEW\_ROW \) is returned to the user and \( T_{old} \) is only required to compute an aggregate, we would like to produce the aggregate using only \( T \) and the transition tables, rather than materializing \( T_{old} \). Note that this approach is exactly the inverse of the incremental view maintenance problem, which computes new aggregates from old values. Consequently, by switching the role of old values and new values, we can directly use existing incremental view maintenance techniques [97] to compute aggregates on \( T_{old} \) using just \( T \) and the transition tables.

The SQL trigger generated for our running example of an UPDATE on vendor is shown in Figure 3.19 (formatted to be more human-readable). Since the view is injective, we do not have to explicitly produce \( OLD\_ROW \) for comparing with \( NEW\_ROW \) (although we still need to compute the aggregate value on \( T_{old} \) to ensure that \( OLD\_ROW \) appeared in the view before the update). The trigger first finds the affected keys by taking a union of the product names associated with rows in the pruned transition tables (lines 5-14). The trigger then computes the number of vendors for each affected product after the update (lines 15-19), and selects only those with more than one vendor as potential \( NEW\_ROW \)s (lines 21-22). Note that vendors are only computed for affected products by using regular query rewrite techniques to push down the join on affected keys [94, 106]. The trigger then computes the number of vendors for each affected product before the update by using the corresponding values after the update and the pruned transition tables (lines 23-36); also note that the selection on the \( OLD\_ROW \)
name is transformed into a join due to trigger grouping. Finally, the action parameters are produced using a sorted outer union (lines 44-59). Since multiple triggers can be fired for the same update, the list of fired triggers is computed as the final leg of the sorted outer union (lines 53-55).

We also investigated the use of relational materialized views to optimize the performance of nested triggers. We considered subqueries for materialization when they: (a) contained distributive aggregates (such as \texttt{count(*)}); (b) could be incrementally maintained by the relational database (which imposes other restrictions such as not allowing nested predicates or \texttt{having} clauses); and (c) did not contain transition tables. The materialized views were then substituted for the subquery in the trigger. For our example trigger (Figure 3.19), the subquery chosen for materialization was \texttt{ProductCount} (lines 15-19), without the join to \texttt{AffectedKeys}. Ideally, we would have materialized only the subset where \texttt{numVendors} \geq 2 (i.e. \texttt{MultiVendorProduct} without the join to \texttt{AffectedKeys}), but such nested predicates are disallowed in materialized views. Due to this restriction, the materialized view needs to incrementally maintain more rows (including those that do not satisfy the nested predicate), which is one explanation for the surprisingly poor performance of this optimization (Section 3.5.2).

### 3.5 Experimental Evaluation

We have developed and evaluated a prototype of the proposed techniques in the context of Quark XML middleware system [16]. Quark uses an internal algebra called XQGM, which extends the nested relational algebra with XML tags and thus produces XML elements instead of nested relations. Therefore, in this section, we use the term \textit{element} in place of \textit{row} to signify an XML element.

For the experimental evaluation, we considered two metrics (1) the \textit{compile time} for
CREATE TRIGGER sqlTrigger AFTER UPDATE ON VENDOR
REFERENCING OLD_TABLE AS DELETED, NEW_TABLE AS INSERTED
FOR EACH STATEMENT

WITH PrunedIns(pid, vid, price) AS {
    SELECT * FROM INSERTED EXCEPT SELECT * FROM DELETED
},
PrunedDel(pid, vid, price) AS {
    SELECT * FROM DELETED EXCEPT SELECT * FROM INSERTED
},
AffectedKeys (name) AS {
    SELECT P.name FROM product AS P, PrunedIns AS V WHERE P.pid = V.pid
    UNION
    SELECT P.name FROM product AS P, PrunedDel AS V WHERE P.pid = V.pid,
    ProductCount (name, numVendors) AS {
        SELECT P.name, COUNT(*) AS numVendors
        FROM AffectedKeys AS C, product AS P, vendor AS V
        WHERE P.name = C.name AND P.pid = V.pid
    GROUP BY P.name,
    MultiVendorProduct (name) AS {
        SELECT name FROM ProductCount WHERE numVendors >= 2)
    UNION
    MultiVendorProduct_old (name) AS {
        SELECT name FROM (SELECT DISTINCT name, numVendors FROM ProductCount PC, Constants1 C
        WHERE PC.name = C.Const1
        UNION ALL
        SELECT DISTINCT name, numVendors FROM deltaCount DC, Constants1 C
        WHERE DC.name = C.Const1
        ) AS T(name, numVendors)
    GROUP BY T.name
    HAVING SUM(T.numVendors) >= 2)
},
ProductInfo (pid, name) AS {
    SELECT P.pid, P.name
    FROM Product AS P, MultiVendorProduct AS MVP,
    MultiVendorProduct_old AS MVP_old
    WHERE MVP.name = MVP_old.name AND MVP.name = P.name,
    outerUnion(type, pname, triggerIds, vid, price) AS {
        -- Produce the product information
        SELECT 1, PI.name, NULL, NULL FROM ProductInfo AS PI
    UNION ALL
    -- Produce the vendor information
    SELECT 2, PI.name, NULL, vid, price FROM Vendor AS V, ProductInfo AS PI
    WHERE V.pid = PI.pid
    UNION ALL
    -- Produce the trigger information
    SELECT 3, PI.name, C.TrigIDs, NULL, NULL FROM Constants AS C,
    ProductInfo AS PI
    WHERE C.value = PI.name,
    SELECT type, pname, triggerIds, vid
    FROM outerUnion
    ORDER BY type, pname, triggerIds, vid
}

Figure 3.19: The generated SQL trigger.
Table 3.3: Experimental parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values (default in bold)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hierarchy depth</td>
<td>2, 3, 4, 5</td>
</tr>
<tr>
<td># leaf tuples (×1000)</td>
<td>32, 64, 128, 256, 512, 1024</td>
</tr>
<tr>
<td># leaf tuples/element</td>
<td>16, 32, 64, 128, 256</td>
</tr>
<tr>
<td># triggers</td>
<td>1, . . ., 10,000, . . ., 100,000</td>
</tr>
<tr>
<td># updated element</td>
<td>1, 20, 40, 60, 100</td>
</tr>
<tr>
<td># fired triggers/updated element</td>
<td>1, 10, 100, 1000, 10000</td>
</tr>
</tbody>
</table>

a trigger, which is the time to manipulate the intermediate graphs and produce the final SQL trigger, and (2) the run time, which is the overhead of evaluating the generated SQL trigger(s) on an update to the underlying base table(s). The compile time is fairly small (on the order of a hundred milliseconds even for a complex view) and is only expended once during the creation of the trigger. Therefore, we focus on the run time performance in the experiments.

3.5.1 Experimental setup

We used two data sets for the experiments. The first data set was a synthetic data set that we generated for this purpose, where we could vary various parameters such as the depth of nesting, number of triggers fired etc. We performed most of our experiments using this data set. The second data set was the DBLP data set; we used this data set because it has real data with some nesting (papers with nested authors). For the most part of this section, we focus on the synthetic data set. We summarize the results obtained using the real data set (which has similar performance to the synthetic data set) in Section 3.5.2.

The parameters of our experimental setup are given in Table 3.3. Hierarchy depth specifies the depth of the relational schema. For depth 2, we use the product/vendor schema and nested view described earlier. For deeper views, we add additional “ancestor” tables above product, so that each child table has a foreign key column referencing
Table 3.4: Evaluated approaches

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>UNGROUPED</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>UNGROUPED-MV</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>UNGROUPED-AOPT</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>UNGROUPED-AOPT-MV</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>UNGROUPED-EOPT</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>UNGROUPED-EOPT-MV</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>UNGROUPED-EOPT-AOPT</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>UNGROUPED-EOPT-AOPT-MV</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>GROUPED</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>GROUPED-MV</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>GROUPED-AOPT</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>GROUPED-AOPT-MV</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>GROUPED-EOPT</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>GROUPED-EOPT-MV</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>GROUPED-EOPT-AOPT</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>GROUPED-EOPT-AOPT-MV</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Figure 3.20: Varying the number of triggers.
its parent’s primary key, and the nested view contains children nested inside of parents. The # of leaf tuples is the number of rows in the leaf (vendor) table. # leaf tuples/element is the number of leaf tuples per top-level XML element produced by the view; this measures the size of OLD_ROW and NEW_ROW. # triggers specifies the number of structurally similar XML triggers in the system, and # updated elements specifies the number of XML elements that are updated for each relational update. Finally, we also vary the number of fired triggers per updated element. In all cases, the trigger over views was placed on the top-level rows in the view, and the count(···) ≥ 2 predicate remained on the lowest level (vendors). We defined the actions of the triggers to insert the entire NEW_ROW into a temporary table.

We evaluated sixteen alternative implementations to evaluate the various aspects of our approach (Table 3.4). We categorize them based on the optimization techniques used. Grouping refers to grouping structurally similar triggers (Section 3.4.1); Ineq
check refers to the inequality check optimization for injective views (Section 3.3.4); 
Agg opt refers to optimizing aggregate computation on $T_{old}$; and MV refers to utilizing 
materialized views in triggers (Section 3.4.2).

Our experiments were performed on a Linux system with a 933MHz PIII processor 
and 1GB of main memory, running IBM DB2 8.1. We defined primary keys for 
all the relational tables and built appropriate indices on the key columns and other join 
columns. Unless otherwise specified, for each experiment, we varied one of the parameters 
in Table 3.3 and used default values for the rest (the default values are in bold). The 
run time was averaged over 100 independent updates to the vendor table using a cold cache.

### 3.5.2 Performance results

**Varying # Triggers**

Figure 3.20 shows the performance of the different approaches when we vary the number of triggers. For this experiment alone, we set # fired triggers/updated element to be 1 instead of its default value of 100, since the default value is not applicable when # triggers is small. As shown, UNGROUPED-EOPT-AOPT does not scale well because it does not benefit from shared computation across triggers. Other UNGROUPED approaches (not shown) performed even more poorly because they do not take advantage of the different optimizations. In contrast, all the GROUPED approaches scale gracefully due to the grouping optimizations (note the log scale in the x-axis). This suggests that we can successfully employ existing grouping techniques for triggers over nested views.

GROUPED-EOPT and GROUPED-AOPT provide a 25-35% improvement over GROUPED due to our aggregation optimization. Interestingly, these optimizations provide an additive benefit, and GROUPED-EOPT-AOPT provides a 50% improvement in performance
over GROUPED. Surprisingly, GROUPED-EOPT-AOPT-MV shows a 25% performance degradation over GROUPED-EOPT-AOPT indicating that there is a performance degradation due to using relational materialized views. There are two reasons for this performance degradation: (1) the overhead of maintaining the view on every relational update, and (2) lack of support for nested predicates, which increases the number of tuples that need to be incrementally maintained.

Varying Hierarchy depth

Figure 3.21 shows the effect of varying the hierarchy depth (for this and subsequent experiments, we do not consider UNGROUPED approaches due to their bad scalability properties). The hierarchy depth is defined as the depth of the relational schema (i.e. the number of tables to join), which usually translates to a much deeper nesting in nested XML views. For instance, the example view (Figure 2.7) could be written so that <price> is nested under <vid>; while this intuitively increases the depth of the nested view, it does not affect the trigger depth: the resulting SQL will still execute the same number of joins. Thus, to characterize performance, we use the hierarchy depth instead of the depth of the resulting nested views.

As shown, the run time of all the approaches increases approximately linearly with the hierarchy depth. This is because, as the depth increases, the relational trigger must evaluate more joins to recreate the hierarchy. Further, the size of the produced result also increases because the number of intermediate nodes grows larger (even though the number of leaf nodes remains constant). In particular, GROUPED-AOPT scales gracefully and indicates that we can get good performance for XML triggers even for deeply-nested views. The performance of approaches using materialized views is less scalable because the overhead of maintaining the materialized view increases with the number
of joins. For the same reason we observe that approaches not using materialized views consistently outperforms their respective ones using materialized views. For the rest of the experiments, we only show the results for non-MV approaches.

**Varying # fired triggers/updated element**

Figure 3.22 shows the effect of varying the number of fired triggers per updated row in the nested view. As shown, the GROUPED approaches scale gracefully and linearly with the number of fired triggers, such that the time to perform a relational update is only between 150-200 milliseconds even when up to 10,000 top-level triggers are activated for each relational update. Again, as with varying # triggers, our different optimizations provide up to a 30% improvement in performance when compared to not applying the optimizations.

**Varying # updated elements**

Figure 3.23 shows the effect of varying the number of rows in the view that are affected by a single relational update. As shown, all approaches have the same relative performance – the running time grows slowly when the number of produced nodes increases.
Figure 3.24: Varying # leaf tuples/element.

However, the benefits of the injective view optimization (GROUPED-EOPT and GROUPED-EOPT-AOPT) are striking because it avoids having to compute unnecessary OLD_ROWS, and this benefit increases as the number of updated rows increases. Thus, in this case, GROUPED-EOPT-AOPT provides approximately a 60% performance gain over GROUPED.

Varying # leaf tuples per element

Figure 3.24 shows the effect of varying the number of leaf tuples per element in the view. All optimized approaches have the same relative performance, and there is only a small increase in runtime as the parameter value increases. This increase is primarily due to the fact that the OuterUnion intermediate result grows as OLD_ROW and NEW_ROW become larger.

Varying # leaf tuples

We vary the data size by varying the number of leaf tuples; the result is shown in Figure 3.25. All optimized approaches scale gracefully when the data size increases. This is because, although the total number of leaf tuples increases, the number of leaf nodes
in the affected rows remains the same. This graph shows that our system indeed benefits from not materializing the entire nested view, so that we only need to compute a small fraction of leaf nodes.

**Performance using DBLP data**

We also evaluated performance the performance of our proposed techniques on the DBLP data set. Figure 3.26 shows the basic structure of elements in DBLP. The DBLP data set contains about 725000 articles, and the size of the original DBLP XML file is about 300MB. We first shredded this file into two relational tables, *articles* and *authors*. The *articles* table contains all types of articles in DBLP, including inproceedings, proceedings and so on. We then create a nested view which contains articles with at least two co-authors. The trigger is placed on the top-level articles. This setting is conceptually the same as our example catalog view with the hierarchy depth 2.

Figure 3.27 shows the performance results obtained by varying the number of triggers. As shown, the update time in the presence of 10000 triggers for GROUPED-EOPT-AOPT is about 50ms, which indicates that our trigger processing architecture is scalable for the DBLP data set. Further, the relative performance of the different alternatives is similar to the performance obtained using the synthetic data set – all grouped approaches
scale gracefully when the number of triggers increases, and our optimizations result in significant benefits.

3.6 Correctness of CreateAKGraph

In this section we prove the correctness of the algorithm CreateAKGraph. Central to our proof is the correctness of the affected-keys algorithm, CreateAKGraph (Figure 3.5).

Correctness of CreateAKGraph

To show the correctness of CreateAKGraph, we first formally define some terminology. In the following, we use $R(T, D)$ to denote the contents of table $T$ in database state $D$. (In other words, $R(T, D) = R(o, D)$ where $o$ is the operator Table$(T)$.)
Definition 3.6.1 (Valid transition tables). For any given single-table database transition $D \xrightarrow{T} D'$, $(\nabla T, \Delta T)$ is a valid pair of transition tables iff

$$\nabla T \subseteq R(T, D), \quad \Delta T \subseteq R(T, D'),$$

$$\nabla T \supseteq \{x | x \in R(T, D) \land x \notin R(T, D')\},$$

$$\Delta T \supseteq \{x | x \in R(T, D') \land x \notin R(T, D)\},$$

and $(R(T, D) - \nabla T) = R(T, D') - \Delta T$.

Figure 3.28 illustrates this definition.

Definition 3.6.2 (View Trigger Post-Update). A row $r$ is said to be post-updated in view $G$ by relational transition $D \xrightarrow{T} D'$ iff $r \in R(o_G, D')$, and $\exists r'(r' \in R(o_G, D) \land v_{e_k_{o_G}}(r) = v_{e_k_{o_G}}(r') \land r \neq r')$.

We say that a row $r$ is pre-updated if $r$ is updated as per definition 3.2.2.

We now prove the correctness of CreateAKGraph. We first note that if $(O', K) = \text{CreateAKGraph}(O, T, dT)$, then the set of columns in $O$ after the invocation of CreateAKGraph is a superset of $K$ (by line 45 of the algorithm). We can then prove the following lemma.
Lemma 3.6.3 (Correctness of CreateAKGraph). Given a view graph $G$, a relational table $T$, and a database transition $D_1 \xrightarrow{\tau} D_2$, let $(O', K) = \text{CreateAKGraph}(o_G, o_{G, old}, T)$. Then for all rows $x$,

(a) If $x$ is inserted/post-updated in $G$ by $D_1 \xrightarrow{\tau} D_2$, then $x \in R(o_G, D_2)$ and $v_K(x) \in \pi_K(R(O', D_2))$.

(b) If $x$ is deleted/pre-updated in $G$ by $D_1 \xrightarrow{\tau} D_2$, then $x \in R(o_{G, old}, D_2)$ and $v_K(x) \in \pi_K(R(O', D_2))$.

Proof. First, for case (a), by definitions 3.2.3 and 3.6.2, if a row $x$ is inserted/post-updated, then $x \in R(o_G, D_2)$. For case (b), by definitions 3.2.4 and 3.2.2, if a row $x$ is deleted/pre-updated, then $x \in R(o_{G, old}, D_2)$. Then by the definition of $G_{old}$ we know that $R(o_G, D_1) = R(o_{G, old}, D_2)$. It follows that $x \in R(o_{G, old}, D_2)$.

Therefore we only need to show the following:

(a') If $x$ is inserted/post-updated in $G$ by $D_1 \xrightarrow{\tau} D_2$, then $v_K(x) \in \pi_K(R(O', D_2))$.

(b') If $x$ is deleted/pre-updated in $G$ by $D_1 \xrightarrow{\tau} D_2$, then $v_K(x) \in \pi_K(R(O', D_1))$.

We now prove it by induction on the depth of $G$.

Base case: depth = 1.

In this case, the view graph only consists of a single operator, Table$(X)$, for some relational table $X$.

Suppose $T \neq X$. Since we stipulated that a database transition $D_1 \xrightarrow{\tau} D_2$ occurred, $D_1$ and $D_2$ are identical states except for the contents of table $T$. Therefore, since $R$(Table$(X)$, $D_1) = R$(Table$(X)$, $D_2)$, there does not exist a row $x$ such that $x$ is inserted, deleted, or updated, so the lemma is vacuously true.

On the other hand, suppose that $T = X$. Then $(O', K) = (\text{Union}(\text{Project}_K(\text{Table}(\Delta T)),$ $\text{Project}_K(\text{Table}(\nabla T))), T.\text{key})$. If row $x$ is inserted or post-updated
in $G$, by the definition of transition tables, $x \in R(\text{Table}(\triangle T), D_2)$ and therefore $v_K(x) \in R(\pi_K(\text{Table}(\triangle T), D_2))$. By the semantics of $\text{Union}$ operator, it follows that $v_K(x) \in \pi_K(R(O', D_2))$. Similarly, if $x$ is deleted/pre-updated in $G$ by $D_1 \xrightarrow{x} D_2$, then $v_K(x) \in \pi_K(R(O', D_2))$.

Thus, the base case holds.

**Induction Hypothesis:** For a graph $H$ of depth $\leq k$, suppose Lemma 3.6.3 holds.

We will now show that Lemma 3.6.3 holds for a graph $G$ of depth $k + 1$. There are six cases, one for each type of operator except for $\text{Table}$ (which can only occur at the leaf level of the graph).

**Case 1:** $o_G$ is a Nest operator.

This case is handled by lines 10-17 of the algorithm.

The algorithm begins by invoking $\text{CreateAKGraph}$ on the input operator $I$, and returns $(I', K')$. First, if $I' = \emptyset$, then there are no rows in $I$ which were affected as a result of the database transition. Since Nest operator is deterministic, there does not exist a row $x$ in $O$ such that $x$ is inserted, deleted, or updated. In this case, we simply return $\emptyset$ and the lemma is vacuously true.

If $I' \neq \emptyset$, the algorithm creates two $\text{Join}$ operators: $J_{new}$ joining $I$ with $I'$, and $J_{old}$ joining $I_{old}$ with $I'$, and then creates a $\text{Union}$ operator $U$: $\text{Union}(J_{new}, J_{old})$. It finally returns a new Nest operator $O'$ on $U$ which merely projects out the values of the non-nesting columns of $o_G$.

Consider a row $x$ that is inserted in $G$ by $D_1 \xrightarrow{x} D_2$. By the semantics of Nest operator, we know that there exists a row $y$ such that $v_K(y) = v_K(x)$ and $y$ was inserted/post-updated in $I$ by $D_1 \xrightarrow{x} D_2$. By the induction hypothesis, we know that $y \in R(I, D_2)$ and $v_K'(y) \in \pi_K'(R(I', D_2))$. Therefore $y \in R(J_{new}, D_2)$, and by the
semantics of \textbf{Union} operator, \( y \in R(U, D_2) \). Then by the semantics of \textbf{Nest} operator, we know that \( v_K(y) \in \pi_K(R(O', D_2)) \). Finally, since \( v_K(y) = v_K(x) \), we can infer that \( v_K(x) \in \pi_K(R(O', D_2)) \).

Next, consider a row \( x \) that is \textit{deleted} in \( G \) by \( D_1 \xrightarrow{T} D_2 \). We thus know that \( x \in R(o_{G,old}, D_2) \). Therefore by the semantics of the \textbf{Nest} operator, there exists a row \( y \) such that \( y \) was deleted/pre-updated in \( I \) by \( D_1 \xrightarrow{T} D_2 \) and \( v_K(y) \). By the induction hypothesis (b), we know that \( y \in R(I_{old}, D_2) \) and \( v_K(y) \in \pi_K(R(I', D_2)) \). Therefore by the semantics of \textbf{Join} operator, \( y \in R(J_{old}, D_2) \), and by the semantics of \textbf{Union} operator, \( y \in R(U, D_2) \). Then by the semantics of \textbf{Nest} operator, we know that \( v_K(y) \in \pi_K(R(O', D_2)) \). Since \( v_K(x) = v_K(y) \), we can infer that \( v_K(x) \in \pi_K(R(O', D_2)) \).

Now consider a row \( x \) that is \textit{post-updated} in \( G \) by \( D_1 \xrightarrow{T} D_2 \). By the semantics of \textbf{Nest} operator, this could be caused by insertion, deletion, or post-updates to its inputs. For example, \texttt{count} is affected by both insertion and deletion. If it is caused by insertion or post-updated to its inputs, then by the semantics of \textbf{Nest} operator, there exists a row \( y \) such that \( v_K(y) = v_K(x) \) and \( y \) was inserted/post-updates in \( I \) by \( D_1 \xrightarrow{T} D_2 \). By the same argument as for in the case where \( x \) is inserted in \( G \), we can infer that \( v_K(x) \in \pi_K(R(O', D_2)) \). Otherwise, if it is caused by deletion to its inputs, then by the semantics of the \textbf{Nest} operator, there exists a row \( y \) such that \( y \) was deleted in \( I \) by \( D_1 \xrightarrow{T} D_2 \) and \( v_K(y) \). By the same argument as in the case where \( x \) is deleted in \( G \), we can infer that \( v_K(x) \in \pi_K(R(O', D_2)) \).

The proof for the case where a row \( x \) is pre-updated is similar to when it is post-updated.

\textbf{Case 2:} \( o_G \) is a Select operator.

This case is handled by lines 18-19 of the algorithm. The algorithm first invokes \texttt{CreateAKGraph} on the input \( I \) and returns \((I', K')\). It then simply returns \((O', K) = \)
(I, K').

First consider a row x that is inserted/post-updated in G by $D_1 \xrightarrow{T} D_2$. By the semantics of Select operator, there exists a row y such that y was inserted/post-updated in I by $D_1 \xrightarrow{T} D_2$ and $x = y$. By the induction hypothesis, we know that $y \in R(I, D_2)$ and $v_{K'}(y) \in \pi_{K'}(R(I', D_2))$. Since $(O', K) = (I, K')$ and we thus have $v_K(x) \in \pi_K(R(O', D_2))$.

Next consider a row x that is deleted/pre-updated in G by $D_1 \xrightarrow{T} D_2$. By the semantics of Select operator, there exists a row y' such that $y'$ is deleted/pre-updated in I by $D_1 \xrightarrow{T} D_2$ and $x = y'$. By the induction hypothesis, we know that $y' \in R(I', D_2)$ and $v_{K'}(y') \in \pi_{K'}(R(I', D_2))$. Since $(O', K) = (I, K')$ and we thus have $v_K(x) \in \pi_K(R(O', D_2))$.

Case 3: $o_G$ is a Project operator.

This case is handled by lines 18-19 of the algorithm. The algorithm first invokes CreateAKGraph on the input I and returns $(I', K')$. It then simply returns $(O', K) = (I, K')$.

First consider a row x that is inserted/post-affected in G by $D_1 \xrightarrow{T} D_2$. By the semantics of the Project operator, there exists a row y such that y was inserted/post-updated in I by $D_1 \xrightarrow{T} D_2$ and $v_{Co_G}(y) = x$. By the induction hypothesis, we know that $y \in R(I, D_2)$ and $v_K(y) \in \pi_K(R(I', D_2))$. Since $(O', K) = (I', K')$ and $K \subseteq \bar{Co}_G$, we can infer that $v_K(y) = v_K(x)$ and therefore $v_K(x) \in \pi_K(R(O', D_2))$.

Next consider a row x that is deleted/pre-affected in G by $D_1 \xrightarrow{T} D_2$. By the semantics of the Project operator, there exists a row y such that $v_{Co_G,y}(y) = x$ and y was deleted/pre-updated in I by $D_1 \xrightarrow{T} D_2$. By the induction hypothesis, we know that $y \in R(I_{old}, D_2)$ and $v_{K'}(y) \in \pi_{K'}(R(I', D_2))$. Since $(O', K) = (I', K')$ and $K \subseteq \bar{Co}_{G_{old}}$, we can infer that $v_K(y) = v_K(x)$ and therefore $v_K(x) \in \pi_K(R(O', D_2))$. 
Case 4: \( o_G \) is a Join operator.

This case is handled by lines 20-39 of the algorithm. We only consider the case where both inputs have changed as the other cases are specific versions of this case. In this case, CreateAKGraph returns \((I'_0, K_0)\) and \((I'_1, K_1)\) for \( I_0 \) and \( I_1 \), respectively. The algorithm first produces rows inserted or post-updated in \( G \) due to \( I'_0 \) by creating a Join operator \( J_{a1}: \text{Join}_{\text{Join}Prd}(\text{Join}_{K_0}(I'_0, I_0), I_1) \). It then creates \( J_{a2}: \text{Join}_{\text{Join}Prd}(\text{Join}_{K_0}(I'_0, I_0, old), I_{1, old}) \) to produce rows deleted or pre-updated in \( G \) due to \( I'_0 \). It then creates a Union operator \( U_0 \) taking the union of \( J_{a1} \) and \( J_{a2} \). Intuitively \( U_0 \) produces all affected rows due to \( I'_0 \). The algorithm creates a \( U_1 \) for \( I_1 \) similarly, and finally returns \( O' \) taking the union of \( U_0 \) and \( U_1 \).

Consider a row \( x \) that is inserted in \( G \) by \( D_1 \xrightarrow{\tau} D_2 \). We thus know that \( x \in R(o_G, D_2) \), and by the semantics of Join operator, there exist rows \( y \) and \( z \) such that \( y \in R(I_0, D_2) \land y = v_{C_{t_0}}(x) \) and \( z \in R(I_1, D_2) \land z = v_{C_{t_1}}(x) \) and \( y, z \) satisfy the join predicates. Further, since \( x \) is inserted in \( G \), we know that at least one (or both) of \( y \) and \( z \) are inserted or post-updated in \( I_0 \) and \( I_1 \), respectively. Without loss of generality, let us assume that \( y \) was inserted/post-updated in \( I_0 \). By the induction hypothesis, we know that \( y \in R(I_0, D_2) \) and \( v_{K_0}(y) \in \pi_{K_0}(R(I'_0, D_2)) \). Thus, \( y \in R(I_0 \times_{K_0} I_0, D_2) \).

Since, \( z \in R(I_1, D_2) \), we can thus infer that \( x \in R(J_{a1}, D_2) \). By the semantics of Project operator, \( v_K(x) \in \pi_K(R(J_{a1}, D_2)) \), and by the semantics of Union operator, \( v_K(x) \in R(U_0, D_2) \), and finally \( v_K(x) \in R(O', D_2) \).

Next consider a row \( x \) that is deleted in \( G \) by \( D_1 \xrightarrow{\tau} D_2 \). We thus know that \( x \in R(o_G, D_2) \), and by the semantics of Join operator, there exist rows \( y \) and \( z \) such that \( y \in R(I_{0, old}, D_2) \land y = v_{C_{t_{0, old}}}(x) \) and \( z \in R(I_{1, old}, D_2) \land z = v_{C_{t_{1, old}}}(x) \) and \( y, z \) satisfy the join predicates. Further, since \( x \) is deleted in \( G \), we know that at least one (or both) of \( y \) and \( z \) are deleted or pre-updated in \( I_0 \) and \( I_1 \), respectively. Without
loss of generality, let us assume that \( y \) was deleted/pre-updated in \( I_0 \). By the induction hypothesis, we know that \( y \in R(I_{0,old}, D_2) \) and \( v_{K_0}(y) \in \pi_{K_0}(R(I'_0, D_2)) \). Thus, \( y \in R(I'_0 \bowtie K_0 I_{0,old}, D_2) \). Since, \( z \in R(I_{1,old}, D_2) \), we can thus infer that \( x \in R(J_{a2}, D_2) \).

By the semantics of \textbf{Project} operator, \( v_K(x) \in \pi_K(R(J_{a2}, D_2)) \), and by the semantics of \textbf{Union} operator, \( v_K(x) \in R(U_0, D_2) \), and finally \( v_K(x) \in R(O', D_2) \).

Then consider a row \( x \) that is post-updated in \( G \) by \( D_1 \xrightarrow{\tau} D_2 \). We thus know that \( x \in R(o_G, D_2) \), and by the semantics of \textbf{Join} operator, there exist rows \( y \) and \( z \) such that \( y \in R(I_0, D_2) \land y = v_{\tilde{C}_0}(x) \) and \( z \in R(I_1, D_2) \land z = v_{\tilde{C}_1}(x) \) and \( y, z \) satisfy the join predicates. Further, since \( x \) is post-updated in \( G \), we know that at least one (or both) of \( y \) and \( z \) are post-updated in \( I_0 \) and \( I_1 \), respectively. Then by the same argument as in the case that \( x \) is inserted in \( G \), we can infer that \( v_K(x) \in R(O', D_2) \).

Similarly we can prove the lemma for the case that \( x \) is pre-updated in \( G \).

**Case 5:** \( o_G \) is a \textbf{Union} operator.

This case is handled in lines 4-18. We only consider the case where both inputs have changed as the other cases are specific versions of this case. In this case, \textbf{Create-AKGraph} returns \((I'_0, K_0)\) and \((I'_1, K_1)\) for \( I_0 \) and \( I_1 \), respectively. The algorithm first creates two \textbf{Join} operators – \( J_{a1} \) computing the join \( I'_0 \bowtie K_0 I_0 \), and \( J_{a2} \) computing the join \( I'_0 \bowtie K_0 I_{0,old} \), and then creates a \textbf{Union} operator \( U_0 \): \textbf{Union}(\( J_{a1}, J_{a2} \)). Intuitively \( U_0 \) produces rows affected due to \( I'_0 \) only. Next, the algorithm creates a \textbf{Union} \( U_1 \) computing \textbf{Union}(\( \text{Join}_{K_1}(I'_1, I_1), \text{Join}_{K_1}(I'_1, I_{1,old}) \)). Finally the algorithm returns \( O' \) taking \textbf{Union} of \( U_0 \) and \( U_1 \) along with the position column.

Consider a row \( x \) that is inserted/post-updated in \( G \) by \( D_1 \xrightarrow{\tau} D_2 \). Based on the position column of \( x \) and the semantics of the \textbf{Union} operator, we can infer that either: (1) \( x \) is inserted/post-updated in \( I_0 \) by \( D_1 \xrightarrow{\tau} D_2 \), or (2) \( x \) is inserted/post-updated in \( I_1 \) by \( D_1 \xrightarrow{\tau} D_2 \). Without loss of generality, we assume that (1) is true. In this case, by
the induction hypothesis, we know that \( x \in R(I_0, D_2) \land v_{K_0}(x) \in \pi_{K_0}(R(I'_0, D_2)) \). By the semantics of Join operator, we can infer that \( x \in R(I_{a1}, D_2) \). By the semantics of Union operator, we know that \( x \in R(U_0, D_2) \). By the semantics of Project operator, we have that \( v_K(x) \in \pi_K(R(U_0, D_2)) \). Hence, \( v_K(x) \in \pi_K(R(O', D_2)) \).

Next, consider a row \( x \) that is deleted/pre-updated in \( G \) by \( D_1 \xrightarrow{T} D_2 \). Based on the position column of \( x \) and the semantics of the Union operator, we can infer that either: (1) \( x \) is deleted/pre-updated in \( I_0 \) by \( D_1 \xrightarrow{T} D_2 \), or (2) \( x \) is delete/pre-updated in \( I_1 \) by \( D_1 \xrightarrow{T} D_2 \). Without loss of generality, we assume that (1) is true. In this case, by the induction hypothesis, we know that \( x \in R(I_0^{old}, D_2) \land v_{K_0}(x) \in \pi_{K_0}(R(I'_0, D_2)) \). By the semantics of Join operator, we can infer that \( x \in R(I_{a2}, D_2) \). By the semantics of Union operator, we know that \( x \in R(U_0, D_2) \). By the semantics of Project operator, we have that \( v_K(x) \in \pi_K(R(U_0, D_2)) \). Hence, \( v_K(x) \in \pi_K(R(O', D_2)) \).

**Case 6:** \( o_G \) is a SetDiff operator.

The algorithm is shown in lines 20-36 in Figure 3.6. We only consider the case where both inputs have changed as the other cases are specific versions of this case. Assume \( I_0 \) is the left input operator and \( I_1 \) is the right. In this case, CreateAKGraph returns \((I'_0,K_0)\) and \((I'_1,K_1)\) for \( I_0 \) and \( I_1 \), respectively. The algorithm first creates two Join operators – \( J_{a1} \) computing the join \( I'_0 \bowtie_{K_0} I_0 \), and \( J_{a2} \) computing the join \( I'_0 \bowtie_{K_0} I_0^{old} \), and then creates a Union operator \( U_0 \): \( \text{Union}(J_{a1}, J_{a2}) \). Intuitively \( U_0 \) produces rows affected due to \( I'_0 \) only. Next, the algorithm creates a Union \( U_1 \) computing \( \text{Union}(\text{Join}_{K_1}(I'_1, I_1), \text{Join}_{K_1}(I'_1, I_0^{old})) \). Finally the algorithm returns \( O' \) taking Union of \( U_0 \) and \( U_1 \) along with the position column.

Consider a row \( x \) that is inserted in \( G \) by \( D_1 \xrightarrow{T} D_2 \). By the semantics of the SetDiff operator, we can infer that either: (1) \( x \) is inserted/post-updated in \( I_0 \) by \( D_1 \xrightarrow{T} D_2 \), or (2) \( x \) is deleted/pre-updated in \( I_1 \) by \( D_1 \xrightarrow{T} D_2 \). First we assume that (1) is true. In this case,
by the induction hypothesis, we know that \( x \in R(I_0, D_2) \land v_{K_0}(x) \in \pi_{K_0} R(I'_0, D_2) \).

By the semantics of Join operator, we can infer that \( x \in R(J_{a_1}, D_2) \). By the semantics of Union operator, we know that \( x \in R(U_0, D_2) \). By the semantics of Project operator, we have that \( v_{K}(x) \in \pi_{K}(R(U_0, D_2)) \). Hence, \( v_{K}(x) \in \pi_{K}(R(O', D_2)) \). Then we assume that (2) is true. In this case, by the induction hypothesis, we know that \( x \in R(I_{1,old}, D_2) \land v_{K_1}(x) \in \pi_{K_1}(R(I'_1, D_2)) \). By the semantics of Join operator, we can infer that \( x \in R(J_{b_2}, D_2) \) where \( J_{b_2} = \text{Join}_{K_1}(I'_1, I_{1,old}) \). By the semantics of Union operator, we know that \( x \in R(U_1, D_2) \). By the semantics of Project operator, we have that \( v_{K}(x) \in \pi_{K}(R(U_1, D_2)) \). Hence, \( v_{K}(x) \in \pi_{K}(R(O', D_2)) \).

Next consider a row \( x \) that is deleted in \( G \) by \( D_1 \xrightarrow{T} D_2 \). By the semantics of the SetDiff operator, we can infer that either: (1) \( x \) is inserted/post-updated in \( I_1 \) by \( D_1 \xrightarrow{T} D_2 \), or (2) \( x \) is deleted/pre-updated in \( I_0 \) by \( D_1 \xrightarrow{T} D_2 \). Then we can argue similarly to the case that \( x \) is inserted in \( G \), and infer that \( v_{K}(x) \in \pi_{K}(R(O', D_2)) \).

Then consider a row \( x \) that is post-updated in \( G \) by \( D_1 \xrightarrow{T} D_2 \). By the semantics of the SetDiff operator, \( x \) is post-updated in \( I_0 \). Therefore we can also argue very similarly to the case that \( x \) is inserted in \( G \) and infer that \( v_{K}(x) \in \pi_{K}(R(O', D_2)) \).

We can also similarly prove the lemma for the case that \( x \) is pre-updated in \( G \).

\( \square \)

**Proof of Correctness of CreateARGraph**

We now proceed with the proof of the correctness of CreateARGraph. Note our algorithm is applicable to trigger-specifiable views, i.e., every operator must have key columns. Please refer to Figure 3.15 for the text of the algorithm, which is referenced throughout this proof.

**Theorem 3.6.4.** Given an event \( E \), a view graph \( G \), and table \( T \), CreateARGraph\( (E, G, T) \)
produces graph $G_{\text{affected}}$ such that for all valid database transitions $D \xrightarrow{\tau} D'$, $(\text{OLD\_ROW},$ \text{NEW\_ROW}) \in \text{R}(o_{G_{\text{affected}}}, D')$ iff:

(a) $E = \text{UPDATE} \land \text{OLD\_ROW} \in \text{R}(o_G, D) \land \text{NEW\_ROW} \in \text{R}(o_G, D') \land v_{ckoG}(\text{OLD\_ROW}) = v_{ckoG}(\text{NEW\_ROW}) \land \text{OLD\_ROW} \neq \text{NEW\_ROW}$, or

(b) $E = \text{INSERT} \land \text{OLD\_ROW} = \emptyset \land \text{NEW\_ROW} \in \text{R}(o_G, D') \land v_{ckoG}(\text{NEW\_ROW}) = v_{ckoG}(x)$, or

(c) $E = \text{DELETE} \land \text{NEW\_ROW} = \emptyset \land \text{OLD\_ROW} \in \text{R}(o_G, D) \land v_{ckoG}(\text{OLD\_ROW}) = v_{ckoG}(x)$.

Proof. Let $(O', K) = \text{CreateAKGraph}(o_G, o_{G_{\text{old}}}, T)$. We now consider each of the three event types separately, and we prove the theorem in both directions for each.

(a) $E = \text{UPDATE}$.

Suppose $E = \text{UPDATE}$, and there exist $\text{OLD\_ROW}$ and $\text{NEW\_ROW}$ such that $\text{OLD\_ROW} \in \text{R}(o_G, D) \land \text{NEW\_ROW} \in \text{R}(o_G, D') \land v_{ckoG}(\text{OLD\_ROW}) = v_{ckoG}(\text{NEW\_ROW}) \land \text{OLD\_ROW} \neq \text{NEW\_ROW}$. This implies that $\text{NEW\_ROW}$ is post-updated in $G$ by $D \xrightarrow{\tau} D'$, and $\text{OLD\_ROW}$ is pre-updated in $G$ by $D \xrightarrow{\tau} D'$. By Lemma 3.6.3, we can infer that $v_K(\text{NEW\_ROW}) \in \pi_K(\text{R}(O', D'))$. We also have $\text{NEW\_ROW} \in \text{R}(o_G, D')$ and therefore $\text{NEW\_ROW} \in \text{R}(O_{\text{new}}, D')$ (line 3: $O_{\text{new}} = \text{Join}_K(O', o_G)$). Similarly, since $\text{OLD\_ROW}$ is pre-updated in $G$ by $D \xrightarrow{\tau} D'$, we can conclude that $\text{OLD\_ROW} \in \text{R}(o_{G_{\text{old}}}, D')$ (line 4: $O_{\text{old}} = \text{Join}_K(O', o_{G_{\text{old}}})$). Further, since we stipulated that $v_{ckoG}(\text{OLD\_ROW}) = v_{ckoG}(\text{NEW\_ROW}) \land \text{OLD\_ROW} \neq \text{NEW\_ROW}$, the inner join in line 8 will produce the pair $(\text{OLD\_ROW}, \text{NEW\_ROW})$, and Select in line 9 will not filter it out. Therefore $(\text{OLD\_ROW}, \text{NEW\_ROW}) \in \text{R}(o_{G_{\text{affected}}}, D')$

Conversely, suppose $E = \text{UPDATE}$, and there exists $(\text{OLD\_ROW}, \text{NEW\_ROW}) \in \text{R}(O_{G_{\text{affected}}}, D')$. Then, this must have been returned in line 9, so we can infer that
OLD_ROW \neq NEW_ROW. Furthermore, OLD_ROW and NEW_ROW come from \(O_{old}\) and \(O_{new}\), respectively, joined on their respective keys; therefore, since the key of both \(O_{old}\) (line 4) and \(O_{new}\) (line 3) is the same as the key of \(G\), we have that \(v_{ckoG}(OLD_ROW) = v_{ckoG}(NEW_ROW)\). Finally, since \(R(O_{new}, D') \subseteq R(o_G, D')\), we can conclude that \(NEW_ROW \in R(o_G, D')\). We can similarly conclude that \(OLD_ROW \in R(o_G, D')\) because \(R(O_{old}, D') \subseteq R(o_G, D')\).

\(b)\) \(E = \text{INSERT}\).

Next, suppose that \(E = \text{INSERT}\), and there exist OLD_ROW and NEW_ROW such that \(OLD_ROW = \emptyset \land NEW_ROW \in R(o_G, D') \land \exists x(x \in R(o_G, D) \land v_{ckoG}(NEW_ROW) = v_{ckoG}(x))\). This indicates that \(NEW_ROW\) is inserted by \(D \xrightarrow{\tau} D'\). By Lemma 3.6.3, we can infer that \(v_K(NEW_ROW) \in \pi_K(R(O', D'))\). Since we already know that \(NEW_ROW \in R(o_G, D')\), it follows that \(NEW_ROW \in R(O_{new}, D')\) (line 3: \(O_{new} \leftarrow \text{Join}_K(O', o_G)\)). Further, since \(\exists x(x \in R(o_G, D) \land v_{ckoG}(NEW_ROW) = v_{ckoG}(x))\), and \(R(o_{old}, D') \subseteq R(o_G, D')\), it follows that there is no row \(y \in R(O_{old}, D')\) such that \(v_{ckoG}(y) = v_{ckoG}(NEW_ROW)\). Therefore, the LeftAntiJoin (line 12) will produce \((\emptyset, NEW_ROW)\).

Conversely, suppose \(E = \text{INSERT}\), and there exists \((OLD_ROW, NEW_ROW) \in R(O_{affected}, D')\). Then, this must have been returned in line 12, so we can infer that \(OLD_ROW = \emptyset\). Furthermore, OLD_ROW and NEW_ROW come from \(O_{old}\) and \(O_{new}\), respectively, anti-joined on their respective keys. Therefore, since the key of both \(O_{old}\) (line 4) and \(O_{new}\) (line 3) is the same as the key of \(G\), we have that \(NEW_ROW \in R(o_G, D')\), and

\((*) \exists y(y \in R(O_{old}, D') \land v_{ckoG}(y) = v_{ckoG}(NEW_ROW))\).

Then, suppose by contradiction, \(\exists x(x \in R(o_G, D) \land v_{ckoG}(NEW_ROW) = v_{ckoG}(x))\). Let \(r = NEW_ROW\), and by definitions 3.2.2 and 3.2.4, it follows that \(r\) does not change,
was post-updated in \( G \), or \( r \) was deleted in \( G \). First consider that \( r \) does not change. This implies \( r \in R(o_G, D) \), and since \( r \in R(O_{\text{new}}, D') \), it follows that \( r \in R(O_{\text{old}}, D') \).

This is contradictory to (*) Next consider that \( r \) was post-updated in \( G \). In this case, by Definition 3.6.2, \( \exists r' \) such that \( r' \in R(o_G, D) \) and \( v_{ck,r} = v_{ck,r'} \). In other words, \( r' \) was pre-updated in \( G \). Then by Lemma 3.6.3, \( r' \in R(o_G, D) \) and \( v_K(r') \in \pi_K(R(o_G, D)) \). Therefore \( r' \in R(O_{\text{old}}, D') \). This is also contradictory to (*). Finally consider that \( r \) was deleted in \( G \). In this case, by the definition of delete, there are no rows \( t \) such that \( t \in R(o_G, D') \) and \( v_{ck_G}(t) = v_{ck_G}(r) \). This is a contradiction because \( r = NEW\_ROW \) and \( NEW\_ROW \in R(o_G, D') \).

Therefore we have that \( \exists x \)(\( x \in R(o_G, D) \wedge v_{ck_G}(NEW\_ROW) = v_{ck_G}(x) \))

(c) \( E = \text{DELETE} \).

The final case is analogous to (b). Suppose that \( E = \text{DELETE} \), and there exist \( OLD\_ROW \) and \( NEW\_ROW \) such that \( NEW\_ROW = \emptyset \wedge OLD\_ROW \in R(o_G, D) \wedge \exists x \)(\( x \in R(o_G, D') \wedge v_{ck_G}(OLD\_ROW) = v_{ck_G}(x) \)). This indicates that \( OLD\_ROW \)

was deleted in \( G \) by \( [T \rightarrow D][D'] \). By Lemma 3.6.3 we can infer that \( v_K(OLD\_ROW) \in \pi_K(R(O', D')) \). Since we already know that \( OLD\_ROW \in R(o_G, D) \) and \( R(o_G, D) = R(o_G_{\text{old}}, D') \), it follows that \( OLD\_ROW \in R(O_{\text{old}}, D') \). Since \( \exists x \)(\( x \in R(o_G, D') \wedge v_{ck_G}(OLD\_ROW) = v_{ck_G}(x) \)), we know that there is no row \( y \in R(O_{\text{new}}, D') \) such that \( v_{ck_G}(y) = v_{ck_G}(OLD\_ROW) \). Therefore, the RightAntiJoin (line 14) will produce \( (OLD\_ROW, \emptyset) \).

Conversely, suppose \( E = \text{DELETE} \), and there exists \( (OLD\_ROW, NEW\_ROW) \in R(O_{G_{\text{affected}}, D'}) \). Then, this must have been returned in line 14, so we can infer that \( NEW\_ROW = \emptyset \). Furthermore, \( OLD\_ROW \) and \( NEW\_ROW \) come from \( O_{\text{old}} \) and \( O_{\text{new}} \), respectively, anti-joined on their respective keys. Therefore, since the key of both \( O_{\text{old}} \) (line 4) and \( O_{\text{new}} \) (line 3) is the same as the key of \( G \), we have that \( OLD\_ROW \in \)
Then, suppose by contradiction, \( \exists x \mid (x \in R(o_G, D') \land v_{ck_G}(x) = v_{ck_G}(OLD\text{\_ROW})) \). Let \( r = OLD\text{\_ROW} \), and by definitions 3.2.2 and 3.2.3, it follows that \( r \) does not change, \( r \) was pre-updated in \( G \), or \( r \) was inserted in \( G \). First consider that \( r \) does not change. This implies \( r \in R(o_G, D') \), and since \( r \in R(O_{old}, D') \), it follows that \( r \in R(O_{new}, D') \). This is contradictory to (*). Next consider that \( r \) was pre-updated in \( G \). In this case, by Definition 3.2.2, \( \exists r' \) such that \( r' \in R(o_G, D') \) and \( v_{ck_r} = v_{ck_r} \). In other words, \( r' \) was post-updated in \( G \). Then by Lemma 3.6.3, \( r' \in R(o_G, D') \) and \( v_K(r') \in \pi_K(R(o_G, D')) \). Therefore \( r' \in R(O_{new}, D') \). This is contradictory to (**). Finally consider that \( r \) was inserted. In this case, by the definition of insert, there are no rows \( t \) such that \( t \in R(o_G, D) \) and \( v_{ck_G}(t) = v_{ck_G}(r) \). This is also a contradiction because \( r = OLD\text{\_ROW} \) and \( OLD\text{\_ROW} \in R(o_G, D) \).

Therefore we have that \( \exists x \mid (x \in R(o_G, D') \land v_{ck_G}(OLD\text{\_ROW}) = v_{ck_G}(x)) \).

\( \square \)

### 3.7 Correctness of CreateAROpt

In this section, we prove the correctness of CreateAROpt, which is the modified version of CreateARGraph that removes the inequality check (Figure 3.15, line 9) for injective views.

We first formally define pruned transition tables by refining Definition 3.6.1:

**Definition 3.7.1** (Pruned transition tables). For any given single-table database transition \( D \rightarrow D' \), the pruned transition tables \( \Delta T \) and \( \nabla T \) are:

\[
\Delta T = \{ x \mid x \in R(T, D') \land x \notin R(T, D) \},
\]

and
\[ \nabla T = \{ x \mid x \in R(T, D) \land x \notin R(T, D') \} \].

Then we first prove the following lemma.

**Lemma 3.7.2.** Given a relational table \( T \), a trigger-specifiable view graph \( G \) which is injective for columns \( C \) w.r.t. \( T \), and a database transition \( D_1 \xrightarrow{T} D_2 \) with pruned transition tables and \( \text{Table}(T) \notin OP_G \). Then \( \text{CreateAKGraph}(o_G, o_{G, \text{old}}, T) = (\emptyset, \emptyset) \).

**Proof.** We prove Lemma 3.7.2 by induction on the depth of \( G \).

**Base case: depth=1.**

In this case, the view graph only consists of a single operator, \( \text{Table}(X) \), i.e., \( OP_G = \{ \text{Table}(X) \} \). Since \( \text{Table}(T) \notin OP_G, T \neq X \). Therefore by line 8 the algorithm will return \((\emptyset, \emptyset)\).

Thus the base case holds.

**Induction Hypothesis:** For a graph \( H \) of depth \( \leq k \), suppose Lemma 3.7.3 holds.

We will now show that Lemma 3.7.2 holds for a graph \( G \) of depth \( k + 1 \). There are six cases, one for each type of operator except for \( \text{Table} \) (which can only occur at the leaf level of the graph).

**Case 1:** \( o_G \) is a Nest operator.

Let \( I \) be the input operator of \( o_G \). \( \text{Table}(T) \notin OP_G \) implies that \( \text{Table}(T) \notin OP_{G_I} \). By the induction hypothesis, we have that \( \text{CreateAKGraph}(I, I_{\text{old}}, T) = (\emptyset, \emptyset) \). Hence by line 11 the algorithm will return \((\emptyset, \emptyset)\)

**Case 2:** \( o_G \) is a Select or Project operator.

Let \( I \) be the input operator of \( o_G \). \( \text{Table}(T) \notin OP_G \) implies that \( \text{Table}(T) \notin OP_{G_I} \). By the induction hypothesis, we have that \( \text{CreateAKGraph}(I, I_{\text{old}}, T) = (\emptyset, \emptyset) \). Hence by line 19 the algorithm will return \((\emptyset, \emptyset)\)
Case 3: $o_G$ is a Join operator.

Without loss of generality, assuming $o_G$ has two inputs, $I_0$ and $I_1$. $\text{Table}(T) \not\in OPG$ implies that $\text{Table}(T) \not\in OPG_{I_0}$. By the induction hypothesis, we have that $\text{CreateAKGraph}(I_0, I_{0,old}, T) = (\emptyset, \emptyset)$. Similarly we infer that $\text{CreateAKGraph}(I_1, I_{1,old}, T) = (\emptyset, \emptyset)$. Therefore by line 23 the algorithm will return $(\emptyset, \emptyset)$.

Case 4: $o_G$ is a Union or SetDiff operator.

This case is similar to Case 3 and by the same argument, we can infer that $\text{CreateAKGraph}(o_G, o_{G,old}, T) = (\emptyset, \emptyset)$.

Now we prove another lemma for injective views.

Lemma 3.7.3. Given a relational table $T$, a trigger-specifiable view graph $G$ which is injective for columns $C$ w.r.t. $T$, a database transition $D_1 \xrightarrow{\tau} D_2$ with pruned transition tables, and $\text{Table}(T) \in OPG$, then $\forall x \in R(o_G, D_1) \forall y \in R(o_G, D_2), v_C(x) = v_C(y) \Rightarrow (x = y)$.

Proof. We prove Lemma 3.7.3 by induction on the depth of $G$.

Base case: depth=1.

In this case, the view graph only consists of a single operator, $\text{Table}(T)$. Then by the definition of injection, $C$ must be all columns of $T$. Since $v_C(x) = v_C(y)$ it follows that $x = y$.

Induction Hypothesis: For a graph $H$ of depth $\leq k$, suppose Lemma 3.7.3 holds.

We will now show that Lemma 3.7.3 holds for a graph $G$ of depth $k + 1$. There are five cases, one for each type of operator except for $\text{Table}$ (which can only occur at the leaf level of the graph).
**Case 1: \( o_G \) is a Nest operator.**

Since \( T \mapsto C \) and by the definition of transitive injection, we know that \( o_G \) is injective for \( C_I \mapsto C \), where \( C_I \) is the set of columns of the input operator, \( I \), and \( T \mapsto C_I \).

First, by the definition of injection, we know that the set of \( I \)-rows used for computing \( v_C(x) \) and \( v_C(y) \) did not change in the transition. Let \( X = \zeta(x, I, o_G, D_1) \) and \( Y = \zeta(y, I, o_G, D_2) \), then \( \pi_{C_I}(X) = \pi_{C_I}(Y) \). This implies \( \forall x' \in X, \exists y' \in Y, v_{C_I}(x') = v_{C_I}(y') \) and \( \forall y' \in Y, \exists x' \in X, v_{C_I}(y') = v_{C_I}(x') \). Then by the induction hypothesis, we know \( \forall x' \in X, y' \in Y, (v_{C_I}(x') = v_{C_I}(y')) \Rightarrow x' = y' \). Therefore \( X = Y \). By the definition of contributing set and since Nest operator is deterministic w.r.t. to its inputs, we can infer that \( x = y \).

**Case 2: \( o_G \) is a Select or Project operator.**

Since \( T \mapsto C \) and by the definition of transitive injection, we know that \( o_G \) is injective for \( C_I \mapsto C \), where \( C_I \) is the set of columns of the input operator, \( I \), and \( T \mapsto C_I \).

Let \( X = \zeta(x, I, o_G, D_1) \) and \( Y = \zeta(y, I, o_G, D_2) \). By the definition of contributing set, \( X = \{ x \} \) and \( Y = \{ y \} \). Further by the definition of injection and \( v_C(x) = v_C(y) \), we know \( v_{C_I}(x) = v_{C_I}(y) \). Then by the induction hypothesis, we infer that \( x = y \).

**Case 3: \( o_G \) is a Join operator.**

Without loss of generality, assuming \( o_G \) has two inputs, \( I_0 \) and \( I_1 \). Since \( T \mapsto C \) and by the definition of transitive injection, we know that \( o_G \) is injective for \( C_0(C_1) \mapsto C \), where \( C_0(C_1) \) is the set of columns of the input operator, \( I_0(I_1) \), and \( T \mapsto C_0(C_1) \).

By the semantics of Join we know that there are rows \( x_0, x_1 \) such that \( x_0 \in R(I_0, D_1) \land x_1 \in R(I_1, D_1) \land x_0 = v_{C_{I_0}}(x) \land x_1 = v_{C_{I_1}}(x) \). And there are rows \( y_0, y_1 \) such that \( y_0 \in R(I_0, D_2) \land y_1 \in R(I_1, D_2) \land y_0 = v_{C_{I_0}}(y) \land y_1 = v_{C_{I_1}}(y) \).

Let \( X_0 = \zeta(x, I_0, o_G, D_1) \) and \( Y_0 = \zeta(y, I_0, o_G, D_2) \). By the definition of contribut-
ing set, we know that $X_0 = \{x_0\}$ and $Y_0 = \{y_0\}$. Since $v_C(x) = v_C(y)$ and by the definition of injection, we infer that $v_{C_0}(x_0) = v_{C_0}(y_0)$. Then by the induction hypothesis, we know $x_0 = y_0$. Similarly, we can show that $x_1 = y_1$. Therefore by the semantics of Join operator, we know that $x = y$.

**Case 4:** $o_G$ is a Union operator.

Without loss of generality, assuming $o_G$ has two inputs, $I_0$ and $I_1$. Since $T \mapsto C$ and by the definition of transitive injection, we know that $o_G$ is injective for $C_0(C_1) \mapsto C$, where $C_0(C_1)$ is the set of columns of the input operator, $I_0(I_1)$, and $T \mapsto C_0(C_1)$.

By the semantics of Union operator, $y \in R(o_G, D_1)$ implies that $y \in R(I_0, D_1)$ or $y \in R(I_1, D_1)$. Without loss of generality, assuming $y \in R(I_0, D_1)$. Then let $X_0 = \zeta(x, I_0, o_G, D_1)$ and $Y_0 = \zeta(y, I_0, o_G, D_2)$. By the definition of contributing set, we have that $X_0 = \{x\}$. Since $v_C(x) = v_C(y)$ and by the definition of injection, $\pi_{C_0}(X_0) = \pi_{C_0}(Y_0)$. Therefore $Y_0$ must be non-empty and $Y_0 = \{y\}$.

We also infer that $v_{C_0}(x) = v_{C_0}(y)$. By the induction hypothesis, we know that $x = y$.

**Case 5:** $o_G$ is a SetDiff operator.

Assuming $I_0$ is the left input operator and $I_1$ is the right. Since $T \mapsto C$ and by the definition of transitive injection, we know that $o_G$ is injective for $C_0(C_1) \mapsto C$, where $C_0(C_1)$ is the set of columns of the input operator, $I_0(I_1)$, and $T \mapsto C_0(C_1)$.

By the semantics of SetDiff we know that $x \in R(I_0, D_1)$ and $y \in R(I_0, D_2)$. Let $X_0 = \zeta(x, I_0, o_G, D_1)$ and $Y_0 = \zeta(y, I_0, o_G, D_2)$, and it follows that $X_0 = \{x\}$ and $Y_0 = \{y\}$. Since $v_C(x) = v_C(y)$ and by the definition of injection, $v_{C_0}(x) = v_{C_0}(y)$. Finally by the induction hypothesis, we can infer that $x = y$. 

\[\square\]
Given an injective view, we now prove a stronger version of Lemma 3.6.3:

**Lemma 3.7.4.** Given a relational table $T$, a trigger-specifiable view graph $G$ which is injective for columns $C$ w.r.t. $T$, and a database transition $D_1 \xrightarrow{T} D_2$ with pruned transition tables, let $(O', K) = \text{CreateAKGraph} (o_G, o_{G,\text{old}}, T)$. Then $\exists x, y, z (x \in \pi_K(R(O', D_2)) \land y \in R(o_G, D_1) \land z \in R(o_G, D_2) \land v_C(y) = v_C(z) \land v_K(y) = x \land v_K(z) = x)$.

**Proof.** First, if $\text{Table}(T) \not\in OP_G$, then by Lemma 3.7.2, $(O', K) = (\emptyset, \emptyset)$. Lemma 3.7.4 is thus vacuously true.

Now suppose $\text{Table}(T) \in OP_G$, and we prove the lemma by induction on the depth of $G$.

**Base case: depth = 1.**

In this case, the view graph only consists of a single operator, $\text{Table}(T)$. Then $(O', K) = (\text{Union}(\text{Project}_K(\text{Table}(\triangle T)), \text{Project}_K(\text{Table}(\triangledown T))), T.\text{key})$.

Assume there exist rows $y, z$ such that $y \in R(T, D_1) \land z \in R(T, D_2) \land v_C(y) = v_C(z)$, we need to show that there are no rows $x \in R(O', D')$ such that $((v_K(y) = x) \land (v_K(z) = x))$.

First, by Lemma 3.7.3 we know that $y = z$. Therefore $y \in R(T, D_1) \land y \in R(T, D_2)$. Then by definitions of pruned transition tables, $y \not\in \triangle T$ and $y \not\in \triangledown T$. Further, we know that $\triangledown y' \in R(T, D_1) \land v_K(y') = v_K(y)$ because $K$ is the primary key. Since $R(\triangledown T, D_1) \subseteq R(T, D_1)$, we know that $\triangledown y'' \in R(\triangledown T, D_1) \land v_K(y'') = v_K(y)$. Similarly we can show that $\triangle z'' \in R(\triangle T, D_2) \land v_K(z'') = v_K(y)$. Therefore by the semantics of $\text{Union}$ operator, there are no rows $x \in R(O', D')$ such that $((v_K(y) = x) \land (v_K(z) = x))$.

Thus, the base case holds.

**Induction Hypothesis:** For a graph $H$ of depth $\leq k$, suppose Lemma 3.7.4 holds.
We will now show that Lemma 3.7.4 holds for a graph \( G \) of depth \( k + 1 \). There are five cases, one for each type of operator except for Table (which can only occur at the leaf level of the graph).

**Case 1: \( o_G \) is a Nest operator.**

Let \((I', K') = \text{CreateAKGraph}(I, I_{old}, T)\). Also, let \( nc \) be the non-nesting columns of \( o_G \) (note that the set of non-nesting columns defines the key of a Nest operator, i.e., \( K = nc \)).

Since \( T \dashv C \) and Table \((T) \in OP_G \), by the definition of transitive injection, we know that \( o_G \) is injective for \( C_1 \dashv C \), where \( C_i \) is the set of columns of the input operator, \( I \), and \( T \dashv C_i \). Assuming there exist rows \( y, z \) such that \( y \in R(o_G, D_1) \) \( \land z \in R(o_G, D_2) \land v_C(y) = v_C(z) \). Now we want to show that there are no rows \( x \in R(O', D_2) \land v_K(z) = x \land v_K(y) = x \).

First, by the definition of injection, we know that the set of \( I \)-rows used for computing \( v_C(y) \) and \( v_C(z) \) did not change in the transition. Let \( U = \zeta(y, I, o_G, D_1) \) and \( V = \zeta(z, I, o_G, D_2) \), then \( \pi_{C_1}(U) = \pi_{C_1}(V) \). This implies \( \forall u \in U, \exists v \in V, v_{C_1}(u) = v_{C_1}(v) \), and \( \forall v \in V, \exists u \in U, v_{C_1}(u) = v_{C_1}(v) \). By Lemma 3.7.3, \( \forall u \in U, \exists v \in V, u = v \), and \( \forall v \in V, \exists u \in U, v = u \).

Then by the induction hypothesis, \( \forall u \in U, \forall v \in V, v_{C_1}(u) = v_{C_1}(v) \Rightarrow (\exists x' \in R(I', D_2), x' = v_K(u) \land x' = v_K(v)) \). We know \( \forall u \in U, \exists v \in V, u = v \), therefore \( \forall u \in U, \exists x' \in R(I', D_2), x' = v_K(u) \). Therefore \( \forall u \in U, u \notin R(J_{old}, D_2) \).

On the other hand, since \( U \) is the contributing set for \( y \) and \( K \) is the non-nesting column, we know that \( \exists r \in R(I_1, D_1), r \notin U \land v_K(y) = v_K(r) \). Therefore we infer that \( \exists j \in R(J_{old}, D_2), v_K(y) = v_K(j) \).

Similarly we can infer that \( \exists j \in R(J_{new}, D_2), v_K(z) = v_K(j) \).

Since \( y = z \), \( v_K(y) \notin \pi_K R(O', D_2) \). In other words, there are no rows \( x \in \)
Case 2: $o_G$ is a Select operator.

Since $T \rightarrow C$ and Table$(T) \in OP_G$, by the definition of transitive injection, we know that $o_G$ is injective for $C_I \rightarrow C$, where $C_I$ is the set of columns of the input operator, $I$, and $T \rightarrow C_I$.

Assuming there exist rows $y, z$ such that $y \in R(o_G, D_1) \wedge z \in R(o_G, D_2) \wedge v_C(y) = v_C(z)$. Now we want to show that there are no rows $x \in R(O', D_2) \wedge v_K(z) = x \wedge v_K(y) = x$.

Let $Y = \zeta(y, I, o_G, D_1)$ and $Z = \zeta(z, I, o_G, D_2)$. By the definition of contributing set, $Y = \{y\}$ and $Z = \{z\}$. By Lemma 3.7.3 and $v_C(y) = v_C(z)$, we know $y = z$ and $v_{C_I}(y) = v_{C_I}(z)$. Then by the induction hypothesis, we infer that there are no rows $x \in R(I', D_2')$ such that $x = v_{K'}(y) \wedge x = v_{K'}(z)$. Since $(O', K) = (I', K')$, it follows that there are no rows $x \in R(O', D_2)$ such that $x = v_K(y) \wedge x = v_K(z)$.

Case 3: $o_G$ is Project operator.

This is similar to Case 2.

Case 4: $o_G$ is a Join.

Assume the input operators are $I_0$ and $I_1$. We prove the lemma in three sub-cases, based on if Table$(T) \in OP_{G_{I_0}}$ or Table$(T) \in OP_{G_{I_1}}$.

Case (a): Table$(T) \in OP_{G_{I_0}} \wedge Table(T) \not\in OP_{G_{I_1}}$

In this case, by Lemma 3.7.2 $(I_1', K_1) = (\emptyset, \emptyset)$. Therefore the algorithm will simply return $(O', K) = (I_0', K_0)$.

Assuming there exist rows $y, z$ such that $y \in R(o_G, D_1) \wedge z \in R(o_G, D_2) \wedge v_C(y) = v_C(z)$. Now we want to show that there are no rows $x \in R(O', D_2) \wedge v_K(z) = x \wedge v_K(y) = x$. 

$R(O', D_2) \wedge v_K(z) = x \wedge v_K(y) = x$. 

Case 2: $o_G$ is a Select operator.

Since $T \rightarrow C$ and Table$(T) \in OP_G$, by the definition of transitive injection, we know that $o_G$ is injective for $C_I \rightarrow C$, where $C_I$ is the set of columns of the input operator, $I$, and $T \rightarrow C_I$.

Assuming there exist rows $y, z$ such that $y \in R(o_G, D_1) \wedge z \in R(o_G, D_2) \wedge v_C(y) = v_C(z)$. Now we want to show that there are no rows $x \in R(O', D_2) \wedge v_K(z) = x \wedge v_K(y) = x$.

Let $Y = \zeta(y, I, o_G, D_1)$ and $Z = \zeta(z, I, o_G, D_2)$. By the definition of contributing set, $Y = \{y\}$ and $Z = \{z\}$. By Lemma 3.7.3 and $v_C(y) = v_C(z)$, we know $y = z$ and $v_{C_I}(y) = v_{C_I}(z)$. Then by the induction hypothesis, we infer that there are no rows $x \in R(I', D_2)$ such that $x = v_{K'}(y) \wedge x = v_{K'}(z)$. Since $(O', K) = (I', K')$, it follows that there are no rows $x \in R(O', D_2)$ such that $x = v_K(y) \wedge x = v_K(z)$.
By the semantics of `Join` operator, since \( y \in R(o_G, D_1) \), \( \exists y_0 \in R(I_0, D_1), \exists y_1 \in R(I_1, D_1), y_0 = v_{C_{i_0}}(y) \land y_1 = v_{C_{i_1}}(y) \). And since \( z \in R(o_G, D_2) \), \( \exists z_0 \in R(I_0, D_2), \exists z_1 \in R(I_1, D_2), z_0 = v_{C_{i_0}}(z) \land z_1 = v_{C_{i_1}}(z) \).

Further, since \( T \mapsto C \) and `Table(T) \in OP_{G_{i_0}}`, by the definition of transitive injection, we know that \( o_G \) is injective for \( C_0 \mapsto C \), where \( C_0 \) is the set of columns of the input operator, \( I_0 \), and \( T \mapsto C_0 \). By Lemma 3.7.3, we know that \( y = z \). Hence \( y_0 = z_0 \) and \( v_{C_0}(y_0) = v_{C_0}(z_0) \).

Therefore by the induction hypothesis, there are no rows \( x' \in R(I'_0, D_2) \) such that \( v_{K_0}(y_0) = x' \land v_{K_0}(z_0) = x' \). Further, since \( (O', K) = (I'_0, K_0) \), \( v_{K_0}(y) = v_{K_0}(y_0) \), and \( v_{K_0}(z) = v_{K_0}(z_0) \) we know that there are no rows \( x \in R(O', D_2) \land v_K(z) = x \land v_K(y) = x \).

**Case (b): `Table(T) \notin OP_{G_{i_0}} \land Table(T) \in OP_{G_{i_1}}**

This case is similar to **Case (a)**.

**Case (c): `Table(T) \in OP_{G_{i_0}} \land Table(T) \in OP_{G_{i_1}}**

In this case `CreateAKGraph` returns non-empty set for both inputs.

Assuming there exist rows \( y, z \) such that \( y \in R(o_G, D_1) \land z \in R(o_G, D_2) \land v_C(y) = v_C(z) \). Now we want to show that there are no rows \( x \in R(O', D_2) \land v_K(z) = x \land v_K(y) = x \).

By the semantics of `Join` operator, since \( y \in R(o_G, D_1) \), \( \exists y_0 \in R(I_0, D_1), \exists y_1 \in R(I_1, D_1), y_0 = v_{C_{i_0}}(y) \land y_1 = v_{C_{i_1}}(y) \). And since \( z \in R(o_G, D_2) \), \( \exists z_0 \in R(I_0, D_2), \exists z_1 \in R(I_1, D_2), z_0 = v_{C_{i_0}}(z) \land z_1 = v_{C_{i_1}}(z) \).

Further, since \( T \mapsto C \) and `Table(T) \in OP_{G_{i_0}}`, by the definition of transitive injection, we know that \( o_G \) is injective for \( C_0 \mapsto C \), where \( C_0 \) is the set of columns of the input operator, \( I_0 \), and \( T \mapsto C_0 \). By Lemma 3.7.3, we know that \( y = z \). Hence \( y_0 = z_0 \) and \( v_{C_0}(y_0) = v_{C_0}(z_0) \).
By the induction hypothesis, we know that there are no rows \( x \in R(I_0', D_2) \) such that \( x = v_{K_0}(y_0) \land x = v_{K_0}(z_0) \). Therefore \( y_0 \notin R(J_{old}, D_2) \). We know \( K = cK_0 \), and by the definition of canonical keys, \( I_0.key = K \cap \bar{C}_0 \) and \( I_1.key = K \cap \bar{C}_1 \). Therefore \( \#y' \in R(I_0, D_1) , y' \neq y \land v_{I_0.key}(y) = v_{I_0.key}(y') \). Therefore \( v_{I_0.key}(y) \notin \pi_{I_0.key}(R(\text{Join} \Join_{\text{Prd}}(J_{old}, I_{1,old}), D_2)) \). Since \( v_{I_0.key}(y) = v_{I_0.key}(y) \), we have that \( v_{I_0.key}(y) \notin \pi_{I_0.key}(R(\text{Join} \Join_{\text{Prd}}(J_{old}, I_{1,old}), D_2)) \). Note we know that \( I_0.key \subseteq K \land I_0.key \neq \varnothing \), therefore \( v_K(y) \notin R(J_{a2}, D_2) \).

We can similarly infer that \( v_K(z) \notin \pi_K R(J_{a1}, D_2) \). Since \( y = z \), \( v_K(y) \notin R(J_{a1}, D_2) \). Hence \( v_K(y) \notin R(U_0, D_2) \). Similarly, we can show that \( v_K(y) \notin R(U_1, D_2) \). Thus \( v_K(y) \notin R(O', D_2) \). Since \( y = z \), we know that \( v_K(z) \notin R(O', D_2) \). Therefore we conclude that there are no rows \( x \in R(O', D_2) \land v_K(z) = x \land v_K(y) = x \).

**Case 5: \( G \) is a Union.**

Assume the inputs are \( I_0 \) and \( I_1 \). We prove the lemma in three sub-cases, based on

<table>
<thead>
<tr>
<th>Table ( T ) in ( OP_{G_{I_0}} ) or Table ( T ) in ( OP_{G_{I_1}} )</th>
</tr>
</thead>
</table>

**Case (a): Table \( T \) in \( OP_{G_{I_0}} \) \land Table \( T \) \notin \( OP_{G_{I_1}} \)**

In this case by Lemma 3.7.2 \((I_1', K_1) = (\varnothing, \varnothing)\). Therefore the algorithm will simply return \( K = K_0 \cup \{\text{position}\} \land O' = \text{Project}_K(I_0) \).

Assuming there exist rows \( y, z \) such that \( y \in R(o_G, D_1) \land z \in R(o_G, D_2) \land v_C(y) = v_C(z) \). Now we want to show that there are no rows \( x \in R(O', D_2) \land v_K(z) = x \land v_K(y) = x \).

By Lemma 3.7.3 we know \( y = z \). By the position column, \( y \) is either produced by \( I_0 \) or \( I_1 \). If \( y \in R(I_1, D_1) \), \( v_{\text{position}}(y) = 1 \), since \( \forall x' \in R(O', D_2), v_{\text{position}}(y) = 0 \), there are no rows \( x \in R(O', D_2) \land v_K(z) = x \land v_K(y) = x \).

Now consider \( y \in R(I_0, D_1) \). Let \( U \) be the contributing set for \( y \): \( U = \zeta(y, I_0, o_G, D_1) \), and \( V \) be the contributing set for \( x \): \( V = \zeta(y, I_0, o_G, D_2) \). By the definition of contribut-
ing set for \textbf{Union} operator, $U = \{y_0 \in R(I_0, D_1)|y_0 = y\}$, therefore $U = \{y\}$.

Further, since $T \rightarrow C$ and $\text{Table}(T) \in OP_{G_{i_0}}$, by the definition of transitive injection, we know that $o_G$ is injective for $C_0 \rightarrow C$, where $C_0$ is the set of columns of the input operator, $I_0$, and $T \rightarrow C_0$. By the induction hypothesis, we infer that $\exists x' \in R(I'_0, D_2)$ such that $v_{K_0}(y) = x' \land v_{K_0}(z) = x'$. Finally, since $K = K_0 \cup \{\text{position}\} \land O' = \text{Project}_K(I_0)$, we can conclude that there are no rows $x \in R(O', D_2) \land v_K(z) = x \land v_K(y) = x$.

\textbf{Case (b):} $\text{Table}(T) \notin OP_{G_{i_0}} \land \text{Table}(T) \in OP_{G_{i_1}}$

This case is similar to \textbf{Case (a)}.

\textbf{Case (c):} $\text{Table}(T) \in OP_{G_{i_0}} \land \text{Table}(T) \in OP_{G_{i_1}}$

In this case CreateAKGraph returns non-empty set for both inputs.

Because $o_G$ is injective for $T \rightarrow C$ and $\text{Table}(T) \in OP_{G_{i_0}}$, it follows that $I_0$ is transitively injective for a set of columns $C_0$ in $I_0$, i.e., $T \rightarrow C_0$. Similarly we infer that $I_1$ is transitively injective for $C_1$ in $I_1$, i.e., $T \rightarrow C_1$.

Assuming there exist rows $y, z$ such that $y \in R(o_G, D_1) \land z \in R(o_G, D_2) \land v_C(y) = v_C(z)$. Now we want to show that there are no rows $x \in R(O', D_2) \land v_K(z) = x \land v_K(y) = x$.

By the semantics of \textbf{Union} operator and the position column, $y$ is either produced by $I_0$ or $I_1$. Without loss of generality, assume $y$ is produced by $I_0$. Let $U$ be the contributing set for $y$: $U = \zeta(y, I_0, o_G, D_1)$, and $V$ be the contributing set for $z$: $V = \zeta(y, I_0, o_G, D_2)$. By the definition of contributing set for \textbf{Union} operator, $U = \{y_0 \in R(I_0, D_1)|y_0 = y\}$, therefore $U = \{y\}$.

Since $v_C(y) = v_C(z)$ and $I_0 \rightarrow o_G$, we know $V \neq \emptyset$ and $V = \{z\}$. Since we already know that $T \rightarrow C_0$, by the induction hypothesis, we infer that $\exists x' \in R(I'_0, D_2)$ such that $v_{K_0}(y) = x' \land v_{K_0}(z) = x'$. Therefore, $y \notin \text{Join}_{K_0}(I_{0, \text{add}}, I_0)$. 
Further, since $K = ck_{o_G}$ and we know that $\exists y'' \in R(o_G, D_2), y'' \neq y \land v_K(y'') = v_K(y)$, therefore by the semantics of $\text{Union}$ operator, $\exists y'' \in R(I_{0,old}, D_2), y'' \neq y \land v_K(y'')$. Therefore $v_K(y) \notin \pi_K(\text{Join}_{K_0}(I_{0,old}, I_0'))$.

Similarly we can show that $v_K(z) \notin \pi_K(\text{Join}_{K_0}(I_0, I_0'))$. By Lemma 3.7.3, we have $y = z$, and therefore $v_K(y) \notin \pi_K(\text{Join}_{K_0}(I_0, I_0'))$.

Hence $v_K(y) \notin R(O', D_2)$. It follows that $v_K(z) \notin R(O', D_2)$. Therefore there are no rows $x \in R(O', D_2) \land v_K(z) = x \land v_K(y) = x$.

**Case 6: $o_G$ is a $\text{SetDiff}$**

Assuming $I_0$ is the left input operator and $I_1$ is the right. Now we prove this case in three sub-cases, based on $\text{Table}(T) \in OP_{G_{I_0}}$ or $\text{Table}(T) \in OP_{G_{I_1}}$.

**Case (a):** $\text{Table}(T) \in OP_{G_{I_0}} \land \text{Table}(T) \notin OP_{G_{I_1}}$.

By Lemma 3.7.2 $(I_1', K_1) = (\emptyset, \emptyset)$. Therefore the algorithm will simply return $K = K_0 \land O' = I_0'$.

Assuming there exist rows $y, z$ such that $y \in R(o_G, D_1) \land z \in R(o_G, D_2) \land v_C(y) = v_C(z)$. Now we want to show that there are no rows $x \in R(O', D_2) \land v_K(z) = x \land v_K(y) = x$.

First by Lemma 3.7.3, $y = z$, and $v_{C_0}(y) = v_{C_0}(z)$. Then by the semantics of $\text{SetDiff}$ operator, we know that $y \in R(I_0, D_1)$ and $z \in R(I_0, D_2)$.

Further, since $T \mapsto C$ and $\text{Table}(T) \in OP_{G_{I_0}}$, by the definition of transitive injection, we know that $o_G$ is injective for $C_0 \mapsto C$, where $C_0$ is the set of columns of the input operator, $I_0$, and $T \mapsto C_0$.

By the induction hypothesis, $\exists x' \in R(I_0', D_2)$ such that $v_{K_0}(z) = x \land v_{K_0}(y) = x'$. Hence our lemma holds because $K = K_0 \land O' = I_0'$.

**Case (b):** $\text{Table}(T) \in OP_{G_{I_0}} \land \text{Table}(T) \notin OP_{G_{I_1}}$.

In this case, by Lemma 3.7.2 $(I_0', K_0) = (\emptyset, \emptyset)$. Therefore the algorithm will return
\[ O' = \text{Join}_{K_1}(I'_1, I_1), K' = ck_1. \]

Since \( T \mapsto C \) and \( \text{Table}(T) \in OP_{G_1} \), by the definition of transitive injection, we know that \( o_G \) is injective for \( C_1 \mapsto C \), where \( C_1 \) is the set of columns of the input operator, \( I_1 \), and \( T \mapsto C_1 \).

By the definition of contributing set, \( Y' : \zeta(y, I_1, o_G, D_1) = R(I_1, D_1) \) and \( Z' : \zeta(z, I_1, o_G, D_2) = R(I_1, D_2) \).

Since \( v_C(y) = v_C(z) \) and \( C_1 \mapsto C \), we know that \( \pi_{C_1}(Y') = \pi_{C_1}(Z') \). I.e., \( \forall y \in Y', \exists z \in Z', y = z \) and \( \forall z \in Z', \exists y \in Y', v_C(y) = v_C(z) \). Then by Lemma 3.7.3, \( \forall y \in Y', \exists z \in Z', y = z \), and \( \forall z \in Z', \exists y \in Y', y = z \). In other words, \( Y' = Z' \).

On the other hand, since \( T \mapsto C_1 \), by the induction hypothesis, \( \forall y \in R(I_1, D_1), \forall z \in R(I_1, D_2), y = z \Rightarrow (\exists x' \in R(I'_1, D_2), v_{K_1}(y) = x' \Rightarrow v_{K_1}(z) = x') \). Since \( \forall y \in Y', \exists z \in Z', y = z \), we infer that \( \forall y \in R(I_1, D_1), \exists x' \in R(I'_1, D_2), v_{K_1}(y) = x' \). Hence \( R(\text{Join}_{K_1}(I_1, I'_1), D_1) = \emptyset \).

Similarly we can show that \( R(\text{Join}_{K_1}(I_{1, odd}, I'_1), D_2) = \emptyset \). Therefore by the semantics of \textbf{Project} operator and \textbf{Union} operator, \( R(O', D_2) = \emptyset \). Therefore there are no rows \( x \in R(O', D_2) \land v_K(z) = x \land v_K(y) = x \).

\textbf{Case (c)}: \( \text{Table}(T) \in OP_{G_{10}} \land \text{Table}(T) \in OP_{G_{11}} \).

Assuming there exist rows \( y, z \) such that \( y \in R(o_G, D_1) \land z \in R(o_G, D_2) \land v_C(y) = v_C(z) \). Now we want to show that there are no rows \( x \in R(O', D_2) \land v_K(z) = x \land v_K(y) = x \).

Then by the semantics of \textbf{SetDiff} we know that \( y \in R(I_0, D_1) \) and \( y \in R(I_0, D_2) \). Let \( Y_0 = \zeta(y, I_0, o_G, D_1) \) and \( Z_0 = \zeta(z, I_0, o_G, D_2) \), and it follows that \( Y_0 = \{ y \} \) and \( Z_0 = \{ z \} \). Further \( v_{C_0}(y) = v_{C_0}(z) \) because \( y = z \), by the induction hypothesis, there are no rows \( x' \in R(I', D_2) \land v_{K'}(z) = x' \land v_{K'}(y) = x' \). Therefore...
$y \not\in R(\text{Join}_{K_0}(I_{0,old}, I_1'), D_2)$.

Since $K$ is the canonical key of $o_G$ and $I_0$, $\exists y' \in R(I_0, D_1) \land v_K(y') = v_K(y)$. Since $K_0 \subseteq \tilde{C}_I$, $R(\text{Join}_{K_0}(I_0, I_0'), D_1) \subseteq R(I_0, D_1)$. Because $R(I_0, D_1) = R(I_{0, old}, D_2)$, we infer that $\exists y' \in R(\text{Join}_{K_0}(I_{0,old}, I_0'), D_2) \land v_K(y') = v_K(y)$. Similarly, we can show that $y \not\in R(\text{Join}_{K_0}(I_0, I_0'), D_2)$ and $\exists y'' \in R(\text{Join}_{K_0}(I_0, I_0'), D_2) \land v_K(y'') = v_K(y)$.

Therefore $y \not\in R(U_0, D_2)$ and $\exists y''' \in R(U_0, D_2) \land v_K(y''') = v_K(y)$. Therefore $v_K(y) \not\in R(\pi_K(U_0), D_2)$.

On the other hand, since $P_b$ is constructed similar to Case (b), and therefore by the same argument, we can show that $v_K(y) \not\in R(\pi_K(U_1), D_2)$.

Therefore we can conclude that $v_K(y) \not\in R(O', D_2)$ and $v_K(z) \not\in R(O', D_2)$. I.e., there are no rows $x \in R(O', D_2) \land v_K(z) = x \land v_K(y) = x$.

\(\square\)

Finally, we can prove that CreateAROpt will not produce spurious updates.

**Theorem 3.7.5.** Given an Update event, a view graph $G$ which is injective for all columns of its top operator, and table $T$, let $G_{\text{affected}} = \text{CreateARGraph}(\text{UPDATE}, G, T)$, and $G_{\text{opt}} = \text{CreateAROpt}(\text{UPDATE}, G, T)$. Then for all valid database transitions $D \xrightarrow{T} D'$ with pruned transition tables $\triangledown T$ and $\blacktriangleleft T$, $(\text{OLD\_ROW}, \text{NEW\_ROW}) \in R(\alpha_{G_{\text{opt}}}, D')$ if and only if $(\text{OLD\_ROW}, \text{NEW\_ROW}) \in R(\alpha_{G_{\text{affected}}}, D')$.

**Proof.** It is easy to see that the “if” direction is true: from the definition of pruned transition tables, it follows that pruned transition tables also satisfy the definition of transition tables, so the correctness of Theorem 2 is not affected by pruning the transition tables. The only difference then is that CreateAROpt does not perform the final selection: $\text{Select}_{\text{NEW\_ROW} \neq \text{OLD\_ROW}}(\text{OLD\_ROW}, \text{NEW\_ROW})$. Since Select can only decrease the cardinality of its input, there will be no affected nodes lost as a result of the change.
Thus, \((\text{OLD}_\text{ROW}, \text{NEW}_\text{ROW}) \in R(o_{\text{affected}}, D')\) implies that \((\text{OLD}_\text{ROW}, \text{NEW}_\text{ROW}) \in R(o_{\text{opt}}, D')\).

Conversely, given an \textsc{Update} event, we need to show that \(\forall \text{OLD}_\text{ROW} \in R(o_G, D), \forall \text{NEW}_\text{ROW} \in R(o_G, D'), \text{OLD}_\text{ROW} = \text{NEW}_\text{ROW} \Rightarrow (\text{OLD}_\text{ROW}, \text{NEW}_\text{ROW}) \notin R(G_{\text{opt}}, D')\). Let \((O', K) = \text{CreateAKGraph}(o_G, o_{G,\text{old}}, T)\), and let \(p_o = \text{OLD}_\text{ROW}, p_n = \text{NEW}_\text{ROW}\). \(p_o = p_n\) implies that \(v_{C(o_G)}(p_o) = v_{C(o_G)}(p_n)\). Since \(G\) is injective for all columns of \(o_G\), by Lemma 3.7.4, \(\exists x \in R(O', D'), x = v_K(p_o) \land x = v_K(p_n)\). Therefore we have (1) \(p_o \notin R(O_{\text{old}}, D')\) and \(p_n \notin R(O_{\text{new}}, D')\); (2) \(p_o \in R(O_{\text{old}}, D')\) and \(p_n \notin R(O_{\text{new}}, D')\); (3) \(p_o \notin R(O_{\text{old}}, D')\) and \(p_n \in R(O_{\text{new}}, D')\). In each of these cases \((\text{OLD}_\text{ROW}, \text{NEW}_\text{ROW}) \notin R(G_{\text{opt}}, D')\).
Chapter 4

Ranked Keyword Search over Virtual XML Views

As mentioned in the introduction, emerging Internet applications such as personal portal and information integration systems often require ranked keyword search queries over virtual (unmaterialized) semi-structured views. In this chapter, we first illustrate two such important classes of applications, and then describe our proposed techniques for efficiently evaluating ranked keyword search queries over virtual XML views.

Consider the following two classes of applications which require ranked keyword search queries over virtual XML views.

Personalized Views: Consider a large online web portal such as MyYahoo that caters to millions of users. Since different users may have different interests, the portal may wish to provide a personalized view of the content to its users (such as books on topics of interest to the user along with their reviews, and latest headlines along with previous related content seen by the user, etc.), and allow users to search such views. As another example, consider an enterprise search platform such as Microsoft Sharepoint that is available to all employees. Since different employees may have different permission levels, the enterprise must provide personalized views according to specific levels, and allow employees to search only such views. In such cases, it may not be feasible to materialize all user views because there are many users and their content is often overlapping, which could lead to data duplication and its associated space-overhead. In contrast, a more scalable strategy is to define virtual views for different users of the system, and allow users to search over their virtual views.

Information Integration: Consider an information integration application involving
two query-able XML web services: the first service provides books and the second service provides reviews for books. Using these services, an aggregator wishes to create a portal in which each book contains its reviews nested under it. A natural way to specify this aggregation is as an XML view, which can be created by joining books and reviews on the isbn number of the book, and then nesting the reviews under the book (Figure 4.1). Note that the view is often virtual (unmaterialized) for various reasons: (a) the aggregator may not have the resources to materialize all the data, (b) if the view is materialized, the contents of the view may be out-of-date with respect to the base data, or maintaining the view in the face of updates may be expensive, and/or (c) the data sources may not wish to provide the entire data set to the aggregator, but may only provide a subset of the data in response to a query. While current systems (e.g., [24, 39, 52]) allow users to query virtual views using query languages such as XQuery, they do not support ranked keyword search queries over such views.
In such scenarios, techniques proposed in traditional information retrieval systems are not applicable because they rely heavily on a fundamental assumption that the set of documents being searched is materialized. For instance, the popular inverted list organization and associated query evaluation algorithms [12, 103] assume that the (materialized) documents can be parsed, tokenized and indexed when the documents are loaded into the system. Further, techniques for ranking (scoring) results such as TF-IDF [103] rely on statistics gathered from materialized documents such as term frequencies (number of occurrences of a keyword in a document) and inverse document frequencies (the inverse of the number of documents that contain a query keyword). Finally, even document filtering systems, which match streaming documents against a set of user keyword search queries (e.g., [29, 46]), assume that the document is fully materialized at the time it is handed to the streaming engine, and all processing is tailored for this scenario.

Consequently, an interesting challenge arises in these applications: how do we efficiently evaluate keyword search queries over virtual XML views? One simple approach is to materialize the entire view at query evaluation time and then evaluate the keyword search query over the materialized view. However, this approach has obvious disadvantages. First, the cost of materializing the entire view at runtime can be prohibitive, especially since only a few documents in the view may contain the query keywords. Further, users issuing keyword search queries are typically interested in only the results with highest scores, and materializing the entire view to produce only top few results is likely to be expensive.

To address the above issues, we propose an alternative strategy for efficiently evaluating keyword search queries over virtual XML views. The key idea is to use regular indices, including inverted list and XML path indices, that are present on the base data to efficiently evaluate keyword search over views. The indices are used to efficiently
identify the portion of the base data that is relevant to the current keyword search query so that only the top ranked results of the view are actually materialized and presented to the user.

The above strategy poses two main challenges. First, XML view definitions can be fairly complex, involving joins and nesting, which leads to various subtleties. As an illustration, consider Figure 4.1 above. If we wish to find all books with nested reviews that contain the keywords “XML” and “search”, then ideally we want to materialize only those books and reviews such that they \textit{together} contain the keywords “XML” and “search” (even though no book or review may \textit{individually} contain both the keywords). However, we cannot determine which reviews belong to which book (to check whether they \textit{together} contain both the keywords) without actually joining the books and reviews on the isbn number, which is a data value. This presents an interesting dilemma: how do we selectively extract some fields needed for determining related items in the view (e.g., isbn number) without actually materializing the entire view?

The second challenge stems from ranking the keyword search results. As mentioned earlier, popular ranking methods such as TF-IDF require statistics gathered from the documents being searched. How do we efficiently compute statistics on the view from the statistics on the base data, so that the resulting scores and rank order of the query results is exactly the same as when the view is materialized?

Our solution to the above problem is a three-phase algorithm that works as follows. In the first phase, the algorithm analyzes the view definition and query keywords to identify a \textit{query pattern tree} (or QPT) for each data source (such as books and reviews); the QPT represents the precise parts of the base data that are required to compute the potential results of the keyword search query. In the second phase, the algorithm uses existing inverted and path indices on the base data to compute \textit{pruned document trees}
(or PDT) for each data source; each PDT contains only small parts of the base data tree that correspond to the QPT. The PDT is constructed solely using indices, without having to access the base data. In this phase, the algorithm also propagates keyword statistics in the PDTs. In the third phase, the query is evaluated over the PDTs, and the top-k result PDTs are expanded into the complete document trees; this is the only phase where the base data is accessed (for the top-k results only). An interesting aspect of our solution is that we require no changes to the normal XQuery query evaluator as all the keyword specific parts are done during query rewrite, pre-processing and post-processing.

We have experimentally compared our approach with two alternatives: the naive approach that materializes the entire view at query time, and GTP [36] with TermJoin [8], which is a state of the art implementation of integrating structure and keyword search queries. Our experimental results show that our approach is more than 10 times faster than these alternatives. The performance gain comes about for two main reasons. First, we use path indices to efficiently create PDTs, thereby avoiding more expensive structural joins. Second, we selectively materialize the element values required during query evaluation using indices, without having to access the base data. We also compared our PDT generation with techniques of projecting XML documents proposed in [85]. Our algorithm again performs more than an order of magnitude faster because we generate PDTs by solely using indices.

In summary, we believe that the proposed approach is the first optimized end-to-end solution for efficient keyword search over virtual XML views. The specific contributions are:

- A system architecture for efficiently evaluating keyword search queries over virtual XML views (Section 4.2).
- Efficient algorithms for generating pruned XML elements needed for query eval-
4.1 Background

We first provide some background on XML storage, XML indexing and XML scoring, before presenting our problem definition.

4.1.1 Background on XML storage & indexing

One of the key concepts in XML storage is the notion of element ids, which is a way to uniquely identify an XML element. Various id formats have been proposed in the literature (e.g., [108, 115]), and one popular variant is Dewey IDs which has been shown to be effective for search [64] and update [92] queries. Dewey IDs is a hierarchical numbering scheme where the ID of an element contains the ID of its parent element as a prefix. An example XML document in which Dewey IDs are assigned to each node is shown in Figure 4.2. Given element ids, various XML document storage formats have

\[<books id="1">\]
\[<book id="1.1">\]
\[<isbn id="1.1.1">111-11-1111</isbn>\]
\[
...
</book>
\[<book id="1.2">...</book>\]
\[
...
</books>\]

Figure 4.2: An XML document with Dewey Ids

uation and scoring, by solely using indices (Section 4.4).

• Evaluation and comparison of the proposed approach using the 500MB INEX dataset\(^1\) (Section 4.5).

\(^1\)http://inex.is.informatik.uni-duisburg.de:2004
been proposed for efficient storage [53, 113].

Another important aspect is XML indexing. At a high-level, there are two types of XML indices: path indices and inverted list indices (these indices can sometimes be combined [79]). Path indices are used to evaluate XML path and twig (i.e., branching path) queries. Inverted list indices are used to evaluate keyword search queries over (materialized) XML documents. We now describe representative implementations for each type of index.

One effective way to implement path indices is to encode XML paths as values, and then index these paths along with the data values. Indexing the paths and values can be done using specialized indices (e.g., [42, 59]) or using regular relational indices such as B+-tree [35, 113]. We focus on the latter as it is easier to implement. Figure 4.3 shows an example path index for the document in Figure 4.1. As shown, the index has two tables, the Paths table and the Value-IDs table. The Paths table contains one row for each unique schema-level path from the root to an element in the document, and each such path is assigned a unique PathID. A B+-tree index is built on the Path attribute. The second table contains one row for each unique (pathid, value) pair, where pathid represents the value of a PathID attribute, and value represents the atomic value of an element on the path corresponding to the pathid. For each unique (pathid, value) pair, the table stores an IDList value, which is the list of ids of all elements on the path corresponding to

<table>
<thead>
<tr>
<th>Path</th>
<th>Value</th>
<th>IDList</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>/books/book/isbn</td>
<td>“111-111-1111”</td>
<td>1.1.1, 1.3.1</td>
</tr>
<tr>
<td>/books/book/isbn</td>
<td>“222-222-2222”</td>
<td>1.2.1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>/books/book/author/fn</td>
<td>“Jane”</td>
<td>1.2.3, 1.7.3</td>
</tr>
</tbody>
</table>
pathid with that atomic value. The table also contains a row corresponding to (pathid, null), and the corresponding IDList value stores the list of all ids of elements on that path, regardless of value. A B+-tree index is built on the (pathid,value) pair.

Given the above two tables, queries are evaluated as follows. First, consider a simple path query such as: /book/author/fn. This query is evaluated in two steps. In the first step, the Paths table is probed to determine the pathid corresponding to the query path (note that the number of distinct paths is usually very small, so the Paths table usually fits in memory). In the second step, the Values-ID table is probed with the search key (pathid,null) to get the list of result element ids. A path query such as /book/author/fn[. = ’Jane’ ] is evaluated similarly, except that during the second step, the index is probed using the search key (pathid,’Jane’). For path queries with descendant axes, such as /book//fn, the first probe may return more than one path id (since there may be many paths such as /book/fn and /book/author/fn that match the path); thus, the second step is done for each path id returned. Finally, twig queries such as /book/author[fn = ’Jane’ and ln = ’Doe’] can be evaluated by first evaluating each individual path query (/book/author/fn = ’Jane’ and /book/author/ ln = ’Doe’), and then merging the results based on the author id (note that the Dewey ID of the author can be obtained from the Dewey IDs of ln and fn).

The second type of XML indices are inverted list indices. XML inverted list indices
let $view :=$
where $book/year > 1995$
return <bookrevs>
  <book>{ $book/title } </book>,
  { for $rev$ in fn:doc(reviews.xml)/reviews//review
    where $rev/isbn = $book/isbn
    return $rev/content }
</bookrevs>
for $bookrev$ in $view$
where $bookrev$ ftcontains('XML' & 'Search')
return $bookrev$

Figure 4.5: Keyword Search over XML view

(e.g., [64, 88, 115]) typically store for each keyword in the document collection, the list of XML elements that directly contain the keyword. Figure 4.4 shows an example inverted list for our example document. In addition, an index such as a B+-tree is usually built on top of each inverted list so that we can efficiently check whether a given element contains a keyword.

4.1.2 Scoring

An important issue for keyword search queries is scoring the results. There have been many proposals for scoring XML keyword search results [9, 11, 55, 64, 86]. In the paper we focus on the commonly used TF-IDF method proposed in the context of XML documents [55]. In this context, tf and idf values are calculated with respect to XML elements, instead of entire documents as in the traditional information retrieval. Specifically, given an XML view $V$ over a database $D$, the TF-IDF method defines two measures:

- $tf(e, k)$, which is the number of distinct occurrences of the keyword $k$ in element $e$ and its descendants (where $e \in V(D)$), and

- $idf(k) = \frac{|V(D)|}{|e|e\in V(D)\wedge contains(e,k)}$ (the ratio of the number of elements in the view result $V(D)$ to the number of elements in $V(D)$ that contain the keyword $k$).
Given the above measure, the score of a result element $e$ for a keyword search query $Q$ is defined to be: $score(e, Q) = \sum_{k \in Q}(tf(e, k) \times idf(k))$.

However, the unnormalized score is biased towards large elements because they tend to have a higher tf value. To alleviate this bias, the score is usually normalized. Various normalization factors are possible but some indicator of the document length, such as the byte size of the document [117], is usually used since it is effective and simple. We use $len(e)$ to denote the byte length of an element $e$. Then, the normalized score for a result element $e$ given some keyword $K$ is: $score(e, K) = \frac{\text{unnorm-score}(e, K)}{len(e)}$.

### 4.1.3 Problem definition

We use a set of keywords $Q = \{k_1, k_2, ..., k_n\}$ to represent a keyword search query, and define the problem of keyword search over views as follows.

**Problem KS:** Given a view $V$ defined over a database $D$, the result of a keyword search query $Q$, denoted as $RES(Q,V,D)$, is the sequence $s$ such that:

- $\forall e \in s, e \in V(D)$, and
- $\forall e \in s \forall k \in Q(contains(e,k))$, and
- $\forall e \in V(D)(\forall k \in Q(contains(e,k)) \Rightarrow e \in s)$

Figure 4.5 illustrates a keyword query \{‘XML’, ‘Search’\} over the view corresponding to the variable $\$view$. Given the definition of score in the previous section, we can further define the problem of ranked keyword search as follows.

**Problem Ranked-KS:** Given a view $V$ defined over a database $D$ and the number of desired results $k$, the result of a ranked keyword query $Q$ is the set of $k$ elements with highest scores in $RES(Q,V,D)$, where we break ties arbitrarily.

The above definition captures the result of conjunctive ranked keyword search queries.
over views. Our system also supports disjunctive queries which can be defined similarly.

4.2 System Overview

In this section we describe the proposed system architecture for processing ranked keyword search queries over virtual XML views.

4.2.1 System architecture

Figure 4.6 shows our proposed system architecture and how it relates to traditional XML full-text query processing. The top big box denotes the query engine sub-system and the bottom big box denotes the storage and index subsystem. The solid lines show the traditional query evaluation path for full-text queries (e.g., [16, 43, 78, 90]). The query is parsed, optimized and evaluated using a mix of structure and inverted list indices and document storage. However, as mentioned in the introduction, traditional query engines are not designed to support efficient keyword search queries over views. Consequently, they either disallow such queries (e.g., [43, 90]), materialize the entire view before evaluating the keyword search query (e.g. [16]), or do not support such queries efficiently (e.g., [78]), as verified in our performance study (Section 4.5).

To efficiently process keyword search queries over views, we adapt the existing query engine architecture by adding three new modules (depicted by dashed boxes in Figure 4.6). The modified query execution path (depicted by dashed lines in Figure 4.6) is as follows. On detecting a keyword search query over a view that satisfies certain conditions (clarified at the end of this section), the parser redirects the query to the Query Pattern Tree (QPT) Generation Module. The QPT, which is a generalization of the GTP [36], identifies the precise parts of the base data that are required to compute
the results of the keyword search query. The QPT is then sent to the Pruned Document Tree (PDT) Generation Module. This module generates PDTs (i.e., a projection of the base data that conforms to the QPT) using only the path indices and inverted list indices; consequently, the generation of PDTs is expected to be fast and cheap.

The QPT Generation Module also rewrites the original query to go over PDTs instead of the base data and sends it to the traditional query optimizer and evaluator. Note that our proposed architecture requires no changes to the XML query evaluator, which is usually a large and complex piece of code. The rewritten query is then evaluated using PDTs to produce the view that contains all view elements with pruned content (determined using path indices), along with information about scores and query keywords contained (determined using inverted indices). These elements are then scored by the Scoring & Materialization Module, and only those with highest scores are fully materialized using document storage.

Our current implementation supports views specified using a powerful subset of XQuery, including XPath expressions with named child and descendant axes, predicates on leaf values, nested FLWOR expressions, non-recursive functions. We currently do not support predicates on the string values of non-leaf elements and other XPath axes

Figure 4.6: Keyword query processing architecture
such as sibling and position based predicates, although it is possible to extend our system to handle these axes by using an underlying structure index that supports these axes (e.g., [37]).

The supported grammar in our keyword query processing architecture is given below, where $Expr$ is the root production and $VAR$ and $TAGNAME$ correspond to variables and element tag names, respectively.

$$
Expr :: \begin{array}{c}
\text{PathExpr} \mid FLWORExpr \mid \text{CondExpr} \\
\text{FunctionCall} \mid \text{FunctionDecl}
\end{array}
$$

$$
\text{PathExpr} ::= \begin{array}{c}
\text{fn:doc(Name)} \mid \text{VAR} \mid . \\
\mid (\text{fn:doc(Name)} \mid \text{VAR} \mid .) (\div \mid \div) \text{PathTailExpr} \\
\mid \text{PathExpr} [ PredExpr ]
\end{array}
$$

$$
\text{PathTailExpr} ::= \begin{array}{c}
\text{TAGNAME} \mid \text{TAGNAME} (\div \mid \div) \text{PathTailExpr}
\end{array}
$$

$$
\text{PredExpr} ::= \begin{array}{c}
\text{PathExpr} \mid \text{PathExpr} \text{Comp} \text{Literal} \\
\mid \text{PathExpr} \text{Comp} \text{PathExpr}
\end{array}
$$

$$
\text{Comp} ::= '=' \mid '<' \mid '>'
$$

$$
\text{CondExpr} ::= 'if' \text{Expr} 'then' \text{Expr} 'else' \text{Expr}
$$

$$
\text{FLWORExpr} ::= (\text{ForClause} \mid \text{LetClause})^+ \\
\text{WhereClause}? \text{ReturnClause}
$$

$$
\text{ForClause} ::= 'for' \text{VAR} 'in' \text{PathExpr}
$$

$$
\text{LetClause} ::= 'let' \text{VAR} 'in' \text{PathExpr}
$$

$$
\text{WhereClause} ::= 'where' \text{PredExpr}
$$

$$
\text{ReturnClause} ::= 'return' \text{RetExpr}
$$

$$
\text{RetExpr} ::= \begin{array}{c}
\text{Expr} \\
\mid '<' \text{TAGNAME} '>' (\{' \text{RetExpr} \'})* '<' \text{TAGNAME} '>' \\
\mid \text{Expr} ',' \text{Expr}
\end{array}
$$

$$
\text{FunctionCall} ::= \text{QName} "(" \text{PathExpr} (',' \text{PathExpr})^? "")"
$$

$$
\text{FunctionDecl} ::= 'declare' 'function' \text{QName} \\
\text{QName} '(' \text{ParamList}? ')'? '{' \text{Expr} '}'
$$

$$
\text{ParamList} ::= \text{VAR} (',' \text{VAR})^*
$$

It is easy to verify that the view in our running example 4.5 conforms to the above grammar.

In the rest of this chapter, we first describe the details of QPT Generation Module and then focus on the PDT Generate Module, which is one of the main technical contributions of the proposed system architecture.
4.3 QPT Generation Module

4.3.1 QPT illustration and definition

Consider the XML view in Figure 4.5. In order to evaluate the view query, we only need a small subset of the data, such as the isbn numbers of books and isbn numbers of reviews (which are required to perform a join). It is only when we want to materialize the view results do we need additional content such as the titles of books and content of reviews. The QPT is essentially a principled way of capturing this information.

The QPT is a generalization of the Generalized Tree Patterns (GTP) [36], which was originally proposed in the context of evaluating complex XQuery queries. The GTP captures the structural parts of an XML document that are required for query processing. The QPT augments the GTP structure with two annotations, one that specifies which parts of the structure and associated data values are required during query evaluation, and the other that specifies which parts are required during result materialization.
Figure 4.7(a) shows the QPTs for the book and review documents referenced in our running example. We first describe the features present in a GTP. GTP consists of nodes (which may have predicates) referenced during query execution and optional or mandatory edges corresponding to an XPath axis that can be traversed (during query execution) from a parent with the parent tag name to a child with the child tag name. In our example, the edge between book and isbn is optional, represented by dotted lines, because a book can be present in the view result even if it does not have an isbn number; the edge between review and isbn is mandatory, represented by solid lines, because a review is of no relevance to query execution unless it has an isbn number (otherwise, it does not join with any book and is just irrelevant to the content of the view). Further, as usual in twigs, a double line denotes ancestor/descendant relationship and a single line denotes a parent/child relationship.

We now describe new features present in a QPT. QPT has up to two annotations on each node, the first type of annotation is a v-annotation, which indicates that the value of an element corresponding to the node is required during query evaluation. In our example, the ‘isbn’ node in both the book and review QPT is marked with a ‘v’ since they are required for performing a join operation. The second type of annotation is a c-annotation, which indicates that the content of an element corresponding to the node is required only during materialization (note that the content of an element present in the base data is the entire subtree rooted at that element). In our example, the ‘title’ and ‘content’ nodes are marked as ‘c’ nodes since their content is propagated to the view output. Note that a node can be marked with both a ‘v’ and a ‘c’ if it is used during evaluation and propagated to the view output, although there is no instance of this case in our example.

We now introduce some notation that is used in subsequent sections. A QPT is a
tree $Q = (N, E)$ where $N$ is the set of nodes and $E$ is the set of edges. For each node $n$ in $N$, $n.tag$ is its tag name, $n.preds$ is the set of predicates associated with $n$, and $n.ann$ is its node annotation(s), which can be 'v', 'c', both, or neither. For each edge $e$ in $E$, $e.parent$ and $e.child$ are the parent and child node of $e$, respectively; $e.axis$ is either '/' or '// corresponding to an XPath axis, and $e.ann$ is either 'o' or 'm' corresponding to an optional or a mandatory edge.

In our system, we design and implement a QPT algorithm GenerateQPT() (details are presented in Section 4.6) similar to the GTP algorithm except that we additionally handle the 'c' and 'v' node annotations and we support a larger XQuery grammar that includes functions.

### 4.4 PDT Generation Module

We now turn our attention to the PDT Generation Module (Figure 4.6), which is one of the main technical contributions in the paper. The PDT Generation Module efficiently generates a PDT for each QPT. Intuitively, the PDT only contains elements that correspond to nodes in the QPT and only contains element values that are required during query evaluation. For example, Figure 4.7(b) shows the PDT of the book document for its QPT shown in Figure 4.7(a). The PDT only contains elements corresponding to the nodes books, book, isbn, title, and year, and only the elements isbn and year have values.

Using PDTs in our architecture offers two main advantages. First, the query evaluation is likely to be more efficient and scalable because the query evaluator processes pruned documents which are much smaller than the underlying data. Further, using PDTs allows us to use the regular (unmodified) query evaluator for keyword query processing.
We note that the idea of creating small documents is similar to projecting XML documents (PROJ for short) proposed in [85]. There are, however, several key differences, both in semantics and in performance. First, while PROJ deals with isolated paths, we consider twigs with more complex semantics. As an example, consider the QPT for the book document in Figure 4.7(a). For the path books/book/isbn, PROJ would produce and materialize all elements corresponding to book (and its subelements corresponding to isbn). In contrast, we only produce book elements which have year subelements whose values are greater than 1995, which is enforced by the entire twig pattern. Second, instead of materializing every element as in PROJ, we selectively materialize a (small) portion of the elements. In our example, only the elements corresponding to isbn and year are materialized. Finally, the most important difference is that we construct the PDTs by solely using indices, while PROJ requires full scan of the underlying documents which is likely to be inefficient in our scenario. Our experimental results in Section 4.5 show that our PDT generation is more than an order of magnitude faster than PROJ.

We now illustrate more details of PDTs before presenting our algorithms.

4.4.1 PDT Illustration & Definition

The key idea of a PDT is that an element $e$ in the document corresponding to a node $n$ in the QPT is selected for inclusion only if it satisfies three types of constraints: (1) an ancestor constraint, which requires that an ancestor element of $e$ that corresponds to the parent of $n$ in the QPT should also be selected, (2) a descendant constraint, which requires that for each mandatory edge from $n$ to a child of $n$ in the QPT, at least one child/ descendant element of $e$ corresponding to that child of $n$ should also be selected, and (3) a predicate constraint, which requires that if $e$ is a leaf node, it satisfies all pred-
icates associated with \( n \). Consequently, there is a mutual restriction between ancestor and descendant elements. In our example, only reviews with at least one isbn subelement are selected (due to the descendant constraint), and only those isbn and content elements that have a selected review are selected (due to the ancestor constraint). Note that this restriction is not “local”: a content element is not selected for a review if that review does not contain an isbn element.

We now formally define notions of PDTs. We first define the notion of candidate elements that only captures descendant restrictions.

**Definition 4.4.1** (candidate elements). *Given a QPT \( Q \), an XML document \( D \), the set of candidate elements in \( D \) associated with a node \( n \in Q \), denoted by \( CE(n, D) \), is defined recursively as follows.*

- \( n \) is a leaf node in \( Q \): \( CE(n, D) = \{ v \in D \mid \text{tag name of } v \text{ is } n.tag \land \text{the value of } v \text{ satisfies all predicates in } n.preds \} \).

- \( n \) is a non-leaf node in \( Q \): \( CE(n, D) = \{ v \in D \mid \text{tag name of } v \text{ is } n.tag \land \text{for every edge } e \text{ in } Q, \text{if } e.parent \text{ is } n \text{ and } e.ann \text{ is } 'm' \text{ (mandatory), then } \exists ec \in CE(e.child, D) \text{ such that (a) } e.axis = '/' \Rightarrow v \text{ is the parent of } ec, \text{ and (b) } e.axis = '//' \Rightarrow v \text{ is an ancestor of } ec \} \)

Definition 4.4.1 recursively captures the descendant constraints from bottom up. For example, in Figure 4.7(a), candidate elements corresponding to “review” must have a child element “isbn”. Now we define notions of PDT elements which capture both ancestor and descendant constraints.

**Definition 4.4.2** (PDT elements). *Given a QPT \( Q \), an XML document \( D \), the set of PDT elements associated with a node \( n \in Q \), denoted by \( PE(n, D) \), is defined recursively as follows.*
• $n$ is the root node of $Q$: $PE(n, D) = CE(n, D)$

• $n$ is the non-root node in $Q$: $PE(n, D) = \{ v \in D \mid v \text{ is in } CE(n, D) \land \text{ for every edge } e \text{ in } Q, \text{ if } e.\text{child} = n, \text{ then } \exists vp \in PE(e.\text{parent}, D) \text{ such that } (a) e.\text{axis} = '/' \Rightarrow vp \text{ is the parent of } v, \text{ and (b) } e.\text{axis} = '//' \Rightarrow vp \text{ is an ancestor of } v \}$

Intuitively, the PDT elements associated with each QPT node are first the corresponding candidate elements and hence satisfy descendant constraints. Further, the PDT elements associated with the root QPT node are just its candidate elements, because the root node does not have any ancestor constraints; the PDT elements associated with a non-root QPT node have the additional restriction that they must have the parent/ancestors that are PDT elements associated the parent QPT node. For example, in Figure 4.7(a), each PDT element corresponding to “content” must have a parent element that is the PDT element with respect to “review”. Using the definition of PDT elements, we can now formally define a PDT.

**Definition 4.4.3 (PDT).** Given a QPT $Q$, an XML document $D$, a set of keywords $K$, a PDT is a tree $(N, E)$ where $N$ is the set of nodes and $E$ is set of edges, which are defined as follows.

- $N = \bigcup_{q \in Q} PE(q, D)$, and nodes in $N$ are associated with required values, tf values and byte lengths.
- $E = \{(p, c) \mid p, c \text{ are in } N \land p \text{ is an ancestor of } c \land \exists q \in N \text{ s.t. } p \text{ is an ancestor of } q \text{ and } q \text{ is an ancestor of } c\}$

### 4.4.2 Proposed Algorithms

We now propose our algorithm for efficiently generating PDTs. The generated PDTs satisfy all restrictions described above and contains selectively materialized element
values. The main feature of our algorithm is that it issues a fixed number of index lookups in proportion to the size of the query, not the size of the underlying data, and only makes a single pass over the relevant path and inverted lists indices.

At a high level, the development of the algorithm requires solving three technical problems. First, how do we minimize the number of index accesses? Second, how do we efficiently materialize required element values? Finally, using the information gathered from indices, how do we efficiently generate the PDTs? We describe our solutions to these problems in turn in the next two sections.

**Optimizing index probes and retrieving join values**

To retrieve Dewey IDs and element values required in PDTs, our algorithm invokes a fixed number of probes on path indices. First, we issue index lookups for nodes in QPT that do not have mandatory child edges; note that this includes all the leaf nodes. The elements corresponding to these nodes could be part of the PDT even if none of its descendants are present in the PDT according to the definition of mandatory edges [36]. For instance, for the book QPT shown Figure 4.7(a), we only need to perform three index lookups on path indices (shown in Figure 4.3) for three paths in QPT: books//book/isbn, books//book/year, and books//book/title.

Second, for nodes with ‘v’ annotation, we issue separate lookups to retrieve their data values (which may be combined with the first round of lookups). The idea of retrieving values from path indices is inspired by a simple yet important observation that path indices already store element values in (Path, Value) pairs. Our algorithm conveniently propagates these values along with Dewey IDs. For example, consider the QPT of the book document in Figure 4.7(a) and the path indices in Figure 4.3. For the path books//book/isbn, we use its path to look up the B+-tree index over (Path, Value)
Figure 4.8: Retrieving IDs and values

pairs in the Path-Values table to identify all corresponding values and Dewey IDs (this can be done efficiently because Path is the prefix of the composite key, (Path, Value)); in Figure 4.3, we would retrieve the second and third rows from the Path-Values table. Note that IDs in individual rows are already sorted. We then merge the ID lists in both rows and generate a single list ordered by Dewey IDs, and also associate element values with the corresponding IDs. For example, the Dewey ID 1.1.1 will be associated with the value “111-111-1111”. Finally, our algorithm also return the relevant inverted index
indices to obtain scoring information.

Figure 4.8 shows the high-level pseudo-code of our algorithm of retrieving Dewey IDs, element values and tf values. The algorithm takes a QPT, Path Index, query keywords, and Inverted Index as input, and first issues a lookup on path indices for each QPT node that has no mandatory child edges (lines 5-13). It then identifies nodes that have a \( v \) annotation (lines 9 & 16), and for each path from the root to one of these nodes, the algorithm issues a query to obtain the values and IDs (by only specifying the path). Finally, the algorithm looks up inverted lists indices and retrieves the list of Dewey IDs containing the keywords along with tf values (lines 20-22). Figure 4.9 shows the output of PrepareList for the book QPT (Figure 4.7(a)). Note that the ID lists corresponding to \( books//book/isbn \) and \( books//book/year \) contain element values, and the ID lists retrieved from inverted lists indices contain tf values.

**Efficiently generating PDTs**

In this section we propose a novel algorithm that makes a single “merge” pass of the lists produced by PrepareList and produces the PDT. The PDT satisfies the mutual constraints (determined using Dewey IDs in pathLists) and contains selectively materialized element values (obtained from pathLists) and tf values w.r.t each query keyword (obtained from invLists). For our running example, our algorithm would produce the PDT shown in Figure 4.7(b) by merging the lists shown in Figure 4.9.
The main challenges in designing such an algorithm are: (1) we must enforce complex ancestor and descendant constraints (described in Section 4.4.1) by scanning the lists of Dewey IDs only once, (2) ancestor/descendant axes may expand to full paths consisting of multiple IDs matching the same QPT nodes, which adds additional complication to the problem.

The key idea of the algorithm is to process ids in Dewey order. By doing so, it can efficiently check descendant restrictions because all descendants of an element will be clustered immediately after that element in pathLists. Figure 4.10 shows the high-level pseudo-code of our algorithm which works as follows. The algorithm takes in an QPT, path index and inverted index of the document, and begins by invoking PrepareList to collect the ordered lists of ids relevant to the view. It then initializes the Candidate Tree (described in more detail shortly) using the minimum ID in each list (lines 4-6). Next,
the algorithm makes a single loop over the IDs in pathLists (lines 7-15), and creates PDT nodes using information stored in the CT. At each loop, the algorithm processes and removes the element corresponding to the minimum ID in the CT. Before processing and removing the element, it adds the next ID from the corresponding path list (lines 8-12) so that we maintain the invariant that there are at least one ID corresponding to each relevant QPT node for checking descendant constraints.

Next the algorithm invokes the function CreatePDTNodes (line 14) and check if the minimum element satisfies both ancestor and descendant constraints. If it does, we will create it in the result PDT. If it satisfies only descendant constraints, we store it in a temporary cache (PdtCache) so that we can check the ancestor constraints in subsequent loops. If it does not satisfies descendant constraints and does not have any children in the current CT, we discard it immediately. The intuition is that in this case, since the CT already contains at least one ID for each relevant QPT node (by the invariant above), and since IDs are retrieved from pathList in Dewey order, we know there do not exist more of its descendant IDs in pathLists and hence it will not satisfy descendant constraints in all subsequent loops. The algorithm exits the loop and terminates after exhausting IDs in pathList and the result PDT contains all and only IDs that satisfy the PDT specifications.

We now describe the Candidate Tree and individual steps of the algorithm in more detail.

**Description of the Candidate Tree**

The Candidate Tree, or the CT, is a tree data structure which consists of candidate nodes for the result PDT. Every CT node $cn$ stores sufficient information for efficiently checking ancestor and descendant constraints and has the following five components.

- **ID**: the unique identifier of $cn$, which always corresponds to a prefix of a Dewey ID in pathLists.
AddCTNode(CTNode parent, DeweyID id, int depth)

newNode ← null

if depth ≤ id.Length then
  curId ← Prefix(id, depth), qNode ← QPTNode(curId)
  if qNode = null then AddCTNode(parent, id, depth+1)
else
  newNode ← parent.findChild(curId)
  if newNode = null then
    newNode ← parent.addChild(curId, qNode)
  end if
  Update the data value and tf values if required
  AddCTNode(newNode, id, depth+1)
end if

if newNode ≠ null ∧ ∀i, newNode.DM[i]=1 then
  ∀ n ∈ newNode.PL, n.DM[newNode.QPTNode] ← 1
end if

Figure 4.11: Algorithm for adding new CT nodes

- QNode: the QPT node to which cn.ID corresponds.
- ParentList (or PL): a list of cn’s ancestors whose QNode’s are the parent node of cn.QNode.
- DescendantMap (or DM): QNode → bit: a mapping containing one entry for each mandatory child/descendant of cn.QNode. For a child QPT node c, DM[c] = 1 iff cn has a child/descendant node is a candidate element with respect to c.
- PdtCache: the cache storing cn’s descendants that satisfy descendant restrictions and whose ancestor restrictions are yet to be checked.

We now illustrate these components using CT shown in Figure 4.13(a), which is created using IDs 1.1.1, 1.1.4, and 1.2.6, corresponding to paths in pathLists shown in Figure 4.9. First, every node has an ID and a QNode and CT nodes are ordered based
CreatePDTNodes (CTNode n, QPT qpt, PDT parentPdtCache)

if $\forall i$, n.DM[i] = 1 \land n.ID not in parentPdtCache then
  pdtNode = parentPdtCache.add(n)
end if

if n.HasChild() = true then
  CreatePDTNodes(n.MinIdChild, qpt, n.PdtCache)
else
  {Handle pdt cache and then remove the node itself}
  for x in n.pdtCache do
    {Update parent list and then propagate x to parentPdtCache}
    if n $\in$ x.PL then
      x.PL.remove(n)
      if $\exists i$, n.DM[i] = 0 \land x.PL = $\emptyset$ then n.pdtCache.remove(x)
    else
      x.PL.replace(n, n.PL)
    end if
  end for
  if x $\in$ pdtCache then Propagate x to parentPdtCache
end if
n.RemoveFromCT()
end if

Figure 4.12: Processing CT.MinIDPath

on their IDs. For example, the ID of the “books” node is 1 which corresponds a prefix of the ID 1.1.1, and the id 1.1.1 corresponds to the QPT node “isbn”. The PL of a CT node stores its ancestor nodes that correspond to the parent QPT node. For instance, book1.PL = \{books\}. Note that cn.PL may contain multiple nodes if cn.QNode is in an ancestor/descendant relations. For example, if “/books//book” expands to “/books/books/book”, then book.PL would include both “books”. Next, DM keeps track of whether a node satisfies descendant restrictions. For instance, book1.DM[year] = 0 because it does not have the mandatory child element “year” while book2.DM[year] = 1 because it does. Consequently, a CT node satisfies the descendant restrictions (and therefore
is a candidate element) when its DM is empty (corresponding to QPT nodes without mandatory child edges), or the values in its DM are all 1 (corresponding to QPT nodes with mandatory child edges). PdtCache will be illustrated in subsequent steps shortly.

Note that for ease of exposition, our illustration focuses on creating the PDT hierarchy; the atomic values and tf values are not shown in the figure and bear in mind that they will be propagated along with the corresponding Dewey IDs.

**Initializing the Candidate Tree**

As mentioned earlier, the algorithm begins by initializing the CT using minimum IDs in pathLists. Figure 4.11 shows the pseudo-code for adding a single Dewey ID and its prefixes to the CT. A prefix is added to the CT if it has a corresponding QPT node and it is not already in the CT (lines 6-13). In addition, if a prefix is associated with a ’c’
annotation, the $tf$ values are retrieved from the corresponding inverted lists (line 10).

Figure 4.13(a), which we just described, is the initial CT for our running example, which is created by adding minimum IDs of paths in pathLists shown in Figure 4.9. Note that for ease of exposition, our algorithm assumes each Dewey ID corresponds to a single QPT node; however, when the QPT contains repeating tag names, one Dewey ID can correspond to more than one QPT nodes. We discuss how to handle this case in Section 4.4.2.

**Description of the main loop**

Next the algorithm enters the loop (lines 7-15 in Figure 4.10) which adds new Dewey IDs to the CT and creates PDT nodes using CT nodes. At each loop, the algorithm ensures the following invariant: the Dewey IDs that are processed and known to be PDT nodes are either in the CT or in the result PDT (hence we do not miss any potential PDT nodes); and the result PDT only contains IDs that satisfy the PDT specifications.

As mentioned earlier, at each loop we focus on the element corresponding to the minimum ID in the CT and its ancestors (denoted by MinIDPath in the algorithm). Specifically, we first retrieve next minimum IDs corresponding to QPT nodes in MinIDPath (Step 1). We then copy IDs in MinIDPath from top down to the result PDT or the PDT cache (Step 2). Last, we remove those nodes in MinIDPath that do not have any children from bottom up (Step 3). We now describe each step in more detail.

**Step 1: adding new IDs**

In this step, the algorithm adds next IDs corresponding to the QPT nodes in CT.MinIDPath. In Figure 4.13(a), this path is “books//book/isbn” and Figure 4.13(b) shows the CT after its next minimum ID 1.2.1 is added (for reason of space, this figure and the rest only show the QPT node and ID).

**Step 2: creating PDT nodes**

In this step, the algorithm creates PDT nodes using CT nodes in CT.MinIDPath from top down (Figure 4.12, lines 2-4). We first check if
the node satisfies the descendant constraints using values in its DM. In Figure 4.13(b), DM of the element “books” has value 1 in all entries, hence we will create its ID in the PDT cache passed to it(lines 2-4), which is the result PDT.

The algorithm then recursively invokes CreatePDTNodes on the element book1 (line 6). Its DM has value 0 and hence it is not a PDT node yet. Next, we find its child element “isbn” has an empty DM and satisfies the descendant restrictions. Hence we create the node “isbn” in book1.PdtCache. Figure 4.13(c) illustrates this step. In general, the pdt cache of a CT node stores its descendants that satisfy the descendant restrictions, and the checking of the ancestor restrictions is deferred until the node itself is being removed (in Step 3).

**Step 3: removing CT nodes**  After the top down processing, the algorithm starts removing nodes from bottom up(Figure 4.12, line 7-20). For instance, in Figure 4.13(c), after we process and remove the node “title”, we will remove the node “book” because it does not have children and it does not satisfy descendant constraints. Figure 4.13(d) shows the CT at this point. Such node can be removed because as mentioned earlier, we can be certain that it will not satisfy the descendant restrictions (as in our example).

Another key issue we consider before removing a node is to handle nodes in its pdt cache. In our example, the pdt cache contains two nodes “isbn” and “title”. As mentioned earlier, they both satisfy descendant constraints. Hence we only need to check if they satisfy ancestor constraints, which is done by checking nodes in their parent lists. If those parent nodes are known to be non-PDT nodes, which is the case for “isbn” and “title”, then we can conclude the nodes in the cache will not satisfy ancestor restrictions at all, and therefore can be removed (line 13). Otherwise the cache node still has other parents (which can be PDT nodes), and will be propagated to the pdt cache of the ancestor. Figure 4.7(e) and (f) illustrates this case in our running example, which
occurs when we remove the node “book” with ID 1.2.

In summary, we remove a node (and its ID) only when it is known to be a non-PDT node, which is either a CT node that does not satisfy descendant constraints, or a node in a pdt cache that does not satisfy ancestor constraints. Further, we only create nodes satisfying descendant constraints in the pdt cache, and always check ancestor constraints before propagating them to ancestors in the CT. Therefore it is easy to verify the the invariant of the main loop holds. Finally, at the last step of the algorithm when we remove the root node “books”, all IDs in its pdt cache will be propagated to the result PDT.

Extensions and optimizations  As mentioned earlier, when the QPT has repeating tag names, a single Dewey ID can match multiple QPT nodes. For example, if the QPT path is “//a//a” and the corresponding full data path is “/a/a/a”, then the second a in the full path matches both nodes in the QPT path. To handle this case, we extend the structure of CT node to contain a set of QNodes, each of which is associated with their own InPdt, PL and DM. This is because in general different QPT nodes capture different ancestor/descendant constraints, hence must be treated separately.

Further, there are two possibilities of optimizations in the current algorithm. First, the algorithm always creates and propagates IDs that satisfy the descendant constraints in the pdt cache. This can be optimized by immediately creating the IDs in the result PDT if they also satisfy the ancestor restrictions. For this purpose, we add a boolean flag InPdt to the CT node, set InPdt to be true when the ID is created in the result PDT, and create the descendant ID in the PDT when one of its parents is in the PDT (InPdt = true). Second, to optimize the memory usage, we enforce the PDT nodes to be output in document order (to external storage). We refer the reader to Section 4.7.1 for complete
details and corresponding revisions to our algorithm.

**Scoring & generating the results**

Once the PDTs are generated, they are fed to a traditional evaluator to produce the temporary results (Figure 4.6). The temporary results produced by evaluating the PDTs in Figure 4.7(b) is shown in Figure 4.14. Then, as mentioned in Section 4.2.1, the Scoring Module uses these pruned results to enforce the conjunctive or disjunctive semantics and calculate the scores. Specifically, for a view result $s$, score($s$) is computed as follows: first calculate $tf(s, k)$ for a keyword $k$ by aggregating values of $tf(s', k)$ of all relevant base elements $s'$; then calculate the value $idf(k)$ by counting the number of view results containing the keyword $k$; next use the formula in Section 4.1.2 to obtain...
the non-normalized scores, which are then normalized using aggregate byte lengths of the relevant base elements. Note that in Figure 4.14, base elements contained in view results have all tf values and byte lengths required in scoring. Then we only select the results with top-k scores. The last step is to convert the candidate results with IDs to text output. It is only in this last phase that the base data is accessed (based on the Dewey ID of content nodes) for the top few result to produce the final output. Figure 4.15 shows the final results.

4.4.3 Complexity and Correctness of Algorithms

The runtime of GeneratePDT is \( O(Nqd + Nqd^2 + Nd^3 + Ndkc) \) where \( N \) is the number of the IDs in pathLists, \( d \) is the depth of the document, \( q \) and \( f \) are the depth and fan-out of the QPT, respectively, \( k \) is the number of keywords, and \( c \) is the average unit cost of retrieving tf values. Intuitively, the top-down and bottom-up processing dominate the overall cost. \( Nqd + Nqd^2 \) determines the cost of the top-down processing: there can be \( Nd \) ID prefixes; every prefix can correspond to \( q \) QPT node; every QPT node can have \( d \) parent CT nodes and \( f \) mandatory child nodes. \( Nd^3 \) determines the cost of bottom-up processing, since every prefix can be propagated \( d \) times and can have \( d \) nodes in its parent list. Finally, \( Ndkc \) determines the cost of retrieving tf values from the inverted index.

Note that this is a worst case bound which assumes multiple repeating tags in queries (\( q \) QPT nodes), and repeating tags in documents (\( d \) parent nodes). In most real-life data, these values are much smaller (e.g., DBLP\(^2\), and SIGMOD Record\(^3\), and INEX), as also seen in our experiments.

\(^2\)http://dblp.uni-trier.de/xml/
\(^3\)http://acm.org/sigmod/record/xml/
We can prove the following correctness theorem (proofs are presented in Section 4.7). If \( I \) is the function transforming Dewey IDs to node contents, PDTTF is the \( tf \) calculation function, and PDTByteLength is the byte length calculation function, \( \text{len}(e) \) is the byte length of a materialized element \( e \), and using the notations of UD, Q, S defined in Section 2.1.

**Theorem 4.4.4** (Correctness). Given a set of keywords \( KW \), an XQuery query \( Q \) and a database \( D \in UD \), if \( PDTDB = \{ \text{GeneratePDT}(QPT, D.\text{PathIndex}, D.\text{InvertedIndex}, KW) | QPT \in \text{GenerateQPT}(Q) \} \), then

- \( I(Q(PDTDB)) = Q(D) \) (The result sequences, after being transformed, are identical)
- \( \forall e \in Q(PDTDB), e' \in Q(D), I(e) = e' \Rightarrow PDTByteLength(e) = \text{len}(e') \) (The byte lengths of each element are identical)
- \( \forall e \in Q(PDTDB), e' \in Q(D), I(e) = e' \Rightarrow (\forall k \in KW, PDTTF(e,k) = tf(e',k)) \) (The term frequencies of each keyword in each element is identical)

### 4.5 Experiments

In this section, we show the experimental results of evaluating our proposed techniques developed in the Quark open-source XML database system.

#### 4.5.1 Experimental setup

In our experiments, we used the 500MB INEX dataset which consists of a large collection of publication records. The DTD relevant to our experiments are shown in Figure 4.16.
Table 4.1: Experimental parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values (default in bold)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of Data ($\times 100 MB$)</td>
<td>1, 2, 3, 4, 5</td>
</tr>
<tr>
<td>Avg. Size of View Element</td>
<td>1X, 2X, 3X, 4X, 5X</td>
</tr>
<tr>
<td># keywords</td>
<td>1, 2, 3, 4</td>
</tr>
<tr>
<td>Selectivity of keywords</td>
<td>Low (IEEE, Computing),</td>
</tr>
<tr>
<td></td>
<td>Medium (Thomas, Control),</td>
</tr>
<tr>
<td></td>
<td>High (Moore, Burnett)</td>
</tr>
<tr>
<td># of joins</td>
<td>0, 1, 2, 3, 4</td>
</tr>
<tr>
<td>Join selectivity</td>
<td>1X, 0.5X, 0.2X, 0.1X</td>
</tr>
<tr>
<td>Level of nestings</td>
<td>1, 2, 3, 4</td>
</tr>
<tr>
<td># of results (K in top-K)</td>
<td>1, 10, 20, 40</td>
</tr>
</tbody>
</table>

We created a view in which publications are nested under their authors, and evaluated our system using this view. When running experiments, we generated the regular path and inverted lists indices implemented in Quark (~1GB each), and also generated the index for storing element lengths for score normalization as described in Section 4.4.2 (~128MB).

Our experimental setup was characterized by parameters in Table 4.1. *Avg. size of element* specifies the average size of each element; we vary this parameter by replicating the content of view elements. *# of joins* is the number of value joins in the view. *Join selectivity* characterizes how many articles are joined with a given author; the default value 1X corresponds to the entire 500MB data; we decrease the selectivity by replicating subsets of the data collection. *Level of nestings* specifies the number of nestings of FLOWR expressions in the view; for value 1, we remove the value join and only leave the selection predicate; for the default value 2, we associate publications under
authors; for the deeper views, we create additional FLOWR expressions by nesting the view with one level shallower under the authors list. The rest of the parameters are self-explanatory. In the experiments, when we varied one parameter, we used the default values for the rest. We evaluated four alternative approaches, described as follows:

**BASELINE**: materializing the view at the query time, and evaluating keyword search queries over view implemented using Quark.

**GTP**: GTP with TermJoin for keyword searches and implemented using Timber [8].

**EFFICIENT**: our proposed keyword query processing architecture (Section 4.2.1) implemented using Quark.

**PROJ**: techniques of projecting XML documents [85].

The experiments were run on a machine with a 3.4Ghz P4 CPU and 2GB memory running Windows XP. The reported results are the average of five runs.

### 4.5.2 Performance results

**Varying size of data**

Figure 4.17 shows the performance results when varying the size of the data. As shown, it only took EFFICIENT less than 5 seconds to evaluate a keyword query without materi-
alizing the view over the 500MB data. Second, the run time increases linearly with the size of the data (note that the y-axis is in log scale), because the I/O cost (determined by the size of the query, as described in Section 4.4.2) and the overhead of query processing increases linearly. This indicates EFFICIENT is a scalable and efficient solution.

In contrast, BASELINE took 59 seconds even for a 13MB data set, which is more than an order of magnitude slower than EFFICIENT. Note the run time includes 58 seconds spent on materializing the view, and 1 second spent on the rest of query evaluation,
including tokenizing the view and evaluating the keyword search query.

Further, Figure 4.17 shows that EFFICIENT performed $\sim$10 times faster than GTP. Note that Figure 4.17 only shows the time spent on structural joins and accessing the base data (for obtaining join values) in GTP. We did not report the overhead of the rest of query evaluation because they were inefficient and did not scale well (in fact, on the 100MB data set, the total running time for GTP, including the time to perform the value join, was more than 5 minutes). GTP is much slower mainly because it relied on (expensive) structural joins to generate the document hierarchy and must access base data to obtain join values.

Finally, while PROJ merely characterizes the cost of generating projected documents (the cost of query processing and post-processing are not included), its runtime is $\sim$15 times slower than EFFICIENT. The main reason is that PROJ scans full documents which leads to relatively poor scalability.

For the rest of the experiments, we only show the results on EFFICIENT since other alternatives performed significantly slower.
Evaluating overhead of individual modules

Figure 4.18 breaks down the run time of EFFICIENT and shows the overhead of individual modules – PDT, Evaluator, and Post-processing. As shown, the cost of generating PDTs scales gracefully with the size of the data. Second, the overhead of post-processing, which includes scoring the results and materializing top-K elements, is negligible (which can be barely seen in the graphs). The most important observation is that the cost of the query evaluator dominates the entire cost when the size of the data increases.
Varying other parameters

**Varying avg. size of the view element:** Figure 4.19 shows the performance results when varying the size of the view element. The run time slightly increases mainly because the overhead of post-processing increases when materializing elements of larger sizes, and the rest of the query processing remains approximately the same.

**Varying # of keywords:** Figure 4.20 shows the performance results when varying the number of keywords. The run time slightly increases because the algorithm accessed a larger number of inverted lists to retrieve tf values, which introduces additional overhead when generating PDTs.

**Varying selectivity of keywords:** Figure 4.21 shows the performance results when varying the selectivity of the keywords. The run time increases slightly when the selectivity of keywords decreases. This is mainly because the overhead of generating PDTs increases – as the selectivity goes down, the length of the inverted list becomes larger which increases the I/O cost of retrieving tf values.

**Varying # of joins:** Figure 4.22 shows the performance results when varying the number of value joins in the view definition. As shown, the run time increases with the number of joins mainly because the cost of the query evaluation increases. The run time increases most significantly when the number of joins increases from 0 to 1 for two reasons. First, the case of 0 joins only requires generating a single PDT while the other requires two. More importantly, the cost of evaluating a selection predicate (in the case of 0 joins) is much cheaper than evaluating value joins.

**Varying the selectivity of joins:** Figure 4.23 shows the performance results when varying the selectivity of value joins in the view definition. As shown, the run time increases slightly when the selectivity decreases mainly because the cost of the query evaluation increases.
Varying the level of nestings: Figure 4.24 shows the performance results when varying the level of nestings in the view. This experiment shows that the run time increases linearly with the level of nestings, while the overhead of the query evaluator grows relatively faster than other modules.

Varying the number of results: Figure 4.25 shows the performance results when varying the number of results (i.e., K in top-K). As shown, the run time remains approximately the same because the overhead of storing and materializing additional results is nearly negligible.

4.6 Additional details of GenerateQPT

In this section, we show the complete details of GenerateQPT, and prove its correctness.

4.6.1 Complete Algorithm

In our system, we design and implement a QPT algorithm GenerateQPT() similar to the GTP algorithm except that we additionally handle the ’c’ and ’v’ node annotations and we support a larger XQuery grammar that includes functions.

We now present an algorithm to generate QPTs for a given view query. The main challenge lies in correctly determining the shape of the tree and the associated annotations for arbitrarily complex views that conform to the grammar specified earlier.

Our algorithm (Figure 4.26) works as follows. The recursive function GenerateQPT takes in the current expression (e) and returns a set of QPTs generated. The algorithm is initially invoked with $e$ set to be the expression that defines the view. When processing an expression, the algorithm also sets the node annotation for the nodes generated in QPTSet indicating whether the corresponding expression contributes to the content of
Figure 4.26: Algorithm for producing Query Pattern Tree (QPT) from a keyword query
1: GenerateQPT (Expr e) : QPTSet
2: if e is PathTailExpr then
3:   if e is 'TAGNAME' then
4:     {Create a '.' node with a child node for 'TAGNAME'}
5:     root ← ('.',{})
6:     child ← (TAGNAME,{})
8:     return ((root, child),{(root, child, ';', 'm')}, root)
9:   else if e is 'TAGNAME' / PathTailExpr then
10:    root ← ('.',{})
11:    child ← (TAGNAME,{})
12:    V ← {root, child}
13:    E ← {(root, child, '/', 'm')}
14:    for all tempQpt in GenerateQPT(PathTailExpr) do
15:       for all (tempQpt.root, n, axis, ann) in tempQpt.E do
16:         E.add(child, n, axis, ann); V .add(n);
17:       end for
18:    end for
19:    return {(V,E,root)}
20: else
21:   {e is 'TAGNAME' // PathTailExpr}
22:   {Similar to 'TAGNAME' // PathTailExpr}
23: end if
24: else if e istype 'if' Expr1 'then' Expr2 else 'Expr3' then
25:   for all Q ∈ GenerateQPT(Expr1), n ∈ Q.V do
26:     C-AnnMap[n] = false;
27:   end for
28:   return GenerateQPT(Expr1) ∪ GenerateQPT(Expr2) ∪ GenerateQPT(Expr3)
29: else if e istype PredExpr then
30:   if e is PathExpr then
31:     return GenerateQPT(PathExpr)
32:   else if e is PathExpr Comp Literal then
33:     pathset = GenerateQPT(PathExpr)
34:     for all pathqpt in pathset do
35:       for all leaf nodes oldnode=(name,pred) in pathqpt do
36:         newnode = (name,pred ∪ {'Comp Literal'})
37:         V-AnnMap[newnode] = false,
38:         C-AnnMap[newnode] = C-AnnMap[oldnode]
39:         tempqpt.V .replace(oldnode, newnode)
40:       end for
41:     end for
42:     return pathset
43:   else
44:     {e is PathExpr1 Comp PathExpr2}
45:     pathset1 = GeneratePST(PathExpr1)
46:     pathset2 = GeneratePST(PathExpr2)
47:     for all leaf node l in pathset1 ∪ pathset2 do
48:       V-AnnMap[l] = true
49:       C-AnnMap[l] = false
50:     end for
51:     return pathset1 ∪ pathset2
52:   end if
53: end if

Figure 4.27: Algorithm for producing QPT for PathTailExpr & PredExpr
Now we use our running example to walk through the algorithm. For ease of exposition, we unfold the recursive call and illustrate the construction of QPTs from bottom up. Figure 4.28 shows the process of creating the QPT for nodes in books.xml at each phase. Initially, we call lines 4-6 in Figure 4.29 and generates the PDT for the expression “$book/year > 1995”. Figure 4.28(a) show the QPT at this point. Note that by line 36 in Figure 4.27, the predicate is now associated with the leaf QPT node. Next, we generate the QPT for the expressions in the return clause (line 9 in Figure 4.29). As shown in Figure 4.28(b), two additional twigs are created with optional edges. The intuition is that by the semantics of FLOWR expression, the existence of the parent element “$book” does not depend on the existence of “isbn” or “title”. This is in contrast to the edge create in step 1 in which case the existence of “$book” is restricted by the given predicate. Further, we indicate that the value of “isbn” is required since it is used in a predicate; and the content of “title” is required since it is part of the view results. Next, we generate the QPT for the path expressions in the for clause, and the resulting QPT is shown in Figure 4.28(c). Finally, we bind the set of QPTs that generated using the
GenerateQPT (Expr e) : QPTSet

1: if e is type FLWORExpr then
2:  qptSet ← ∅
3: if FLWORExpr.WhereClause is present then
4:  PredExpr ← FLWORExpr.WhereClause.PredExpr
5:  ∀ node ∈ qptSet, C-AnnMap[node] = false
6:  qptSet ← GenerateQPT(PredExpr)
7:  RetExpr ← FLWORExpr.ReturnClause.RetExpr
8:  qptSet ← qptSet ∪ GenerateQPT(RetExpr)
9: end if
10: for all forLetClause in forLetClauses do
11:  VAR ← forLetClause.VAR;
12:  pathSet ← GenerateQPT(forLetClause.PathExpr)
13:  for all pathQpt in pathSet where prevQpt.root is VAR do
14:    E' ← qpt.E with all (qpt.root, n, axis, ann) edges replaced with ( qpt.root, n, axis, 'o')
15:  end for
16:  tempset.add(currSet.V, E', currSet.root)
17: end for
18: if forLetClause is ForClause then
19:  VAR ← forLetClause.VAR;
20:  pathSet ← GenerateQPT(forLetClause.PathExpr)
21: end if
22: for all RetExpr1 in RetExprList do
23:  currset ← GenerateQPT(RetExpr1)
24:  for all qpt in currset where qpt.root is VAR do
25:    E' ← qpt.E with all (qpt.root, n, axis, ann) edges replaced with (qpt.root, n, axis, 'o')
26:  end for
27:  tempset.add(currSet.V, E', currSet.root)
28: end for
29: for all Expr1 in e do
30:  currset ← GenerateQPT(Expr1)
31:  for all qpt in currset where qpt.root is VAR do
32:    E' ← qpt.E with all (qpt.root, n, axis, ann) edges replaced with (qpt.root, n, axis, 'o')
33:  end for
34:  tempset.add(currSet.V, E', currSet.root)
35: end for
36: else
37:  if e is Expr then GenerateQPT(Expr)
38:  else if e is '<' TAGNAME ' />' RetExprList '<' TAGNAME ' />' then
39:    tempset ← ∅
40:    for all RetExpr1 in RetExprList do
41:      currset ← GenerateQPT(RetExpr1)
42:      for all qpt in currset where qpt.root is VAR do
43:        E' ← qpt.E with all (qpt.root, n, axis, ann) edges replaced with (qpt.root, n, axis, 'o')
44:      end for
45:      tempset.add(currSet.V, E', currSet.root)
46:    end for
47:    end if
48: end if
49: return QPTSet

Figure 4.29: Algorithm for producing QPT: FLWORExpr
where clause and the return clause to the variable in the for clause. In our example, we simple replace the node “$book” with the leaf node “//book” in Figure 4.28(c) and Figure 4.28(d) shows the final QPT. Note that the C-Annotation of the leaf node “//book” in Figure 4.28(c) is changed to false because the algorithm determines that it is not part of the view results (lines 17 -22 in Figure 4.29.

4.6.2 Proofs of correctness

In this section we prove the correctness of our QPT generation algorithms. We first prove that for a query expression $E$ which conforms to the core XQuery grammar, $\text{GenerateQPT}(E)$ generates the correct set of QPTs (Theorem 4.6.1). We then show that $\text{GenerateQPT}(E) = \text{GenerateQPT}(E')$ where $E$ is a query expression that conform to our grammar and $E'$ is the corresponding normalized query in the core grammar (Theorem 4.6.6).

We first introduce some notation. We use the XQuery formal semantics [2] for evaluating queries, and we use $\text{Env} \vdash E \Rightarrow V$ to denote that in the evaluation context $\text{Env}$, the query expression $E$ evaluates to the value $V$. For notational convenience, we also use $\text{Eval}(E, \text{Env})$ to denote $V$. Note that $\text{Env}$ captures both static context and dynamic context used in the formal semantics.

Further, in our post-processing, we say $I$ is the function transforming Dewey IDs to node contents in the database, PDTTF is the tf calculation function, and PDTByteLength is the byte length calculation function, and $\text{len}(e)$ is the byte length of a materialized element $e$, then we can prove the correctness of $\text{GenerateQPT}$ in Theorem 4.6.1

**Theorem 4.6.1** (Correctness of GenerateQPT). *Given a set of keywords $KW$, an XQuery query expression $E$ that conforms to the core grammar, a database instance $D$, then $\forall \delta \in UE(D, \text{FreeVars}(E))$*
(a) \( I(\text{Eval}(E, \{Q.\text{root} \Rightarrow \text{PDT}(Q, \text{KW}, \delta) \mid Q \in \text{GenerateQPT}(E)\})) = \text{Eval}(E, \delta) \) (The result sequences, after being transformed, are identical)

(b) \( \forall e \in \text{Eval}(E, \{Q.\text{root} \Rightarrow \text{PDT}(Q, \text{KW}, \delta) \mid Q \in \text{GenerateQPT}(E)\}), e' \in \text{Eval}(E, \delta), I(e) = e' \Rightarrow \text{PDTByteLength}(e) = \text{len}(e') \) (The byte lengths of each element are identical)

(c) \( \forall e \in \text{Eval}(E, \{Q.\text{root} \Rightarrow \text{PDT}(Q, \text{KW}, \delta) \mid Q \in \text{GenerateQPT}(E)\}), e' \in \text{Eval}(E, \delta), I(e) = e' \Rightarrow (\forall k \in \text{KW}, \text{PDTTF}(e, k) = \text{tf}(e', k)) \) (The term frequencies of each keyword in each element is identical)

Before proving the lemma, we first prove two supporting lemmas.

If \( \text{Leaf}(Q) \) is the set of the leaf nodes in a QPT \( Q \), then we first show Lemma 4.6.2 showing that \( \text{GenerateQPT}(	ext{PathExpr}) \) is a singleton set and it has only one leaf node.

**Lemma 4.6.2** (GenerateQPT(PathExpr)). Given a path expression \( E \),

- \( |\text{GenerateQPT}(E)| = 1, \) and
- \( \forall Q \in \text{GenerateQPT}(E), |\text{Leaf}(Q)| = 1. \)

*Proof. Sketch* We show the lemma by structural induction on \( E \).

**Base case:** \( E = \text{fn:doc(Name)} \) or \( \text{VAR} \) or \( '.' \) By line 7 in Figure 4.26, it is easy to see that the base case holds.

**Inducting hypothesis:** Suppose Lemma 4.6.2 holds for sub-expressions of \( E \), now we show it holds for \( E \).

There are several cases and their proofs are similar. Now we only show for \( E = \text{fn:doc(Name)} 'l' \) PathTailExpr.

First, by I.H., we know that \( |\text{GenerateQPT}(	ext{PathTailExpr})| = 1 \) and \( |\text{Leaf}(Q')| = 1 \) where \( \{Q'\} = \text{GenerateQPT}(	ext{PathTailExpr}) \). We also know that \( |\text{GenerateQPT} (\text{fn:doc(Name)})| = 1 \) and \( |\text{Leaf}(Q'')| = 1 \) where \( \{Q''\} = \text{GenerateQPT} (\text{fn:doc(Name)}) \).
Then by lines 11-16 in Figure 4.26, we know that $\text{GenerateQP} \ E = \{Q\}$ where $Q.V = Q''.V \cup Q'.V - \{Q'.root\}, \ Q.E = Q''.E - \{(Q''.root, x, axis, ann) \mid x \in Q''.V\}$ $\cup \{(Q'.root, x, axis, ann) \mid (Q''.root, x, axis, ann) \in Q''.E\}$, and $\text{Leaf}(Q) = \text{Leaf}(Q')$. 
Hence $|\text{GenerateQP} \ E| = 1$ and $\forall Q \in \text{GenerateQP} \ E, |\text{Leaf}(Q) = 1|$.

Lemma 4.6.3 (Mandatory Child Edges). Given a query expression $E$, an XML database $D$, $(\forall \delta \in U.E(D, \text{FreeVars}(E)), \exists Q \in \text{GenerateQP} \ E, c \in Q.V, r \in \text{Nodes}(D)$ $(\delta(Q.root) = r \land (Q.root, c', l', m') \in Q.E \land (\nexists n \in CE(c, r), \text{parent}(r, n))) \Rightarrow \text{Eval}(E, \delta) = ()$.

Proof. Sketch We prove Lemma 4.6.3 by structural inductions on $E$.

Base case 1: $E = \text{fn:doc(Name)}$ or $\text{VAR}$ or $'.'$. This case is handled by lines 5-7 in Figure 4.26. It is easy to see that $E_q = \emptyset$. Therefore the lemma is vacuously true.

Base case 2: $E = \text{TAGNAME}$ This case is handled by lines 3-8 in Figure 4.27. It is easy to see that $\text{GenerateQP} \ E$ is a singleton set. Assume $\{Q\} \in \text{GenerateQP} \ E}, then by the algorithm we know $Q.V = \{Q.root, l\}$ where $l$ is the leaf node and $l.name = \text{TAGNAME}$.

By definition, if $n$ is not in $CE(l, r)$, then $n.Name \neq \text{TAGNAME}$. Therefore if $(\nexists n \in CE(l, r)$ $\text{parent}(r, n))$, we know that $r$ does not have a child node with the tag name $\text{TAGNAME}$.

On the other hand, by the semantics of $E$, $\text{Eval}(\text{TAGNAME}, \delta)$ is evaluated by invoking $\text{NameTest}$ on child nodes of $r$. Since $r$ does not have a child node with the tag name $\text{TAGNAME}$, we can infer that $\text{Eval}(\text{TAGNAME}, \delta) = ()$.

Induction hypothesis: Suppose the lemma holds for sub-expressions of $E$. We now show the lemma also holds for $E$. 
Here we only prove the case \( E = \text{for VAR in PathExpr return Expr} \) as it covers the main points of all other cases.

**Case 1:** \( E = \text{for VAR in PathExpr return Expr} \) This case is handled by lines 9-22 in Figure 4.29. Essentially, we first obtain the set of QPTs corresponding to \( Expr \), then if \( VAR \) is referenced in \( Expr \) (as the root), we bind the \( VAR \) node in \( Expr \) to the leaf nodes of \( \text{GenerateQPT}(\text{PathExpr}) \).

By Lemma 4.6.2, we know that \(|\text{GenerateQPT}(\text{PathExpr})| = 1|\). Assume \( \{P\} = \text{GenerateQPT}(\text{PathExpr}) \), and w.o.l.g., we assume \( \{X\} = \text{GenerateQPT}(\text{Expr}) \).

Now we assume \( \exists Q \in \text{GenerateQPT}(E), \exists c \in Q.V \)

\[
(r_q, c', \%/m') \in Q.E \land \delta(Q.root) = r
\]

\(\land(\exists n \in C(E(c, r) \text{ parent}(r, n)))\), and need to show that \( \text{Eval}(E, \delta) = () \).

There are two cases depending on the value of \( X.\text{root} \).

**Case A:** \( Q.\text{root} \neq VAR \) In this case, \( Expr \) does not reference \( VAR \). Therefore by lines 15-22 in Figure 4.29, we know that \( \text{GenerateQPT}(E) = \{P, X\} \). I.e., \( Q = P \) or \( Q = X \). If \( Q = P \), then by I.H., \( \text{Eval}(P, \delta) = () \). According to the semantics of \( E \), we know that \( \text{Eval}(E, \delta) = () \). Otherwise \( \text{Eval}(P, \delta) \neq () \) and by I.H., \( \text{Eval}(X, \delta) = () \). Again, according to the semantics, \( \text{Eval}(E, \delta) = \text{Eval}(X, \delta) \) and hence \( \text{Eval}(E, \delta) = () \).

**Case B:** \( Q.\text{root} = VAR \) In this case, \( Expr \) does reference \( VAR \). By lines 15-22 in Figure 4.29, we know that \( \text{GenerateQPT}(E) = \{P'\} \) where \( P'.V = P.V \cup X.V - \{X.\text{root}\}, P'.E = P.E \cup X.E - \{(X.\text{root}, l, \text{axis}, \text{ann})\} \)

\(\cup\{(l_p, l, \text{axis}, \text{ann}) \mid (X.\text{root}, l, \text{axis}, \text{ann}) \in X.E\} \) where \( l_p \) is the leaf node in \( P \), and \( P'.\text{root} = P.\text{root} \).

Now if \( l_p = P.\text{root} \), i.e., if \( P \) is a tree with a single node, then \( P = X \) and therefore
we can apply I.H. on X and conclude that $Eval(E, \delta) = ()$.

Otherwise $l_p \neq P.root$. Since $P'.root = P.root$, hence $c \in P$. Now we reason by analyzing the relations of 
$CE(c, r)$ w.r.t. P and P', denoted by $CE_P(c, r)$ and $CE_{P'}(c, r)$, respectively.

First, by definition of candidate elements, we know 
$CE_{P'}(c, r) \subseteq CE_P(c, r)$ since intuitively $P'$ contains all edges in P and has additional 
edges in X. Therefore $\nexists n \in CE_P(c, r) parent(r, n)$ implies that $\nexists n \in CE_{P'}(c, r)$ 
$parent(r, n)$. Therefore we can apply I.H. on PathExpr and conclude that $Eval(PathExpr, \delta) = ()$, and hence according to the semantics of E, $Eval(E, \delta) = ()$.

Hence the lemma holds for E. □

We can similarly show that,

**Lemma 4.6.4** (Mandatory Descendant Edges). Given a query expression $E$, an XML 
database $D$,

$(\forall \delta \in U(E(D, FreeVars(E)), \exists Q \in GenerateQPT(E), c \in Q.V, r \in Nodes(D) 
((Q.root, c, /''', m') \in Q.E \land \delta(Q.root) = r \land (\nexists n \in CE(c, r) parent(r, n))) \Rightarrow 
Eval(E, \delta) = ()$.

Lemma 4.6.3 and Lemma 4.6.4 indicate that if an element corresponding to the root 
of an expression (and its QPT) does not have a mandatory child (descendant), then the 
evaluation results using this element as the context is an empty sequence.

Now we show Theorem 4.6.1(a).

**Proof.** We prove Theorem 4.6.1(a) by structural induction on the query expression. For 
notational convenience, let $\delta' = \{Q.root \Rightarrow PDT(Q, KW, \delta)| Q \in GenerateQPT(E)\}$.

**Base case 1**: $E = fn:doc(Name) or VAR or '.'
This case is handled by lines 3-7 in Figure 4.26. By line 7, we know that GenerateQPT(E)=\{Q\} where \(Q.V = \{Q.root\}\), \(Q.E = \emptyset\) and \(Q.root.name=E\). If \(\delta(Q.root) = d\), then by the formal semantics, we know that \(Eval(E, \delta) = d\). On the other hand, since \(Q.root.name=E\), by definition of \(\delta\) and the formal semantics, we know that \(Eval(E, \{Q.root \Rightarrow PDT(Q, KW, \delta)\}) = PDT(Q, KW, \delta).root\). Since \(\delta(Q.root) = d\), by the definition of PDT, \(PDT(q, KW, \delta).root = d\), and therefore \(Eval(E, \{Q.root \Rightarrow PDT(Q.root, KW, \delta)\}) = d\). Last, by line 6 in Figure 4.26, we know that \(C-AnnMap[Q.root] = true\). Therefore it is easy to see \(I(Eval(E, \{r_q \Rightarrow PDT(q, KW, \delta)\} \mid Q \in GenerateQPT(E)}) = Eval(E, \delta)\).

Thus the base case 1 holds.

**Base case 2: \(E = TAGNAME\)**

This case is handled by lines 3-8 in Figure 4.27. It is easy to see that GenerateQPT(E) is a singleton set. Assume \(\{Q\} \in GenerateQPT(E)\}, then by the algorithm we know \(Q.V = \{Q.root, l\}\) where \(l\) is the leaf node, \(l.name = TAGNAME\), and \(Q.root.name='.'\). \(Q.E = \{(Q.root, l, '/', 'm')\}\).

If \(\delta(Q.root) = e\) where \(e \in D'.V\) where \(D'\) is a document in \(D\), NameTest\((S, tag) = V\) is the standard function in the specification of formal semantics that given a sequence of nodes \(S\), a tag name \(tag\), returns the sequence of nodes \(V \subseteq S\) and \(\forall node \in V\) \(node.name = \text{tag}\). Then according to the formal semantics, \(E=TAGNAME\) is evaluated using NameTest\((Value('.').Children, \text{TAGNAME})\) where \(Value('.'))\ is the node \('.\) is bounded to.

Let \(C = \{c \in e.Children | \text{Tag}(c) = \text{TAGNAME}\}, \) then \(Eval(E, \delta) = Concatenate(C)\) where Concatenate\((C)\) concatenates items in the set \(C\) in document order. On the other hand, since \(Q.E = \{(Q.root, l, '/', 'm')\}\), by definition of PDT, we
know that $\forall c \in C, c \in CE(l, D') \land c \in PDT(Q, KW, \delta).V$. Hence $Eval(E, \delta') = Concatenate(C)$. Finally, since $C-AnnMap[I]=true$ (line 7 in Figure 4.27), we know that $\forall x \in Eval(E, \delta'), x \text{has id}$. It is then easy to see that $I(Eval(E, \delta')) = Eval(E, \delta)$.

Thus the base case 2 holds.

**Induction Hypothesis:** For an expression E that is derived using grammar rules, suppose Theorem 4.6.1 holds for its sub-expressions.

We will now show that Theorem 4.6.1 holds for E itself. There are six cases, one for each different kind of derivation.

**Case 1: E= for Var in PathExpr return Expr**

The main evaluation rules of $Eval(E, \delta)$ are as follows.

The iteration expression $PathExpr$ is evaluated to produce the sequence $Item_1, ..., Item_n$. For each item $Item_i$ in this sequence, the body of the for expression $Expr$ is evaluated in the environment $\delta$ extended with $Var$ bound to $Item_i$. This produces values $Value_i, ..., Value_n$ which are concatenated to produce the result sequence.

The specific rules for $Eval(E, \delta)$ are:

\[
\delta \vdash PathExpr \Rightarrow Item_1, ..., Item_n \\
\delta + VAR \Rightarrow Item_1 \vdash Expr \Rightarrow Value_1 \\
\vdots \\
\delta + VAR \Rightarrow Item_n \vdash Expr \Rightarrow Value_n \\
\delta \vdash E \Rightarrow Value_1, ..., Value_n
\]

The evaluation rules for $Eval(E, \delta')$ are:

\[
\delta' \vdash PathExpr \Rightarrow Item'_1, ..., Item'_m \\
\delta' + VAR \Rightarrow Item'_1 \vdash Expr \Rightarrow Value'_1 \\
\vdots \\
\delta' + VAR \Rightarrow Item'_m \vdash Expr \Rightarrow Value'_m \\
\delta' \vdash E \Rightarrow Value'_1, ..., Value'_m
\]

W.o.l.g, assume $GenerateQPT(Expr) = \{Q_e\}$. There are two cases according to the value of $Q_e.root.name$. 

Case A: \(Q_e.root.name \neq \text{VAR}\). Intuitively, in this case, \(Expr\) does not reference \(\text{VAR}\). Therefore by \(\delta + \text{VAR} \Rightarrow \text{Item}_i \vdash Expr \Rightarrow \text{Value}_i\), we can infer that \(\delta \vdash Expr \Rightarrow \text{Value}_i\). This indicates that \(\forall i, j, \text{Value}_i = \text{Value}_j = \text{Value}\). Therefore \(\text{Eval}(E, \delta) = (\text{Value}, ..., \text{Value})\). Similarly, we can infer that \(\delta' \vdash Expr \Rightarrow \text{Value}'_i \land \forall i, \text{Value}'_i = \text{Value}'_j = \text{Value}' \land \text{Eval}(E, \delta') = (\text{Value}', ..., \text{Value}')\). Further, since \(r_e \neq \text{VAR}\), by lines 15-22 and line 25 in Figure 4.29, we know that \(Q_e \in \text{GenerateQPT}(E)\) and therefore \(Q_e.root \Rightarrow \text{PDT}(Q_e, KW, \delta) \in \delta'\). Therefore by I.H. on the sub-expression \(Expr\), we know that \(\text{Value} = I(\text{Value}')\). Further, by I.H. on \(\text{PathExpr}\), we know that \(\text{Eval}(\text{PathExpr}, \delta) = I(\text{Eval}(\text{PathExpr}, \delta'))\) and hence \(m = n\). Therefore we finally know \(I(\text{Eval}(E, \delta')) = \text{Eval}(E, \delta)\).

Case B: \(Q_e.root.name = \text{VAR}\). There are two different cases depending on whether \(Q_e.root\) has child edges in \(Q_e\).

Case B.1: \(Q_e.root\) has no child edges. In this case, the return expression \(Expr\) is just \(\text{VAR}\). By lines 15-22 in Figure 4.29, we know that \(Q_e \notin \text{GenerateQPT}(E)\). Therefore \(\text{GenerateQPT}(E) = \text{GenerateQPT}(\text{PathExpr})\). Hence if \(\delta'' = \{Q_q.root \Rightarrow \text{PDT}(Q_q, KW, \delta) \mid Q_q \in \text{GenerateQPT}(\text{PathExpr})\}\), then \(\delta' = \delta''\). So we can apply I.H. on \(\text{PathExpr}\) and know that \(I(\text{Item}'_i) = \text{Item}_i\).

Then since \(Q_e.V = \{Q_e.root\}, \text{Eval}(Expr, \delta + \text{VAR} \Rightarrow \text{Item}_i) = \text{Item}_i \) and \(\text{Eval}(Expr, \delta + \text{VAR} \Rightarrow \text{Item}'_i) = \text{Item}'_i\). Hence \(\text{Eval}(Expr, \delta + \text{VAR} \Rightarrow \text{Item}_i) = I(\text{Eval}(Expr, \delta + \text{VAR} \Rightarrow \text{Item}'_i))\) for all \(i\). Consequently, \(I(\text{Eval}(E, \delta')) = \text{Eval}(E, \delta)\).

Case B.2: \(Q_e.root\) has child edges. In this case, by lines 15-22 in Figure 4.29, the algorithm will create edges between the leaf nodes in \(\text{GenerateQPT}(\text{PathExpr})\) and the
child nodes of $Q_e.root$. W.o.l.g, assume $Q_e.root$ has a single child $x$. By Lemma 4.6.2, we know that $|\text{GenerateQPT}(\text{PathExpr})| = 1$. Let $\{q\} = \text{GenerateQPT}(\text{PathExpr})$.

Then by Lemma 4.6.2, we know that $|\text{Leaf}(q)| = 1$. Let $l \in q.V$ be the single leaf node in $q$. Then by lines 15-22 in Figure 4.29, it is easy to see that $|\text{GenerateQPT}(E)| = 1$. Assume $\{Q_q\} = \text{GenerateQPT}(E)$. Now depending on the edge annotations, there are further two different cases. Let $l' \in Q_q.V$ be $l$ in $\text{GenerateQPT}(E)$, and $e = (l, x, \text{axis}, \text{ann}) \in q.E$.

First, if $\text{ann} = 'o'$, then by definition, $CE(l) = CE(l')$. Therefore by lines 15-22 in Figure 4.29, it is easy to see that $PDT(q, KW, \delta).V = PDT(q, KW, \delta).V \setminus \{x\}$ and $PDT(q, KW, \delta).E = PDT(q, KW, \delta).E \setminus \{e\}$. Hence $\text{Eval}(\text{PathExpr}, \delta') = \text{Eval}(\text{PathExpr}, \{Q_q.root \Rightarrow PDT(q, KW, \delta)\})$. Further, by I.H. on PathExpr, we know that $I(\text{Eval}(\text{PathExpr}, \{Q_q.root \Rightarrow PDT(q, KW, \delta)\})) = \text{Eval}(\text{PathExpr}, \delta)$, and therefore we have $I(\text{Eval}(\text{PathExpr}, \delta')) = \text{Eval}(\text{PathExpr}, \delta)$. I.e., for all $i$, $I(\text{Item}_i') = \text{Item}_i$.

Then by I.H. on Expr, we know that $\forall \delta I(\text{Eval}(\text{Expr}, \{Q_q.root \Rightarrow PDT(q, KW, \delta)\}| q = \in \text{GenerateQPT}(\text{Expr}))) = \text{Eval}(\text{Expr}, \delta)$. If $\delta'' = \delta + \text{VAR} \Rightarrow \text{Item}_i$, then since $I(\text{Item}_i') = \text{Item}_i$, it is easy to see that $(Q_e.root \Rightarrow \text{Item}_i') = \{Q_q.root \Rightarrow PDT(q, KW, \delta'')| q \in \text{GenerateQPT}(\text{Expr})\}$ and hence $I(\text{Eval}, \{Q_q.root \Rightarrow PDT(q, KW, \delta'')| q \in \text{GenerateQPT}(\text{Expr})\}) = \text{Eval}(\text{Expr}, \delta'')$. Therefore $\text{Value}_i = I(\text{Value}_i')$ for all $i$. Consequently, $I(\text{Eval}(E, \delta')) = \text{Eval}(E, \delta)$.

Second, if $\text{ann} = 'm'$. If $\text{Eval}(\text{PathExpr}, \delta') = \text{Eval}(\text{PathExpr}, \{Q_q.root \Rightarrow PDT(q, KW, \delta)\}| q \in \text{GenerateQPT}(E))$ (i.e., $I(\text{Item}_i') = \text{Item}_i$ for all $i$), then we can use the same argument as above. Otherwise we know that $\text{Eval}(\text{PathExpr}, \delta') \subset \text{Eval}(\text{PathExpr}, \{Q_q.root \Rightarrow PDT(q, KW, \delta)\}| q \in \text{GenerateQPT}(E))$. Let
\[ X = \text{Eval}(\text{PathExpr}, \delta') \cap \text{Eval}(\text{PathExpr}, \{Q_q.\text{root} \Rightarrow \text{PDT}(q, \text{KW}, \delta) \mid q \in \text{GenerateQPT}(E)\}) \] and \[ Y = \text{Eval}(\text{PathExpr}, \{Q_q.\text{root} \Rightarrow \text{PDT}(q, \text{KW}, \delta) \mid q \in \text{GenerateQPT}(E)\}) \]

- \[ \text{Eval}(\text{PathExpr}, \delta') \]. For Item\(_i\)' in X, we can use the similar argument in the Case B.1 and show that \( I(\text{Value}_i') = \text{Value}_i \). Further, by the definition of PDT and definitions of CE, we know that \( \exists c, (l, c, \text{axis}', m') \in Q_q.E, \forall y \in Y, \forall n \in CE(n, D)\text{parent}(y, n) \).

Then, by Lemma 4.6.3, we can infer that \( \forall y \in Y, \text{Eval}(\text{Expr}, Q_e.\text{root} \Rightarrow y) = () \). Then we can use I.H. on \text{Expr} and infer that \( I(\text{Eval}(E, \delta')) = \text{Eval}(E, \delta) \).

**Case 2: \( E = \text{for VAR in PathExpr return } \langle \text{TAGNAME} \rangle \text{Expr} \langle / \text{TAGNAME} \rangle \)**

The evaluation rules are similar to Case 1 with the following additional rule for constructing the element.

\[
\text{Eval}(\langle \text{TAGNAME} \rangle \text{Expr} \langle / \text{TAGNAME} \rangle, \delta) = \text{element QName} \text{Eval}(\text{Expr}, \delta), \text{and}
\]

\[
\text{Eval}(\langle \text{TAGNAME} \rangle \text{Expr} \langle / \text{TAGNAME} \rangle, \delta') = \text{element QName} \{\text{Eval}(\text{Expr}, \delta')\},
\]

where element QName is the element construction function defined in the formal semantics.

Now we present the entire rules of \( \text{Eval}(E, \delta) \).

\[
\delta \vdash \text{PathExpr} \Rightarrow \text{Item}_1, ..., \text{Item}_n \\
\delta + \text{VAR} \Rightarrow \text{Item}_1 \vdash \text{Expr} \Rightarrow \text{Value}_1 \\
\quad ... \\
\delta + \text{VAR} \Rightarrow \text{Item}_n \vdash \text{Expr} \Rightarrow \text{Value}_n \\
\delta \vdash E \Rightarrow \text{element QName} \{\text{Value}_1\}, ..., \text{element QName} \{\text{Value}_n\}
\]

The evaluation rule for \( \text{Eval}(\text{expr}, \delta') \) is:

\[
\delta' \vdash \text{PathExpr} \Rightarrow \text{Item}_1', ..., \text{Item}_n' \\
\delta' + \text{VAR} \Rightarrow \text{Item}_1' \vdash \text{Expr} \Rightarrow \text{Value}_1' \\
\quad ... \\
\delta' + \text{VAR} \Rightarrow \text{Item}_n' \vdash \text{Expr} \Rightarrow \text{Value}_n' \\
\delta' \vdash E \Rightarrow \text{element QName} \{\text{Value}_1'\}, ..., \text{element QName} \{\text{Value}_n'\}
\]
By Lemma 4.6.2, we know $|\text{GenerateQPT}(\text{PathExpr})| = 1$. Let \( \{q\} = \text{GenerateQPT}(\text{PathExpr}) \), and w.o.l.g, assume $\text{GenerateQPT}(\text{Expr}) = \{Q_e\}$. Similar to Case 1, there are two cases according to the value of $Q_e.root$.

**Case A:** $Q_e.root \neq \text{VAR}$. The proof of this case is identical to Case A in proofs of Case 1 and therefore the proof is skipped here.

**Case B:** $Q_e.root = \text{VAR}$. There are two different cases depending on whether $Q_e.root$ has child edges.

**Case B.1:** $Q_e.root$ has no child edges. The proof of this case is identical to Case B.1 in Case 1 except that instead of returning $Value_i$, $\text{Eval}(E, \delta)$ now returns sequence of element $QName \{Value_i\}$, and $\text{Eval}(E, \delta')$ now returns sequence of element $QName \{Value'_i\}$. Therefore the proof is skipped here.

**Case B.2:** $Q_e.root$ has child edges. In this case, lines 15-22 in Figure 4.29, the algorithm will create edges between the leaf nodes in $\text{GenerateQPT}(\text{PathExpr})$ and the child nodes of $r_e$. W.o.l.g, assume $Q_e.root$ has a single child $x$. Assume $l$ is the single leaf node in $q$. Assume $l'$ is $l$ in GenerateQPT(E). Then by lines 16 and 39 in Figure 4.29, we know that $\forall e = (l, x, axis, ann), \text{ann} = \text{'}o\text{'}$.

Therefore we can first use the same argument as in Case B.2 in Case 1 when $\text{ann} = \text{'}o\text{'}$ and infer that for all $i$, $I(Item_i) = Item_i$. Then we can also use the same argument as in Case B.2 in Case 1 and use I.H. on $\text{Expr}$ to infer that $Value_i = I(Value'_i)$ for all $i$. Hence element $QName\{Value_i\}$

$= I(\text{element QName}\{Value'_i\})$ for all $i$. Consequently,

$I(\text{Eval}(E, \delta')) = \text{Eval}(E, \delta)$.

**Case 3:** $E = \text{for VAR in PathExpr return Expr1,Expr2}$

Let $Expr' = Expr1, Expr2$, the evaluation rules of $\text{Eval}(Expr1', \delta)$ is,
\[
\delta \vdash \text{Expr} \Rightarrow \text{Value}_1 \\
\delta \vdash \text{Expr} \Rightarrow \text{Value}_2 \\
\delta \vdash \text{Expr}' \Rightarrow \text{Value}_1, \text{Value}_2
\]

And the complete rules of \(\text{Eval}(E, \delta)\) are,

\[
\delta \vdash \text{PathExpr} \Rightarrow \text{Item}_1, ..., \text{Item}_n \\
\delta + \text{VAR} \Rightarrow \text{Item}_1 \vdash \text{Expr} \Rightarrow \text{Value}_{11} \\
\delta + \text{VAR} \Rightarrow \text{Item}_1 \vdash \text{Expr} \Rightarrow \text{Value}_{12} \\
\vdots \\
\delta + \text{VAR} \Rightarrow \text{Item}_n \vdash \text{Expr} \Rightarrow \text{Value}_{n1} \\
\delta + \text{VAR} \Rightarrow \text{Item}_n \vdash \text{Expr} \Rightarrow \text{Value}_{n2} \\
\delta \vdash E \Rightarrow \text{Value}_{11}, \text{Value}_{12}, ..., \text{Value}_{n1}, \text{Value}_{n2}
\]

The evaluation rule for \(\text{Eval}(\text{expr}, \delta')\) is:

\[
\delta' \vdash \text{PathExpr} \Rightarrow \text{Item}'_1, ..., \text{Item}'_n \\
\delta' + \text{VAR} \Rightarrow \text{Item}'_1 \vdash \text{Expr} \Rightarrow \text{Value}'_{11} \\
\delta' + \text{VAR} \Rightarrow \text{Item}'_1 \vdash \text{Expr} \Rightarrow \text{Value}'_{12} \\
\vdots \\
\delta' + \text{VAR} \Rightarrow \text{Item}'_n \vdash \text{Expr} \Rightarrow \text{Value}'_{n1} \\
\delta' + \text{VAR} \Rightarrow \text{Item}'_n \vdash \text{Expr} \Rightarrow \text{Value}'_{n2} \\
\delta' \vdash E \Rightarrow \text{Value}'_{11}, \text{Value}'_{12}, ..., \text{Value}'_{n1}, \text{Value}'_{n2}
\]

Therefore we need to show that (1) \(\forall i, I(\text{Value}'_{i1}) = \text{Value}_{i1}\), and (2) \(I(\text{Value}'_{i2}) = \text{Value}_{i2}\).

We now prove (1) holds and it is analogous to prove (2). By Lemma 4.6.2, we know \(|\text{GenerateQPT}(\text{PathExpr})| = 1\). Let \(\{q\} = \text{GenerateQPT}(\text{PathExpr})\), and \(\forall Q_e \in \text{GenerateQPT}(\text{Expr}, \delta)\), by line 49, we know that an optional edge will be created between leaf nodes of \(q\) and the child nodes of \(Q_e.\text{root}\). Now similar to Case 2, there are two difference cases.

**Case A:** \(Q_e.\text{root} \neq \text{VAR}\). The proof of this case is identical to Case A in Case 1 and therefore the proof is skipped here.

**Case B:** \(Q_e.\text{root} = \text{VAR}\). There are two different cases depending on whether \(r_e\) has child edges.

**Case B.1:** \(Q_e.\text{root} \text{ has no child edges.}\) The proof of this case is identical to Case B.1 in Case 1 and therefore the proof is skipped here.
Case B.2: $Q_e.root$ has child edges. The proof of this case is similar to Case B.2 in Case 2 in the sense that optional edges are created between leaf nodes in $q$ and the child nodes of $Q_e.root$. Therefore we can show that $I(Eval(PathExpr, \delta')) = Eval(PathExpr, \{r_q \Rightarrow PDT(q, KW, \delta)\})$, and therefore $I(Item'_{i1}) = Item_{i1}$, and we can also similarly infer that for all $i$, $I(Value'_{i2}) = Value_{i2}$.

Consequently, $I(Eval(E, \delta')) = Eval(E, \delta)$

Case 4: $E = let Var := PathExpr return Expr$

The evaluation rule of $Eval(E, \delta)$ is,

$$
\frac{\delta \vdash PathExpr \Rightarrow Item \quad \delta + VAR \Rightarrow Item \vdash Expr \Rightarrow Value}{\delta \vdash E \Rightarrow Value}
$$

The evaluation rule of $Eval(E, \delta')$ is,

$$
\frac{\delta' \vdash PathExpr \Rightarrow Item' \quad \delta' + VAR \Rightarrow Item' \vdash Expr \Rightarrow Value'}{\delta' \vdash E \Rightarrow Value'}
$$

By line 28 in Figure 4.29, the algorithm handles this case the same way as Case 1, and the proof of this case can be viewed as a special case of Case 1 in which $n = 1$. Therefore the complete proof is skipped.

And we can similarly prove the cases of $E = let Var := PathExpr return \langle TAGNAME \rangle Expr \langle /TAGNAME \rangle$ and $E = let Var := PathExpr return Expr1,Expr2$.

Case 5: $E = \text{'if'}\ Expr1 \ \text{'then'}\ Expr2 \ \text{'else'}\ Expr3$

The evaluation rules for $Eval(E, \delta)$ is,

$$
\frac{\delta \vdash fn : boolean(Expr1) \Rightarrow true \quad \delta \vdash Expr2 \Rightarrow Value_1}{\delta \vdash E \Rightarrow Value_1}
$$

and
\[\delta \vdash fn : boolean(Expr1) \Rightarrow false\]
\[\delta \vdash Expr3 \Rightarrow Value_2\]
\[\delta \vdash E \Rightarrow Value_2\]

The evaluation rules for \(Eval(E, \delta')\) is,

\[\delta' \vdash fn : boolean(Expr1) \Rightarrow true\]
\[\delta' \vdash Expr2 \Rightarrow Value'_1\]
\[\delta' \vdash E \Rightarrow Value'_1\]

and

\[\delta' \vdash fn : boolean(Expr1) \Rightarrow false\]
\[\delta' \vdash Expr3 \Rightarrow Value'_2\]
\[\delta' \vdash E \Rightarrow Value'_2\]

By line 28 in Figure 4.26, if \(Q1 = \text{GenerateQPT}(Expr1), Q2 = \text{GenerateQPT}(Expr2), \) and \(Q3 = \text{GenerateQPT}(Expr3), \) then \(\text{GenerateQPT}(E) = Q = Q1 \cup Q2 \cup Q3.\) If \(\delta_1 = \{r'_q \Rightarrow PDT(Q', KW, \delta) | Q' \in Q1\}, \delta_2 = \{r''_q \Rightarrow PDT(Q'', KW, \delta) | Q'' \in Q2\}, \delta_3 = \{r'''_q \Rightarrow PDT(Q''', KW, \delta) | Q''' \in Q3\}, \) by definition of PDT, we have \(\delta' = \delta_1 \cup \delta_2 \cup \delta_3.\)

By I.H., we know that \(I(Eval(Expr1, \delta_1)) = Eval(Expr1, \delta), I(Eval(Expr2, \delta_2)) = Eval(Expr2, \delta),\) and \(I(Eval(Expr3, \delta_3)) = Eval(Expr3, \delta).\)

Hence we have \(I(Eval(Expr1, \delta')) = Eval(Expr1, \delta), I(Eval(Expr2, \delta')) = Eval(Expr2, \delta),\) and \(I(Eval(Expr3, \delta')) = Eval(Expr3, \delta).\)

And then it is easy to see that \(I(Eval(E, \delta')) = Eval(E, \delta).\)

**Case 6: \(E = QName "(" PathExpr1,..., PathExprn ")"\)**

This case corresponds to function call and the evaluation rules for \(Eval(E, \delta)\) is,

\[\delta \vdash QName \text{ expands to QName}(VAR1,...,VARn)\{Expr\}\]
\[\delta \vdash \text{PathExpr1} \Rightarrow Value_1\]
\[...\]
\[\delta \vdash \text{PathExprn} \Rightarrow Value_n\]
\[\delta + \text{VAR1} \Rightarrow Value_1 + ... + \text{VARn} \Rightarrow Value_n \vdash \text{Expr} \Rightarrow Value\]
\[\delta \vdash E \Rightarrow Value\]

The rules for \(Eval(E, \delta')\) is,
δ′ ⊬ QName expands to QName(VAR1,...,VARn){Expr}

δ′ ⊬ PathExpr1 ⇒ Value′1
...
δ′ ⊬ PathExprn ⇒ Value′n
δ′ + VAR1 ⇒ Value′1 + ... + VARn ⇒ Value′n ⊬ Expr ⇒ Value′

There are two cases based on whether the function takes parameters.

Case 1: n=0. In this case, the function takes no parameters. By lines 41-53 in Figure 4.26, GenerateQPT(E) = GenerateQPT(Expr). Further, by I.H., we know that

I(Eval(Expr, {Q.root⇒PDT(Q, KW, δ) | Q ∈ GenerateQPT(Expr)}) = Eval(Expr, δ).

Hence I(Eval(Expr, {Q.root⇒PDT(Q, KW, δ) | Q ∈ GenerateQPT(E)}) = Eval(Expr, δ). I.e., I(Value') = Value. Therefore I(Eval(E, δ′)) = Eval(E, δ).

Case 1: n > 0. By the evaluation rules and lines 41 -53 in Figure 4.29, this case is similar to the case where E = let VAR1 := PathExpr1 ... let VARn := Exprn return Expr, which will be shown to be correct (by Case 2 and Theorem 4.6.6). Therefore the details are skipped here.

□

We now briefly show that Theorem 4.6.1(b) hold. First, for an expression E and an environment δ, for an element e ∈ Eval(E, δ), PDT ByteLength(e) = Σe′.Length where e′ ∈ e.Descendants ∧ e′ is a base element. Second, note in the algorithm, we set the annotation for the QPT node that is used in constructing the views in C-AnnMap to be true (Theorem 4.6.5) and therefore the required byte lengths of the base elements will be correctly collected and generated in the PDT (Theorem 4.6.5). Therefore if I(e) = e"wheree" ∈ Eval(E, δ) (Theorem 4.6.1(a), then we know e contains all required base elements and therefore Σe′.Length = len(e")

If Nodes(e, D) is the set of nodes in the subtree in D rooted at the node e, then we can show the following theorem.
Theorem 4.6.5 (C-AnnMap). Given a query expression $E$, an XML document $D$, an environment $\delta \in U E(D)$,

\[ \forall e' \in \{ \text{Nodes}(e, D) \mid e \in \text{Eval}(E, \{ Q.root \Rightarrow \text{PDT}(Q, KW, \delta) \mid Q \in \text{GeneraetQPT}(E) \}) \}, (\exists c \in \{ Q.V \mid Q \in \text{GeneraetQPT}(E) \}) e' \in CE(c) \Rightarrow C - \text{AnnMap}[c] = true \]

Proof. Sketch We prove Theorem 4.6.1(b) by structural inductions on $E$. Let $\delta' = \{ Q.root \Rightarrow \text{PDT}(Q, KW, \delta) \mid Q \in \text{GeneraetQPT}(E) \}$

Base case: $E = \text{fn:doc(Name)}$ or VAR or ‘.’

In this case, by the algorithm we know $\text{GenerateQPT}(E)$ produces a singleton set $\{Q\}$. And by line 6 in Figure 4.26, $C-\text{AnnMap}[Q.root]=true$.

On the other hand, according to formal semantics, we know that $\text{Eval}(E, \delta') = r$ where $\delta(Q.root) = r$. Therefore $r \in \text{candidatElems}(Q.root)$. Since we just show $C-\text{AnnMap}[Q.root]=true$, hence our theorem holds.

Induction hypothesis: Assume the theorem holds for sub-expressions of $E$. We need to show it holds for $E$ itself.

Here we show the case $E = \text{for VAR in PathExpr return Expr}$ to illustrate the main points. Other cases are similar and their proofs are ignored.

Case 1: $E = \text{for VAR in PathExpr return Expr}$

First, by the formal semantics, essentially $\text{Eval}(E, \delta') = \{ \text{Eval}(Expr, \delta' + \text{VAR} \Rightarrow \text{Item} \mid \text{Item} \in \text{Eval}(\text{PathExpr}, \delta')) \}$ where we overload the set operator ‘{ }’ to concatenate the items in the set. By I.H. on $Expr$, we know that if $e \in \text{Eval}(Expr, \delta')$ and $e \in CE(c')$ where $c'$ is a QPT node $\text{GenerateQPT}(Expr)$, then $C-\text{AnnMap}[c'] = true$.

Then, by lines 17-22 in Figure 4.29, we know that for all non-leaf nodes $x$ in $\text{GenerateQPT}(Expr)$, $C-\text{AnnMap}[x]$ remains the same. Now w.o.l.g., assume $\{ G \} =$
GenerateQPT(Expr). If C-AnnMap[G.root] = true and e ∈ candidateElem(G.root) and e ∈ Eval(Expr, δ'), then by the formal semantics of XQuery, we know Item ∈ Eval(Expr, δ') where Item ∈ Eval(PathExpr, δ') and Eval(Expr, δ' + VAR ⇒ Item) = e. Then by I.H. on PathExpr, know that Item ∈ CE(l) and C-AnnMap[l] = true, therefore our theorem holds.

We can similarly show that Theorem 4.6.1(c) also holds.

**Equivalence of QPT**

Given a query expression E that conforms to our grammar, if UEXPR is the universe of such expressions, and $E_{core}$ is the normalized expression of $E$ using the core grammar, then we show the following theorem.

**Theorem 4.6.6 (Equivalence of QPT).** $\forall E \in UEXPR, \text{GenerateQPT}(E) = \text{GenerateQPT}(E_{core})$

**Proof.** There are five cases to consider depending on types of the expression E.

**Base case: $E = (fn:doc(Name)|VAR|.)**

In this case $E = E_{core}$, and therefore the theorem is vacuously true.

**Induction Hypothesis:** Suppose the lemma holds for sub-expressions of E. We now prove the lemma also holds for E.

**Case 1: $E = (fn:doc(Name)|VAR|.) \ '/ PathTailExpr**

If $E' = (fn : doc(Name)|VAR|.)$, then $E_{core} = for$dot in $E'$ return PathTailExpr. (Note the variable $dot$ and '.' in our grammar indicate the same context item).

First by Lemma 4.6.2, we know that $|GenerateQPT(E')| = 1$. Also, by the argument of Case B.2 in Theorem 4.6.1, we know that $|GenerateQPT(E_{core})|$. Assume
\[\text{GenerateQPT}(E) = \{Q_e\}, \text{GenerateQPT}(E_{\text{core}}) = \{Q_c\}, \text{and GenerateQPT}(E') = \{q'\} .\]

By Lemma 4.6.2, we know that \(|\text{GenerateQPT}(\text{PathTailExpr})| = 1\). Assume \{q\} = \text{GenerateQPT}(\text{PathTailExpr}), and by line 7 in Figure 4.26, we know that \(q'.V = \{q'.root\} \text{ and } q'.E = \emptyset\). If \(E'' = \{(q'.root, l'/', ann)|l \in q.root.\text{Children}\}\) and \(E''' = \{(q.root, l'/', ann)|l \in q.root.\text{Children}\}\) and, then by lines 11-16 in Figure 4.26, we know that \(Q_e.V = (q.V - \{q.root\}) \cup \{q'.root\}, Q_e.E = (q.E - E''') \cup E'' \text{ and } Q_e.root = q'.root.\)

On the other hand, by line 7 in Figure 4.26, it is easy to see that \(q.root = ' \). Therefore in \(\text{GenerateQPT}(E_{\text{core}})\), by lines 15-22 in Figure 4.29, the algorithm will create edges from \(q'.root\) to the child nodes of \(q.root\). Therefore \(Q_c.V = (q.V - \{q.root\}) \cup \{q'.root\}, Q_c.E = (q.E - E''') \cup E'' \) and \(Q_c.root = q'.root\).

**Case 2: \(E = \text{PathExpr '} [' \text{PredExpr } ']\)**

In this case, \(E_{\text{core}} = \text{for } \$dot \text{ in PathExpr return if PredExpr then } \$dot \text{ else } ()\)

There are two cases according to whether \(\$dot\) is referenced in PredExpr. First, by line 28 in Figure 4.26, we know that \(\text{GenerateQPT (if PredExpr then } \$dot \text{ else } () ) = \text{GenerateQPT(PredExpr) } \cup \text{GenerateQPT('').}\)

First, if \(\forall q \in \text{GenerateQPT(PredExpr), } q.root \neq ' \). Then by lines 15-22 in Figure 4.29, we know that \(Q_C = \text{GenerateQPT(PathExpr) } \cup \text{GenerateQPT(PredExpr).}\) \(\text{GenerateQPT(').}\) is not in \(Q_C\) because it only has a single root node and therefore is ignored (lines 15-22). On the other hand, we know that if \(r_q \neq ' \), then line 24-25 in Figure 4.26 will not be executed, and therefore \(Q_E = \text{GenerateQPT(PathExpr) } \cup \text{GenerateQPT(PredExpr).}\) Consequently \(Q_E = Q_C.\)
Second, if \( \exists q \in \text{GenerateQPT}(\text{PredExpr}) \) \( q \cdot \text{root} = '!' \). Let \( X = \{ x \in \text{GenerateQPT}(\text{PredExpr}) \mid x \cdot \text{root} = '!' \} \), \( \forall x \in X \), if \( \{ q \} = \text{GenerateQPT}(\text{PathExpr}) \), 

\[
E'' = \{ (x \cdot \text{root}, l, '/') | l \in x \cdot \text{root}.\text{Children} \}
\]

and 

\[
E''' = \{ (q \cdot \text{root}, l, '/') | l \in x \cdot \text{root}.\text{Children} \}
\]

then by lines 15-22 in Figure 4.29, we know that \( Q_C = \{ Q' \} \cup \{ y | y \in \text{GenerateQPT}(\text{PredExpr}) - X \} \) where \( Q' = (V', E', r') \) and \( V' = \cup \{ x.V - \{ x \cdot \text{root} \} | x \in X \} \), \( E' = (q \cdot E - E'') \cup E'' \) and \( r' = q \cdot \text{root} \).

On the other hand, note that when invoking \( \text{GenerateQPT}(E) \), in Figure 4.29, lines 24-25 are essentially identical to lines 15-22, hence if \( \exists q \in \text{GenerateQPT}(\text{PredExpr}) \) \( q \cdot \text{root} = '!' \), 

\[
Q_e = \{ Q' \} \cup \{ y | y \in \text{GenerateQPT}(\text{PredExpr}) - X \}
\]

and consequently \( Q_E = Q_C \).

**Case 3: for VAR in PathExpr where Expr1 return Expr2**

In this case, \( E_{\text{core}} = \) for VAR in PathExpr return if Expr1 then Expr2 else ()

First, by line 28 in Figure 4.26, we know that

\[
\text{GenerateQPT(if Expr1 then Expr2 else())} = \text{GenerateQPT(Expr1)} \cup \text{GenerateQPT(Expr2)}
\]

Let \( G = \text{GenerateQPT(Expr1)} \cup \text{GenerateQPT(Expr2)} \). Then there are two cases according to whether VAR is referenced in \( G \).

First, if \( \forall g \in G \) \( g \cdot \text{root} \neq \text{VAR} \). Then by lines 15-22 in Figure 4.26, we know that 

\[
Q_C = \text{GenerateQPT(PathExpr)} \cup G
\]

On the other hand, we know that if \( g \cdot \text{root} \neq '!' \), lines 15-22 in Figure 4.26 will not be executed, and therefore 

\[
Q_E = \text{GenerateQPT(PathExpr)} \cup G
\]

Consequently \( Q_E = Q_C \).

Second, if \( \exists g \in G \) \( g \cdot \text{root} = '!' \). Let \( X = \{ x \in G \mid x \cdot \text{root} = '!' \} \), \( \forall x \in X \), if \( \{ q \} = \text{GenerateQPT(PathExpr)} \), 

\[
E'' = \{ (x \cdot \text{root}, l, '/') | l \in x \cdot \text{root}.\text{Children} \}
\]

and 

\[
E''' = \{ (q \cdot \text{root}, l, '/') | l \in x \cdot \text{root}.\text{Children} \}
\]

then by lines 15-22 in Figure 4.29, we know that 

\[
Q_C = \{ Q' \} \cup \{ y | y \in G - X \}
\]

where \( Q' = (V', E', r') \) and \( V' = \)
\{x.V - \{x.root\} | x \in X\} , E' = (q.E - E'') \cup E'' and r' = q.root.

On the other hand, note that when invoking GenerateQPT(E), by lines 4-9 in Figure 4.29, we also first produce a set of QPT \(G' = GenerateQPT(Expr1) \cup GenerateQPT(Expr2)\), therefore using the same argument on \(G'\) as above and using the same notations (with \(G'\) in place of \(G\)), we can infer that \(Q_E = \{Q'\} \cup \{y | y \in G' - X\}\) where \(Q = (V', E', r')\) and \(V' = \{x.V - \{r_x\} | x \in X\}\), \(E' = (q.E - E'') \cup E''\) and \(r' = q.root\). and consequently \(Q_E = Q_C\).

**Case 4: (forClause|letClause)+ return Expr**

This is proved separately in Theorem 4.6.7.

**Case 5: Other cases**

In all of other cases \(E = E_{core}\) and therefore the theorem is vacuously true. 

\[\square\]

**Theorem 4.6.7 (Equivalence of QPT of FLOWR).** For all \(E = (forClause|letClause)+ return Expr\), \(GenerateQPT(E) = GenerateQPT(E_{core})\)

**Proof.** For notational convenience, let \(Q_E = GenerateQPT(E)\) and \(Q_C = GenerateQPT(E_{core})\).

We prove the lemma by inductions on the number of for/let clauses, denoted by \(d\).

**Base case: \(d=1\)** In this case, \(E = for VAR in PathExpr return Expr\) or \(E = let VAR := PathExpr return Expr\).

In both cases \(E = E_{core}\) and therefore the lemma is vacuously true.

**Induction hypothesis:** Assume the lemma holds for \(d \leq n\). We now show the lemma holds for \(d = n + 1\).

There are two cases, one for each different types of root clauses.
Case 1: $E = \text{for VAR in PathExpr (forLetClause)+ return Expr}$ In this case, if $E' = (\text{forLetClause}+ \text{return Expr})$, then $E_{\text{core}} = \text{for VAR in PathExpr return } E'_{\text{core}}$

By I.H., we know that $\text{GenerateQPT}(E') = \text{GenerateQPT}(E'_{\text{core}}) = G$. Note that we use the same lines of code (lines 15-22 in Figure 4.29) to handle $G$ in $\text{GenerateQPT}(E)$ and $\text{GenerateQPT}(E')$, therefore it is easy to see that $Q_E = Q_C$.

Case 2: $E = \text{let VAR := PathExpr (forLetClause)+ return Expr}$ The proof of this case is identical to Case 1 due to line 28 in Figure 4.29.

4.7 Correctness of GeneratePDT

In this section, we first show the full version of the algorithm GeneratePDT, and then show its correctness.

4.7.1 Generalized version of GeneratePDT

As described in Section 4.4.2, the full version of the algorithm makes two extensions to the original one presented in Section 4.4.2. First, the original algorithm always creates and propagates IDs that satisfy the descendant constraints in the pdt cache. This can be optimized by immediately creating the IDs in the result PDT if they also satisfy the ancestor restrictions. For this purpose, we add a boolean flag InPdt to the CT node, set InPdt to be true when the ID is created in the result PDT, and create the descendant ID in the PDT when one of its parents is in the PDT (InPdt = true).

Further, as mentioned earlier, when the QPT have repeating tag names, a single Dewey ID can match multiple QPT nodes. For example, if the QPT path is “//a/a” and
Figure 4.30: Algorithm for generating PDTs

the corresponding full data path is “/a/a/a”, then the second a in the full path matches both nodes in the QPT path. To handle this case, we extend the structure of CT node to have a set of QPTNodes, each of which is associated with their own InPdt, PL and DM. This is because in general different QPT nodes capture different ancestor-descendant constraints, hence must be treated separately.

4.7.2 Generalized PDT definitions

In this section, we generalize the definitions of PDT described earlier in Section 4.4 so that it also handles the cases where the root of the QPT is mapped to arbitrary nodes in an XML database.

We first introduce some notation. We \( \text{Nodes}(D) \) to denote the set of nodes in an XML database \( D \), \( \text{FreeVars}(E) \) to denote the set of free variables in a query expression \( E \), \( \text{Env}(D, \text{FreeVars}(E)) \) to denote the evaluation environment which binds variables in \( \text{FreeVars}(E) \) to nodes in \( \text{Nodes}(D) \), \( UE(D, \text{FreeVars}(E)) \) to denote the universe
AddCTNode(CTNode parent, DeweyID id, int depth)

if depth ≤ id.Depth then
    curId ← Prefix(id, depth); qNodes ← QPTNodes(curId)
    if qNodes = ∅ then AddCTNode(parent, id, depth+1)
else
    newNode ← parent.findChild(curId)
    if newNode = null then
        newNode ← parent.addChild(curId, qNodes)
        Initialize newNode.CTQPTNodeSet using qNodes
        Update the data value and tf values if required
    end if
    AddCTNode(newNode, id, depth+1)
end if
for all q in qNodes do
    if ∀i, q.DescendantMap[i]=1 then
        set DescendantMap[q] to 1 for nodes in q.ParentList
    end if
end for

Figure 4.31: Algorithm for adding new CT nodes

of such environments. In Env(D, FreeVars(E)), we use var ⇒ n to denote that var in FreeVars(e) is bounded to the node n. Similarly, for a QPT Q, we say Env(D, Q) is an environment that binds Q.root whose name is a free variable to a node in the database D, and UE(D, Q) is the universe of such environments. Note by definition of QPT, only the root of a QPT can be a free variable.

Further, if QSet = GenerateQPT(E) is a set of QPTs corresponding the expression E, then ∀Q ∈ QSet, ∀δ ∈ UE(D, FreeVars(E)), ∀δ′ ∈ UE(D, Q), (∃x ∈ FreeVars(E), x = Q.root.name ⇒ δ′(Q.root) = δ(x)). In this case, for notational convenience, we use UE(D, FreeVars(E)) and UE(D, Q) interchangeably.

Finally, given a node d ∈ Nodes(D), we use T(d) to denote the XML sub-tree rooted at d, and T(d) is a 4-tuple (V, E, Tag, Value) where V is the set of nodes in T(d), E is the set of edges in T(d), Tag are the mappings from nodes in V to their tag names, and Value are the mappings from nodes in V to their data values.

Now we generalized the notions of PDTs defined in the main body to handle arbi-
CreatePDTNodes (CTNode n, QPT qpt, PDT pdt, PDT parentPdtCache)
{
Create PDT nodes using CT nodes in left most path}
1: for all q in n.CTQPTNodes where q.InPdt = false do
2: if ∃ i, q.DescendantMap[i] = 1 then
3: if q.ParentList = ∅ ∨ ∃ p ∈ q.ParentList, p.InPdt = true then
4: q.InPdt = true; Write n.id to pdt if n.id /∈ pdt
5: else
6: pdtCacheNode = parentPdtCache.find(n.Id)
7: if parentCacheNode = null
8: pdtCacheNode = parentPdtCache.add(n.Id)
9: for all q in n.CTQPTNodes where ∃ i, q.DescendantMap[i] = 1 do
10: pdtCacheNode.ParentList.add(q.ParentList)
11: end for
12: end if
13: end if
14: end for
15: if n.HasChild() = true then
16: {Recursively handle the left most child(LMC)}
17: CreatePDTNodes(LMC, qpt, pdt, n.PdtCache)
18: else
19: {Handle pdtCache and then remove the node itself}
20: for x in n.pdtCache do
21: if x.ParentList = ∅ ∨ ∃ p ∈ x.ParentList, p.InPdt = true then Write x.id to pdt if x.id /∈ pdt
22: else
23: {Update parent list and then propagate x to parentPdtCache}
24: for all q in n.CTQPTNodes where q in ParentList(x) do
25: x.ParentList.remove(q)
26: if ∃ i, q.DescendantMap[i] = 0 ∧ ParentList(x) = ∅ then n.pdtCache.remove(x)
27: else
28: x.ParentList.replace(q, q.ParentList)
29: end if
30: end for
31: if x ∈ pdtCache then PropagatePDT(x, parentPdtCache)
32: end if
33: end for
34: n.RemoveFromCT()
35: end if

Figure 4.32: Algorithm for generating PDTs

Tertiary nodes. Given database D, a QPT Q, an environment δ ∈ UE(D, FreeVars(Q)), δ(r_q) = d where d ∈ Nodes(D), then we have the following generalized definitions of candidate elements, PDT elements, and PDT using the same notations as in Section 4.4.1, except that the document is replaced by the subtree rooted at d.

Definition 4.7.1 (candidate elements). For any node n in the QPT Q,

- n is a leaf node in Q: CE(n, d) = \{v ∈ T(d) | tag name of v is n.tag ∧ the value of v satisfies all predicates in n.preds \}. 
• n is a non-leaf node in Q: 

\[ CE(n, d) = \{ v \in T(d) \mid \text{tag name of } v \text{ is } n.tag \land \text{for every edge } e \text{ in } Q, \text{ if } e\.parent = n \text{ and } e\.ann = 'm' (mandatory), \text{ then } \exists ec \in CE(e\.child, d) \text{ such that (a) } e.axis = '/' \Rightarrow v \text{ is the parent of } ec, \text{ and (b) } e.axis = '//' \Rightarrow v \text{ is an ancestor of } ec \} \]

Definition 4.7.2 (PDT elements). For any node n in the QPT Q.

• n is the root node: 

\[ PE(n, d) = CE(n, d) \]

• n is the non-root node: 

\[ PE(n, d) = \{ v \in d \mid v \text{ is in } CE(n, d) \land \text{for every edge } e \text{ in } Q, \text{ if } e\.child = n, \text{ then } \exists vp \in PE(e\.parent, d) \text{ such that (a) } e.axis = '/' \Rightarrow vp \text{ is the parent of } v, \text{ and (b) } e.axis = '//' \Rightarrow vp \text{ is an ancestor of } v \} \]

Definition 4.7.3 (PDT). The PDT is a tree \((N, E)\) where \(N\) is the set of nodes and \(E\) is the set of edges which are defined as follows.

• \(N = \bigcup_{q \in Q} PE(q, d)\), and nodes in \(N\) are associated with required values, \(tf\) values and byte lengths.

• \(E = \{(p, c) \mid p, c \text{ are in } N \land p \text{ is an ancestor of } c \land \exists q \in N \text{ s.t. } p \text{ is an ancestor of } q \text{ and } q \text{ is an ancestor of } c\}\)

4.7.3 Proofs of correctness

Now we show that given a QPT, the algorithm GeneratePDT generates the correct PDT that conforms to our PDT specifications. Theorem 4.7.4 formally describes the correctness of GeneratePDT.

We first introduce some notations. Given a QPT \(Q\), a database \(D\), a node \(d \in Nodes(D)\), an environment \(\delta \in UE(D, Q)\), we use \((d\.PathIndex)\) and \(d\.InvIndex\) to denote the path indices and inverted indices associated with \(T(d)\), respectively. Given a
QPT Node qn, d.PathIndex.LookUp(qn) returns an ordered list of node ids that correspond to the root to leaf path leading to qn in Q. Each node in the list also satisfies the predicates associated with qn. Given a keyword k, d.InvIndex returns a list of node ids that contains the keyword, along with the tf value.

The following Theorem 4.7.4 shows the correctness of the algorithm GeneratePDT.

**Theorem 4.7.4.** Given a set of keyword KW, a QPT Q, an XML database D, an environment δ ∈ UE(D, Q), GeneratePDT(Q, δ(Q.root).PathIndex, δ(Q.root).InvIndex, KW) = PDT(Q, KW, δ).

**Notations**

We now introduce more notations before proving the Theorem 4.7.4.

**Prefixes**

Given a set of keyword KW, a QPT Q, an XML database D, an environment δ ∈ UE(D, Q), δ(Q.root) = d, (pathLists, invLists) = PrepareList(Q, d.PathIndex, d.InvIndex, KW). At a given time # t, we say $H(t, pathLists) = \{id ∈ l | l ∈ pathLists ∧ id is retrieved by the time # t\}$ is the set of ids that has been retrieved from pathLists by the time # t (including t). In our algorithm, t corresponds to the number of loops (lines 8-19 in Figure 4.30).

Next, given a QPT node q in Q, for all $q'$ in ancestor nodes of q, and an Dewey id did in pathLists corresponding to q, we use Prefix(q, did, q') to denote the set of prefixes of did that corresponds to $q'$. Note Prefix(q, did, q') is a set because when the path containing q and $q'$ have the axis '//', there can be multiple matchings of $q'$ in prefixes of did.

Further, $\forall l ∈ L$, we say Prefix(l) = \{x ∈ Prefix(l.QPTNode, lid, q) | lid ∈ l, q ∈
anc(l.QPTNode) \} is the set of prefixes of ids in \( l \) w.r.t \( l.QPTNode \), and \( \text{Prefix}(L) = \{ x \in \text{Prefix}(l) \mid l \in L \} \) is the set of prefixes of ids in \( H(t, \text{pathLists}) \).

**Pruned Document Tree Based on ID Lists**

Note since \( \text{pathLists} \) is retrieved by \( d.\text{PathIndex} \), ids in \( \text{pathLists} \) can be used to re-create a pruned document tree of \( T(d) \). We call lists of ids that can be used to create a valid XML document tree the document-compatible id lists. Essentially in such lists, if two Dewey ids are identical, then their corresponding path must have the same tag names at each step. If \( UL \) is the universe of ordered document-compatible id lists, we use \( \text{Comp}(H(t, \text{pathLists})) \in 2^{UL} \) to denote the universe of completions of id lists in \( H(t, \text{pathLists}) \).

For a set of id lasts \( L \in UL \), we use \( T(L) = (V, E, \text{Tag}, \text{Value}, \text{Cont}) \) to denote the document tree that contains all and only ids in \( L \). More formally, if \( \text{rootId}(L) \) is the first id component that all ids in \( L \) shares and \( \text{root}(T) \) is the root node of tree \( T \), then \( T \) first satisfies the following properties concerning ids.

- \( \text{id}(\text{root}(T(L))) = \text{rootId}(L) \) (The id of the root node is the first component of the Dewey ID in the lists.)
- \( \forall m, n \in T(L), \text{parent}(m, n) \iff (m.\text{id}, n.\text{id} \in \text{Prefix}(L) \land \text{parent}(\text{id}(m), \text{id}(n))) \) (the parent child relations of nodes in \( T \) is decided by the Dewey ids are in the lists).
- \( \forall \text{pid} \in \text{Prefix}(L), \exists n \in T, \text{id}(n) \in T.\text{Cont} \land \text{id}(n) = \text{pid} \land \exists m \in T, m \neq n \land \text{id}(m) = \text{pid} \). (there is a unique nodes corresponding to each component of the Dewey id).

Intuitively, \( T \) and \( L \) has one-to-one mappings on ids. For an id \( \text{did} \in \text{Prefix}(L) \), if \( \text{Node}(T, \text{did}) \) is the node in \( T \) s.t. \( \text{Node}(T, \text{did}).\text{id} = \text{did} \), then \( T \) further satisfies the
following properties.

- $\forall l \in L, \forall id \in l, \forall aq \in \text{anc}(l.QPTNode), \forall pid \in \text{Prefix}(l.QPTNode, id, aq), \Tag(\Node(T, pid)) = aq.name$.

- $\forall pid \in \text{prefix}(L), \text{Value}(pid) \neq null \Rightarrow \text{Value}(\Node(pid)) = \text{Value}(pid) \land \forall pid \in \text{prefix}(L), id(n) = pid \Rightarrow \text{Value}(pid) = null \Rightarrow \text{Value}(\Node(T, pid)) = null$.

Hence $T(H(t, pathLists))$ denotes the hypothetical of subtree of $T(pathLists)$ that contains ids in $H(t, pathLists)$.

Further, we use $CT(t)$ and $GenPDT(t)$ to denote the candidate tree and the PDT the algorithm generates after the loop # $t$. We also use $CT(t-) = CT(t - 1)$ with new IDs added in the beginning of the loop # $t$ by lines 10-14. and use $CT(t-)$ to denote the candidate tree after we process nodes in the $CT(t-)$ (lines 2-17).

We define $C(0-) = C(0 --) = C(0)$.

For notational convenience, given a Dewey id $did$, if there exists a node $n \in CT(t).V$ (or $GenPDT(t).V$, or $PDT$), $n.id=did$, then we say $did \in CT(t)$ (or $GenPDT(t)$, or $PDT$).

And given a id $pid$, a QPT $Q$, a set of keywords $KW$, $L \in UL$, we say the predicate Qualified($pid, Q, KW, L$) = true $\iff$ pid $\in$ PDT($Q$, KW, $\{Q.root \Rightarrow T(L).root\}$).

Proofs

At a high level, the algorithm GeneratePDT consists of three steps. First, it invokes PrepareList to construct lists of Dewey ids, ordered by id, that correspond to nodes without mandatory children nodes in the QPT. Then, it initializes the candidate tree using the minimum ID from each id list. Next, it enters a loop which keeps creating PDT nodes using qualified (defined later) CT nodes and creating new CT nodes using
available IDs. The algorithm terminates after processing all IDs, and removing all nodes in the CT.

The core part of the algorithm GeneratePDT is the while-loop (lines 8-19 in Figure 4.30) which keeps creating PDT nodes using nodes in the candidate tree, and creating new nodes in the candidate tree using the next available id in the id lists. We first prove a theorem that characterizes the invariant of this loop.

**Lemma 4.7.5.** Given a set of keyword $KW$, a QPT $Q$, an XML database $D$, an environment $\delta \in U_{E}(D, Q)$, if $\delta(Q.root) = d$ and $(\text{pathLists}, \text{invLists}) = \text{PrepareList}(Q, d.\text{PathIndex}, d.\text{InvIndex}, KW)$, then after the loop $\# t$,

(a) $\forall \text{pid} \in \text{Prefix}(H(t, \text{pathLists}))$, $\text{Qualified}($\text{pid}, $Q, KW, H(t, \text{pathLists})) = \text{true} \Rightarrow (\text{pid} \in \text{GenPDT}(t) \lor \text{pid} \in \text{CT}(t)) \lor \exists n \in \text{CT}(t).V, \text{pid} \in n.\text{PDTCache})$ (qualified nodes are in the candidate tree or the result PDT), and

(b) $\forall \text{id} \in \text{GenPDT}(t)$, $\text{id} \in \text{Prefix}(H(t, \text{pathLists})) \land \text{Qualified}($\text{id}, $Q, KW, H(t, \text{pathLists})) = \text{true}$. (all nodes in the PDT are qualified)

Lemma 4.7.5 indicates that after the loop $\# t$, if a Dewey id is a result PDT node based on the ids we have processed by $t$, then the id must be kept in GenPDT($t$), CT($t$), or pdt caches of CT($t$). Further, if for any possible completion of the id lists we have processed, this Dewey id is not qualified, then it is not in CT($t$), GenPDT($t$), or pdt caches of CT($t$)

**Supporting lemmas for Lemma 4.7.5**

We now present a set of lemmas that will be used in the proof of Lemma 4.7.5. Proofs will be presented after we show the main theorem.

First, by the definition of PDT, it is easy to show the following lemma.
Lemma 4.7.6 (Monotonicity). Given a set of keyword KW, a QPT Q, an XML database D, an environment \( \delta \in UE(D, Q) \), if \( \delta(Q.root) = d \) and \((pathLists, invLists) = PrepareList(Q, d.PathIndex, d.InvIndex, KW)\), then for any loop \( \# t \),

(a) \( \forall pid \in Prefix(H(t, pathLists)), Qualified(pid, Q, KW, H(t, pathLists)) = true \Rightarrow \forall L \in Comp(H(t, pathLists)), Qualified(pid, Q, KW, L) = true. \)

(b) \( \forall cn \in CT(t), \forall cnq \in cn.CTQPTNodeSet, id(cn) \in CE(cnq. T(H(t, pathLists)).root) \Rightarrow \forall t' \geq t, (cn \in CT(t') \Rightarrow id(cn) \in CE(cnq, T(H(t', pathLists)).root)). \)

(c) \( \forall cn \in CT(t), \forall cnq \in cn.CTQPTNodeSet, id(cn) \in PE(cnq. T(H(t, pathLists)).root) \Rightarrow \forall t' \geq t, cn \in CT(t') \Rightarrow \in PE(cnq. T(H(t', pathLists)).root). \)

The key idea is that the membership of a PDT node is determined by existence of its ancestor nodes and its mandatory children nodes in the PDT. Hence given a QPT and a set of ids SI, if an id is included in the PDT as per the definition, then this id is also included in the PDT using any superset of SI because all of its ancestor and children nodes must also be in the superset.

Given a QPT q and a node \( qn \in q \), we say MC(qn) = \{qnc \mid (qn, qnc, axis, ’m’) \in q.E \wedge axis = ’/’ or ’//’\} is the mandatory children nodes of \( qn \) in q. We also represent an edge e in the QPT using the 4-tuple (parent, child, axis, ann) where parent and child are the parent and child node of e, respectively, axis is ’/’ or ’//’, and ann is ’o’ or ’m’ as described earlier.

Given a CT node cn, a QPT node \( qn \in MC(CT.CTQPTNode) \), the following Lemma 4.7.7 indicates that the value of cn.DescendantMap[qcd] corresponds to whether cn has a child/descendant node that is also a candidate element. Since we add new ids by calling AddNewCTNodes(), we use \( List_t \) to denote the lists of IDs that have been retrieved after calling t times of AddNewCTNodes, and \( CT_t \) denote the candidate tree after calling t
times of AddNewCTNodes.

**Lemma 4.7.7** (DescendantMap). Given a set of keyword KW, a QPT Q, an XML database D, an environment $\delta \in UE(D, Q)$, if $\delta(Q.root) = d$ and $(\text{PathLists}, \text{InvLists}) = \text{PrepareList}(Q, d.\text{PathIndex}, d.\text{InvIndex}, KW)$, then after adding $##t$ IDs, $\forall cn \in CT$, $\forall cnq \in cn.\text{CTQPTNodeSet}$ $\forall qcd \in MC(\text{cnq.}\text{QPTNode})$, $\text{cnq. DescendantMap}[qcd] = 1 \iff$

$((\text{cnq.}\text{QPTNode}, qcd, '/', 'm') \in Q.E \Rightarrow \exists l \in \text{List}_t, \exists \text{id} \in l, \exists \text{cid} \in \text{Prefix(l.}\text{QPTNode, lid, qcd}), \exists \text{ce} \in \text{CE(qcd, T(H(t, PathLists)).root)}, \text{ce.id = cid} \land \text{id}(cn) \in \text{Prefix(l.}\text{QPTNode, lid, cnq.}\text{QPTNode}) \land \text{parent(id}(cn), \text{cid})) \land$

$((\text{cnq.}\text{QPTNode}, qcd, '//', 'm') \in Q.E \Rightarrow \exists l \in \text{List}_t, \exists \text{id} \in l, \exists \text{cid} \in \text{Prefix(l.}\text{QPTNode, lid, qcd}), \exists \text{ce} \in \text{CE(qcd, T(H(t, PathLists)).root)}, \text{ce.id = cid} \land \text{id}(cn) \in \text{Prefix(l.}\text{QPTNode, lid, cnq.}\text{QPTNode}) \land \text{anc(id}(cn), \text{cid}))$)

Since at each loop (lines 8-19), we start by adding new IDs corresponding to the current left most path, Then it is easy to infer the following lemma from Lemma 4.7.7.

**Lemma 4.7.8** (DescendantMap). Given a set of keyword KW, a QPT Q, an XML database D, an environment $\delta \in UE(D, Q)$, if $\delta(Q.root) = d$ and $(\text{PathLists}, \text{InvLists}) = \text{PrepareList}(Q, d.\text{PathIndex}, d.\text{InvIndex}, KW)$, then for every loop $##t$, $\forall cn \in CT(t - -)$, $\forall cnq \in cn.\text{CTQPTNodeSet}$ $\forall qcd \in MC(\text{cnq.}\text{QPTNode})$, $\text{cnq. DescendantMap}[qcd] = 1 \iff$

$((\text{cnq.}\text{QPTNode}, qcd, '/', 'm') \in Q.E \Rightarrow \exists l \in H(t, \text{PathLists}), \exists \text{id} \in l, \exists \text{cid} \in \text{Prefix(l.}\text{QPTNode, lid, qcd}), \exists \text{ce} \in \text{CE(qcd, T(H(t, PathLists)).root)}, \text{ce.id = cid} \land \text{id}(cn) \in \text{Prefix(l.}\text{QPTNode, lid, cnq.}\text{QPTNode}) \land \text{parent(id}(cn), \text{cid})) \land$

$((\text{cnq.}\text{QPTNode}, qcd, '//', 'm') \in Q.E \Rightarrow \exists l \in H(t, \text{PathLists}), \exists \text{id} \in l, \exists \text{cid} \in \text{Prefix(l.}\text{QPTNode, lid, qcd}), \exists \text{ce} \in \text{CE(qcd, T(H(t, PathLists)).root)}, \text{ce.id = cid} \land \text{id}(cn) \in \text{Prefix(l.}\text{QPTNode, lid, cnq.}\text{QPTNode}) \land \text{anc(id}(cn), \text{cid}))$)
Now, Lemma 4.7.9 indicates that if the flag InPdt of a CT node is true, the id of this node is qualified.

**Lemma 4.7.9 (InPdt).** Given a set of keyword KW, a QPT Q, an XML database D, an environment $\delta \in UE(D, Q)$, if $\delta(Q.root) = d$ and $(pathLists, invLists) = PrepareList(Q, d.PathIndex, d.InvIndex, KW)$, then at the loop # $t$,

(a) $\forall n \in CT(t-), \forall nq \in n.CTQPTNodeSet, nq.InPdt = true \Rightarrow cn \in PE(nq.QPTNode, T(H(t, pathLists)).root)$.

(b) $\forall n \in CT(t-).LeftMostPath, \forall nq \in n.CTQPTNodeSet, t > 0 \land cn \in PE(nq.QPTNode, T(H(t, pathLists)).root)) \Rightarrow nq.InPdt = true \land (\forall t' \geq t, n \in CT(t') \Rightarrow nq.InPdt = true \land n \in CT(t' - -) \Rightarrow (nq \in n.CTQPTNodeSet \land nq.InPdt = true))$.

The following Lemma 4.7.10 characterizes the properties of pdt cache. Note that for ease of exposition, we additionally associate each node in the pdt cache with a set of QPT node, denoted as PDTQPTNodes, as CTQPTNodeSet in CT nodes. Formally, we change line 10 in Figure 4.32 to the following.

`pdtCacheNode.PDTQPTNodes.add(q.QPTNode, q.ParentList)`

Then for a node $n$ in the pdt cache, it is easy to see that $ParentList(n) = \{x \in q.ParentList| q \in n.PDTQPTNodes\}$, and we use $n.ParentList$ and $ParentList(n)$ interchangeably.

**Lemma 4.7.10 (PDTCache).** Given a set of keyword KW, a QPT Q, an XML database D, an environment $\delta \in UE(D, Q)$, if $\delta(Q.root) = d$ and $(pathLists, invLists) = PrepareList(Q, d.PathIndex, d.InvIndex, KW)$, then at the loop # $t$,

(a) $\forall cn \in CT(t), \forall cnp \in cn.pdtCache, \forall q \in cnp.PDTQPTNodes, \exists ce \in CE(q, T(H(t, pathLists)).root), ce.id = cnp.id$ (nodes in the pdt caches satisfy the descendant restrictions).
(b) $\forall cn \in CT(t), \forall cnp \in cn.pdtCache, (\text{ParentList}(cnp) \neq \emptyset \land \forall cnpp \in \text{ParentList}(cn), cnpp.\text{InPdt} = \text{false}) \Rightarrow \text{Qualified}(cnp.id, H(t, pathLists)) = \text{false}$ (if parents are not qualified, then the node itself is not qualified).

(c) $\forall cn \in CT(t), \forall cnp \in cn.pdtCache, (\text{ParentList}(cnp) = \emptyset \lor \exists cnpp \in \text{ParentList}(cnp), cnpp.\text{InPdt} = \text{true}) \Rightarrow \text{Qualified}(cnpp.id, H(t, pathLists)) = \text{true}$ (if the node does not have parents or at least one parent is qualified, then the node is qualified).

For notational convenience, given a Dewey id did and a candidate tree CT, if there exists a node n $\in$ CT and did $\in$ n.pdtCache, then we say did $\in$ pdtCache(CT).

**Lemma 4.7.11** (Completeness of CT). Given a set of keyword KW, a QPT Q, an XML database D, an environment $\delta \in UE(D, Q)$, if $\delta(Q.\text{root}) = d$ and (pathLists, invLists) = $\text{PrepareList}(Q, d.\text{PathIndex}, d.\text{InvIndex}, KW)$, then at the loop # $t$, $\forall \text{pid} \in \text{Prefix}(H(t, pathLists))$, $\text{Qualified}(\text{pid}, Q, KW, H(t, pathLists)) = \text{false}$ $\land$ $\exists L \in \text{Comp}(H(t, pathLists))$, $\text{Qualified}(\text{pid}, Q, KW, L) = \text{true}$ $\Rightarrow$ $\text{pid} \in CT(t) \lor \text{pid} \in \text{pdtCache}(CT(t))$.

Lemma 4.7.11 indicates that if a Dewey id could potentially be a qualified id, then it will be included in the candidate tree.

Finally, when the algorithm initializes the candidate tree (lines 5-6 in Figure 4.30), it simply creates nodes in the candidate tree using the minimum ids from each list, and does not remove nodes or create node in the pdt cache. Therefore if MinimumID(l) is the minimum Dewey id in the list l, then it is straightforward to infer the following lemma.

**Lemma 4.7.12** (Initialization of CT). Given a set of keyword KW, a QPT Q, an XML database D, an environment $\delta \in UE(D, Q)$, if $\delta(Q.\text{root}) = d$ and (pathLists, invLists) = $\text{PrepareList}(Q, d.\text{PathIndex}, d.\text{InvIndex}, KW)$, then after initializing the candidate tree CT,
(a) \( \forall id \in CT, \exists l \in pathLists, \exists q \in \text{anc}(l.QPTNode), \exists pid \in \text{Prefix}(l.QPTNode, \text{MinimumID}(l), q), id = pid \)

(b) \( \forall l \in pathLists, \forall q \in \text{anc}(l.QPTNode), \forall pid \in \text{Prefix}(l.QPTNode, \text{MinimumID}(l), q), pid \in CT. \)

Proofs of Lemma 4.7.5

We separate Lemma 4.7.5 into two parts and prove each of them separately.

Lemma 4.7.13. Given a set of keyword KW, a QPT Q, an XML database D, an environment \( \delta \in U E(D, Q) \), if \( \delta(Q.root) = d \) and \( (\text{pathLists}, \text{invLists}) = \text{PrepareList}(Q, d.PathIndex, d.InvIndex, KW) \), then after the loop \# t, \( \forall pid \in \text{Prefix}(H(t, pathLists)), \text{Qualified}(pid, Q, KW, H(t, pathLists)) = \text{true} \) \( \Rightarrow (pid \in \text{GenPDT}(t) \lor pid \in CT(t)) \lor \exists n \in CT(t).V, pid \in n.PDTCache \) (qualified nodes are in the candidate tree or the PDT).

Lemma 4.7.14. Given a set of keyword KW, a QPT Q, an XML database D, an environment \( \delta \in U E(D, Q) \), if \( \delta(Q.root) = d \) and \( (\text{pathLists}, \text{invLists}) = \text{PrepareList}(Q, d.PathIndex, d.InvIndex, KW) \), then after the loop \# t, \( \forall id \in \text{GenPDT}(t), id \in \text{Prefix}(H(t, pathLists)) \land \text{Qualified}(id, Q, KW, H(t, pathLists)) = \text{true} \). (all nodes in the PDT are qualified).

We first prove Lemma 4.7.13.

Proof. We prove Lemma 4.7.13 by induction on the loop \# t.

Base case: \( t = 0 \) In this case, the algorithm just initializes the candidate tree using the minimum ids from each list in pathLists, and it is easy to see that GenPDT(t) = null, and \( \forall n \in CT(t), n.PDTCache = null \). On the other hand, by Lemma 4.7.12, we know
that $\forall l \in \text{pathLists}, \forall q \in \text{anc}(l,\text{QPTNode}), \forall \text{pid} \in \text{Prefix}(l,\text{QPTNode}, \text{MinimumID}(l), q), \text{pid} \in \text{CT}$. This implies that $\forall \text{pid} \in \text{Prefix}(H(0, \text{pathLists})), \text{pid} \in \text{CT}(0)$ and hence Lemma 4.7.13 is vacuously tree.

*Induction Hypothesis:* Suppose the lemma holds for loop $# n$, and we need to show it also holds for loop $# n+1.

Given a list $l$, if $Q(t, l) = \{ x \in \text{Prefix}(l,\text{QPTNode}, \text{id}, q) \mid q \in \text{anc}(l,\text{QPTNode}) \land \text{id} \in l \land \text{Qualified}(x, Q, KW, H(t, \text{pathLists})) = \text{true} \}$ is the set of qualified ids in $l$ at a given loop $# t$, and $Q(t) = \{ x \in Q(t, l) \mid l \in H(t, \text{pathLists}) \}$ is the set of all qualified ids at the loop $# t$, we prove the lemma in three different cases, one for each different case of $\text{id} \in Q(n+1)$. (a) $\text{id} \in Q(n)$, i.e., $\text{id}$ is already qualified at the loop $# n$; (b) $\text{id} \in \text{Prefix}(H(n, \text{pathLists})) \land \text{id} \notin Q(n)$, i.e., $\text{id}$ is in $\text{Prefix}(H(n, \text{pathLists}))$ and just becomes qualified at the loop $# n+1$; and (c) $\text{id} \in \text{Prefix}(H(n+1, \text{pathLists})) \land \text{Prefix}(H(n, \text{pathLists}))$, i.e., $\text{id}$ is just introduced at the loop $# n+1$.

*Case a: $\text{id} \in Q(n)$.* In this case, $\text{id}$ is already qualified at the loop $# n$. Therefore by I.H., $\text{id} \in \text{CT}(n), \text{id} \in \text{GenPDT}(n), \text{or} \exists \text{cn} \in \text{CT}(n), \text{id} \in \text{cn.pdtCache}$. Now we discuss these three cases separately.

*Case a.1.* First, if $\text{id} \in \text{GenPDT}(n)$, then by the algorithm $\text{GenPDT}(n) \subseteq \text{GenPDT}(n+1)$, we know that $\text{id} \in \text{GenPDT}(n+1)$.

*Case a.2.* Second, if $\text{id} \in \text{CT}(n)$, there are further two different mini-cases.

*Case a.2.1.* First, if $\text{id} \notin \text{CT}((n+1) \rightarrow \rightarrow ).\text{LeftMostPath}$, then by the algorithm, we know that $\text{id}$ will not be processed at the loop $# n+1$, and hence $\text{id} \in \text{CT}(n+1)$.

*Case a.2.2.* Second, if $\text{id} \in \text{CT}((n+1) \rightarrow \rightarrow ).\text{LeftMostPath}$, assume $\text{cn}$ is the node in $\text{CT}((n+1) \rightarrow \rightarrow ).\text{LeftMostPath}$ s.t. $\text{id}(\text{cn})=\text{id}$. Since $\text{Qualified}(\text{id}, Q, KW, H(n,\text{pathLists}))=\text{true}$, by definition we know that $\exists \text{cnq} \in \text{cn.\text{CTQPTNodeSet}}, \forall \text{cn} \in \text{CNQPTNodeSet}$.
MC(cnq), ((cnq.QPTNode, cnc, 'l', 'm') ∈ Q.E ⇒ ∃ce ∈ CE(cnc, T(H(n, pathLists)). root), parent(id(cn), ce.id)) ∧ ((cnq.QPTNode, cnc, 'l', 'm') ∈ Q.E ⇒ ∃ce ∈ CE(cnc, T(H(n, pathLists)). root), anc(id(cn), ce.id)) (*). Hence at the loop n+1, by Lemma 4.7.7, we know that ∀qcd ∈ MC(cnq), cnq.DescendantMap [qcd] = 1.

Further, also by Qualified(id, Q, KW, H(n,pathLists))=true, we know that ∃cqn ∈ cn.CTQPTNodes s.t. cqn satisfies the property (*) as described above and (cnq.ParentList = ∅ ∨ ∃cnp ∈ CT((n+1)−−).LeftMostPath, ∃p ∈ cnp.CTQPTNodeSet, p ∈ cnq.ParentList ∧ cnp ∈ PE(p, T(H(n, pathLists)).root).

Then by Lemma 4.7.9, at the loop n+1, p.InPdt = true. Hence by lines 2 -5, id ∈ GenPDT(n+1).

Case a.3. Third, if ∃cn ∈ CT(n).LeftMostPath, id ∈ cn.pdtCache. If cn ∈ CT(n+1), then by the algorithm the nodes in cn.pdtCache will not be removed, and hence the lemma holds. Otherwise we can use the same argument as in Case a.2.2 and show that cn.id ∈ GenPDT(n+1).

Case (b): id ∈ H(n, pathLists) ∧ id ∉ Q(n). First, since id ∈ Q(n+1), by Lemma 4.7.11, we know that id ∈ CT(n) ∨ ∃cn ∈ CT(n), id ∈ cn.pdtCache. Then we can use the similar argument to reason $CT((n+1)−−)$, as in Case a.2 and a.3, and show the lemma holds.

Case (c): id ∈ Prefix(H(n+1, pathLists))− H(n, pathLists). In this case, the algorithm will first add id in CT(n) and then process $CT((n+1)−−)$. LeftMostPath. Then if id ∉ CT(n). LeftMostPath, the lemma is vacuously true; otherwise we can prove the lemma using the same argument as in Case a.2 and Case a.3.

Therefore the lemma holds for all ids in Q(n+1).

We now prove Lemma 4.7.14.
Proof. We prove the lemma by induction on the loop # t.

*Base case t = 0:* It is vacuously true because GenPDT(t) = null.

*Induction Hypothesis:* Assume the lemma holds for the loop # t ≤ n, we show that it also holds for loop # n+1.

First, ∀id ∈ GenPDT(n) ∩ GenPDT(n+1), by I.H., we know that ∃id ∈ Prefix(H(n, pathLists)) ∧ Qualified(id, Q, KW, H(n, pathLists)) = true.

Therefore by lemma 4.7.6, we know Qualified(id, Q, KW, H(n+1, pathLists)) = true, and hence the lemma holds.

Now we prove the lemma for all g ∈ (GenPDT(n+1) − GenPDT(n)). By the algorithm there are three possible cases, one for each different scenario where g is created in GenPDT(n+1).

*Case 1:* g.id ∈ CT((n + 1) − −). In this case, since g is in GenPDT(n+1), by line 5 we know that ∃q ∈ g.CTQPTNodes, q.InPdt = true, and hence by Lemma 4.7.9, Qualified(g.id, Q, KW, H(n+1, pathLists)) = true.

*Case 2:* ∃cn ∈ CT((n + 1) − −), g.id ∈ cn.PDTCache. Since g is created in GenPDT(n+1), by line 21 in Figure 4.32, we know that either (1) ParentList(g) = ∅ or (2) ∃p ∈ ParentList(g), p.inPDT = true. Hence by Lemma 4.7.10, we know that Qualified(g.id, Q, KW, H(n+1, pathLists)) = true.

*Case 3:* g.id ∈ Prefix(H(n+1, pathLists)) - Prefix(H(n, pathLists)) In this case, g.id ∈ CT((n + 1) − −), and hence we can use the similar argument to Case 1 to show the lemma holds.

□
Proofs of supporting lemmas for Lemma 4.7.5

Proof of Lemma 4.7.7

Proof. First, if \( \text{MC}(\text{cnq}) = \emptyset \), then the lemma is vacuously true and hence we only consider the case where \( \text{MC}(\text{cnq}) \neq \emptyset \).

\[ \implies \]

We prove the inductions on \( \# t \).

Base case: \( t = 0 \). In this case, \( \text{CT}(0) \) is empty and thus the lemma is vacuously true.

Induction Hypothesis: Assume the lemma holds for \( t \leq n \), we show that it also holds for \( t = n+1 \).

Note by the algorithm, \( n+1 \) and \( n \) can be in the same or different loops in lines 8-19. However, since we never modify the value of DescendantMap after adding IDs, we do not differentiate these two cases.

Now we assume that after adding the ID at the time \( n+1 \), given a \( \text{cn} \in \text{CT}_{n+1}, \text{cnq} \in \text{cn.\text{CTQPTNodeSet}}, \text{qcd} \in \text{MC}(\text{cnq}), \text{cnq.DescendantMap}[\text{qcd}] = 1 \). There are four different cases to consider. (a) \( \text{cn} \in \text{CT}_n \land \text{cnq} \in \text{CT}_n \land \text{cnq}[\text{qcd}] = 1 \) at the time \( n \), and (b)\( \text{cn} \in \text{CT}_n \land \text{cnq} \in \text{CT}_n \land \text{cnq}[\text{qcd}] = 0 \) at the time \( n \), and (c)\( \text{cn} \in \text{CT}_n \land \text{cnq} \notin \text{CT}_n \), and (d)\( \text{cn} \notin \text{CT}_n \).

Case (a). In this case, by I.H., we know that \( ((\text{cnq.\text{QPTNode}}, \text{qcd}, 'L', 'm') \in \text{Q.E} \Rightarrow \exists l \in \text{List}_n, \exists \text{id} \in l, \exists \text{cid} \in \text{Prefix}(l.\text{QPTNode}, \text{id}, \text{qcd}), \exists \text{ce} \in \text{CE}(\text{qcd}, \text{T}((\text{List}_n).\text{root})), \text{ce}.\text{id} = \text{cid} \land \text{id}((\text{cn})) \in \text{Prefix}(l.\text{QPTNode}, \text{id}, \text{cnq.\text{QPTNode}}) \land \text{parent}((\text{id}((\text{cn})), \text{cid})) \land ((\text{cnq.\text{QPTNode}}, \text{qcd}, 'L', 'm') \in \text{Q.E} \Rightarrow \exists l \in \text{List}_n, \exists \text{id} \in l, \exists \text{cid} \in \text{Prefix}(l.\text{QPTNode}, \text{id}, \text{cnq.\text{QPTNode}}) \land \text{anc}((\text{id}((\text{cn})), \text{cid})))\).
Hence by the definition of candidate elements, it is easy to infer that \(((\text{cnq.QPTNode}, \text{qcd}, '/', 'm') \in Q.E \Rightarrow \exists l \in List_{n+1}, \exists lid \in l, \exists cid \in \text{Prefix}(l, \text{QPTNode}, \text{lid}, \text{qcd}), \exists ce \in \text{CE}(\text{qcd}, T(List_{n+1}).\text{root}), ce.\text{id} = \text{id}(\text{cn}) \land \text{id}(\text{cn}) \in \text{Prefix}(l, \text{QPTNode}, \text{lid}, \text{cnq.QPTNode}) \land \text{parent}(\text{id}(\text{cn}), \text{cid})) \land ((\text{cnq.QPTNode}, \text{qcd}, '/\', 'm') \in Q.E \Rightarrow \exists l \in List_{n+1}, \exists lid \in l, \exists cid \in \text{Prefix}(l, \text{QPTNode}, \text{lid}, \text{qcd}), \exists ce \in \text{CE}(\text{qcd}, T(List_{n+1}).\text{root}), ce.\text{id} = \text{id}(\text{cn}) \land \text{id}(\text{cn}) \in \text{Prefix}(l, \text{QPTNode}, \text{lid}, \text{cnq.QPTNode}) \land \text{parent}(\text{id}(\text{cn}), \text{cid}))\).

Hence the lemma holds.

Case (b) In this case, we show the lemma by induction on the depth of qcd.

Base case: qcd is the leaf node. In this case, since cnq. DescendantMap[qcd] is set to 1, if \text{nid} \in l the id we add at the time n+1, then we can infer that \exists qid \in \text{Prefix}(l, \text{QPTNode}, \text{nid}, \text{qcd}). Further, since qcd is the leaf node, by definition of candidate elements and by the specification of path index, we know that \exists qid \in \text{CE}(\text{qcd}, T(List_{n+1}).\text{root}). Further since we set cnq.DescendantMap[qcd] = 1, we know that cnq \in qcd.ParentList, therefore we can finally conclude that \((\text{cnq.QPTNode}, \text{qcd}, '/', 'm') \in Q.E \Rightarrow \exists l \in List_{n+1}, \exists lid \in l, \exists cid \in \text{Prefix}(l, \text{QPTNode}, \text{lid}, \text{qcd}), \exists ce \in \text{CE}(\text{qcd}, T(List_{n+1}).\text{root}), ce.\text{id} = \text{id}(\text{cn}) \land \text{id}(\text{cn}) \in \text{Prefix}(l, \text{QPTNode}, \text{lid}, \text{cnq.QPTNode}) \land \text{parent}(\text{id}(\text{cn}), \text{cid})) \land ((\text{cnq.QPTNode}, \text{qcd}, '/\', 'm') \in Q.E \Rightarrow \exists l \in List_{n+1}, \exists lid \in l, \exists cid \in \text{Prefix}(l, \text{QPTNode}, \text{lid}, \text{qcd}), \exists ce \in \text{CE}(\text{qcd}, T(List_{n+1}).\text{root}), ce.\text{id} = \text{id}(\text{cn}) \land \text{id}(\text{cn}) \in \text{Prefix}(l, \text{QPTNode}, \text{lid}, \text{cnq.QPTNode}) \land \text{anc}(\text{id}(\text{cn}), \text{cid}))\).

Induction Hypothesis: Assume the lemma holds for qcd of depth \(\geq d\), we need to show the lemma also holds for d-1.

If \text{MC}(qcd) = \emptyset, then we can use the similar argument as the base case and show the
lemma holds. Otherwise $\text{MC}(\text{qcd}) \neq \emptyset$.

There are two mini-cases here, depending on whether there exists a child node of $\text{cn}$ in $CT_n$ which contains $\text{qcd}$ in its $\text{CTQPTNodeSet}$.

First, assume $\exists qn \in CT_n, \exists qc \in \text{cn.CTQPTNodeSet}$, $\text{qcd} = \text{qc.QPTNode} \land \forall mq \in \text{MC(qc)}$, $\text{q}.\text{DescendantMap}[mq] = 1$ at the time $n+1$. In this case, since at the time $n$, $\text{cnq}.\text{DescendantMap}[\text{qcd}] = 0$, intuitively we know that certain descendant restrictions of $\text{qcd}$ are not satisfied at the time $n$. If $X = \{x | x \in \text{MC(\text{qcd}) wedge q}.\text{DescendantMap}[x] = 1 \text{ in } CT_n\}$, and $Y = \{y | x \in \text{MC(\text{qcd}) wedge q}.\text{DescendantMap}[y] = 0 \text{ in } CT_{n+1}\}$.

Then by I.H. on the number $n$, we know at the time $n$, the lemma holds for all $x$ in $X$. Further, by I.H. on the depth, we know the lemma also holds for all $y$ in $Y$.

Therefore we know that at the time $n+1$, $\forall mq \in \text{MC(qc)}$, $(\text{qcd}, mq, '\prime', '\prime$') $\in$ $\text{Q.E}$ $\Rightarrow$ $\exists l \in List_{n+1}$, $\exists \text{lid} \in l$, $\exists \text{cid} \in \text{Prefix(l.QPTNode, lid, mq)}$, $\exists ce \in \text{CE(mq, T(List}_{n+1}).\text{root})}$, $\text{ce.id} = \text{cid} \land \text{id(qn)} \in \text{Prefix(l.QPTNode, lid, qcd)} \land \text{parent(id(qn), cid))}$ $\land$

$((\text{cnq}.\text{QPTNode, qcd, '\prime', '\prime$'}) \in \text{Q.E} \Rightarrow \exists l \in List_{n+1}$, $\exists \text{lid} \in l$, $\exists \text{cid} \in \text{Prefix(l.QPTNode, lid, qcd)}$, $\exists ce \in \text{CE(qcd, T(List}_{n+1}).\text{root})}$, $\text{ce.id} = \text{cid} \land \text{id(\text{cn})} \in \text{Prefix(l.QPTNode, lid, cnq.QPTNode)} \land \text{parent(id(\text{cn}), cid))}$ $\land$

$((\text{cnq}.\text{QPTNode, qcd, '\prime', '\prime$'}) \in \text{Q.E} \Rightarrow \exists l \in List_{n+1}$, $\exists \text{lid} \in l$, $\exists \text{cid} \in \text{Prefix(l.QPTNode, lid, qcd)}$, $\exists ce \in \text{CE(qcd, T(List}_{n+1}).\text{root})}$, $\text{ce.id} = \text{cid} \land \text{id(\text{cn})} \in \text{Prefix(l.QPTNode, lid, cnq.QPTNode)} \land \text{parent(id(\text{cn}), cid))}$.

Therefore $qn \in \text{CE(qcd, T(List}_{n+1}).\text{root})$, and hence $(\text{cnq}.\text{QPTNode, qcd, '\prime', '\prime$'}) \in \text{Q.E}$ $\Rightarrow$ $\exists l \in List_{n+1}$, $\exists \text{lid} \in l$, $\exists \text{cid} \in \text{Prefix(l.QPTNode, lid, qcd)}$, $\exists ce \in \text{CE(qcd, T(List}_{n+1}).\text{root})}$, $\text{ce.id} = \text{cid} \land \text{id(\text{cn})} \in \text{Prefix(l.QPTNode, lid, cnq.QPTNode)} \land \text{parent(id(\text{cn}), cid))}$ $\land$

Second, if $\nexists qn \in CT_n$, $\exists qc \in \text{cn.CTQPTNodeSet}$, $\text{qcd} = \text{qc.QPTNode}$. In this case,
since we only add a single Dewey id, we can use the similar induction as in the first case from bottom up and show the lemma holds.

**Case (c) and (d)** In this case, we just add the QPT node cqn at the time n+1. Then we can also show the lemma using an easy induction on the depth of qcd, similar to Case (b).

“⇐”

We prove the inductions on # t.

**Base case:** t = 0. In this case, no IDs have been retrieved and CT(0) is empty and hence the lemma is vacuously true.

**Induction Hypothesis:** Assume the lemma holds for t ≤ n, we show that it also holds for t = n+1.

If B denote the RHS of the statement, then given a cn ∈ CT_{n+1}, cnq ∈ cn.CTQPTNode, qcd ∈ MC(cqn.QPTNode), there are five cases to consider. (a) cn ∈ CT_n ∧ cnq ∈ CT_n ∧ qcd ∈ CT_n ∧ B = true, and (b) cn ∈ CT_n ∧ cnq ∈ CT_n ∧ qcd ∈ CT_n ∧ B = false, and (c) cn ∈ CT_n ∧ cnq ∈ CT_n ∧ qcd ∉ CT_n ∧ B = false, and (d) cn ∈ CT_n ∧ cnq ∉ CT_n ∧ qcd ∉ CT_n ∧ B = false, and (e) cn ∉ CT_n ∧ cnq ∉ CT_n ∧ qcd ∉ CT_n ∧ B = false.

We now show each of them separately.

**Case (a)** In this case, by I.H., we know that cnq. DescendantMap[qcd] = 1. By the algorithm, we never change the value from 1 to 0, and hence the lemma holds.

**Case (b)** In this case, we can infer that cid ∉ CE(qcd, T(List_{n}).root). But since we assume that cid ∈ CE(qcd, T(List_{n+1}).root), we know that MC(qcd) ≠ ∅, and ∃mq ∈ MC(qcd), (qcd, mq, ’/’, ’m’) ∈ Q.E. ⇒ ∃l ∈ List_n, ∃lid ∈ l, ∃mid ∈ Prefix(l.QPTNode, lid, mq), mid ∈ CE(mq, T(List_n).root), ∧ cid ∈ Prefix(l.QPTNode, lid, qcd) ∧ parent(cid, mid)) ∧
((qcd, mq, '/l', 'm') ∈ Q.E ⇒ ∃ l ∈ List_n, ∃ lid ∈ l, ∃ mid ∈ Prefix(l.QPTNode, lid, mq), mid ∈ CE(mq, T(List_n).root) ∧ cid ∈ Prefix(l.QPTNode, lid, qcd) ∧ anc(cid, mid)) (*).

We assume qn is a node in CT_n + 1 and CT_n s.t. qn.id = cid. we say X = \{x | x ∈ MC(qcd), property (*) does not hold\}, and Y = \{y | y ∈ MC(qcd), property (*) holds\}.

First, for all x in X, by I.H., we know that the lemma holds. Hence ∀x ∈ X, if qnc ∈ qn.CTQPTNode and qnc.QPTNode = qcd, then qnc.DescendantMap[x] = 1. For all y in Y, we can show that qnc.DescendantMap[y] = 1 in CT_n+1 by induction on the depth of y. If y is the leaf node and cy is the CT node corresponding to y, then by the algorithm we will set qnc.DescendantMap[y] to be 1. Inductively, if y is the non-leaf node. Then if MC(y) = \emptyset, by the algorithm, we will also set qnc.DescendantMap[y] = 1. Otherwise by I.H. on the depth, we know for all yy ∈ MC(y), the corresponding entries in DescendantMap are set to 1, and hence qnc.DescendantMap[y] is set to 1. Hence by the algorithm, cnq.DescendantMap[qcd] is set to 1. Hence the lemma holds in this case.

Case (c), (d), and (e) In all of these cases, we can prove induction on the depth of qcd in a similar fashion to Case (b). As the base case, if qcd is the leaf node, if did ∈ l is the single Dewey ID that we add to CT_n+1, we know that cid ∈ Prefix(l.QPTNode, did, qcd), and hence by the algorithm, we will set cnq.DescendantMap[qcd] to be 1. Inductively, if qcd is a non-leaf node, then if MC(qcd) = \emptyset, we can show the lemma similar to the base case. Otherwise by I.H. on the depth, if cnn is the CT node s.t. ∃ qn ∈ cnn.CTQPTNodeSet, qn.QPTNode = qcd, then ∀x ∈ MC(qcd), qn.DescendantMap[x] = 1. And hence by the algorithm, we will set cnq.DescendantMap[qcd] to be 1.
Proof of Lemma 4.7.9

Proof. We first prove (a) by induction on the loop # t.

Base case: \( t = 0 \). The lemma is vacuously true since in \( CT(0-) \), we do not change the values of InPdt from false to true.

Induction hypothesis: Assume the lemma holds for loop # \( \leq t \), we show it also holds for loop # \( t+1 \).

First, if \( cn \in CT(t-) \) and \( cnq \in cn.CTQPTNodeSet \) and \( cnq.InPdt = true \), then by I.H., we know that the lemma holds. Otherwise the value of inPdt is set to true at the loop # \( t+1 \).

We show the lemma holds in this case by inductions on the depth of nodes, starting from the root.

Base case: depth = 0. In this case, the node \( cn \) is the root node and hence we know that \( \forall cnq \in cn.CTQPTNodeSet \), \( cnq.QPTNode \) is also the root node (in fact, by definition there is only a single node in \( cn.CTQPTNodeSet \) in this case). Hence \( cnq.ParentList = \emptyset \), and by line 4 in Figure 4.32, \( cnq.inPdt \) is set to true when \( \forall i \in cnq.DescendantMap[i] = 1 \). By Lemma 4.7.7, this implies that \( cn \in CE(cnq.QPTNode, T(H(t, pathLists)).root) \). Therefore by definition of PE, we know that \( cn \in PE(cnq.QPTNode, T(H(t, pathLists)).root) \), and hence the lemma holds.

Induction hypothesis: Assume the lemma holds for nodes of depth \( \leq n \), we now show the lemma also holds for nodes of depth \( n+1 \).

Given \( cnq \in cn.CTQPTNodes \), if \( cnq.inPdt = true \), then by lines 4-5 in Figure 4.32, we know that \( \forall i \in cnq.DescendantMap[i] = 1 \). By Lemma 4.7.7, this implies that \( cn \in CE(cnq.QPTNode, T(H(t+1, pathLists)).root) \). We also know that \( cnq.ParentList = \emptyset \) or \( \exists p \in cnq.ParentList \), If \( cnq.ParentList = \emptyset \), then \( cnq \) is the root node in the QPT and by
definition, \( cn \in \text{PE}(cnq.QPTNode, T(H(t+1, pathLists)).root) \). If \( \exists p \in cnq.ParentList, q.\text{inPdt} = \text{true} \), and if \( cnp \) is the CT node s.t. \( p \in cnp. \text{CTQPTNodeSet} \), then by the algorithm, we know that \( cnp \) is an ancestor node of \( p \). Then if \( cnp.\text{InPdt} = \text{true} \) before the loop \# \( t+1 \), we can apply I.H. on the loop \# \( t \) and using Lemma 4.7.6 to infer that \( cnp \in \text{PE}(p, T(H(t+1, pathLists)).root) \); otherwise we can use I.H. on the depth of nodes and infer that \( cnp \in \text{PE}(p, T(H(t+1, pathLists)).root) \). Hence by definition of PDT, we know that \( cnp \in \text{PE}(p, T(H(t+1, pathLists)).root) \). Hence (a) holds.

(b) We only show that \( \forall n \in CT(t-).\text{LeftMostPath}, \forall nq \in n.\text{CTQPTNodeSet}, cn \in \text{PE}(nq.QPTNode, T(H(t+1, pathLists)).root) \Rightarrow nq.\text{InPdt} = \text{true} \).

It is straightforward to infer that \( \forall n \in CT(t-).\text{LeftMostPath}, \forall nq \in n.\text{CTQPTNodeSet}, cn \in \text{PE}(nq.QPTNode, T(H(t, pathLists)).root) \Rightarrow nq.\text{InPdt} = \text{true} \wedge (\forall t' \geq t, n \in CT(t') \Rightarrow nq.\text{InPdt} = \text{true} \wedge n \in CT(t'-) \Rightarrow (nq \in n.\text{CTQPTNodeSet} \land nq.\text{InPdt} = \text{true}) \land n \in CT(t'-)) \) because we never change the flag from true to false.

We now prove (b) by induction on the loop \# \( t \).

**Base case: \( t = 1 \).** We prove the lemma holds in this case by induction on the depth of the nodes in \( CT(1-) \).

**Base case: depth = 0.** In this case, the node \( cn \) is the root node and hence we know that \( \forall cnq \in cn.\text{CTQPTNodeSet}, cnq.QPTNode \) is also the root node. In fact, by definition there is only a single node in \( cn.\text{CTQPTNodeSet} \), call it \( sq \). Since \( cn \in \text{PE}(sq.QPTNode, T(H(t, pathLists)).root) \), we can infer that \( cn \in CE(sq.QPTNode, T(H(t, pathLists)).root) \). Hence by Lemma 4.7.7, \( \forall i \in cnq.\text{DescendantMap}[i] = 1 \). Further, since \( sq \) is the root node in the QPT, \( sq.\text{ParentList} = \varnothing \). Then by line 4 in Fig-
Induction hypothesis: Assume the lemma holds for nodes of depth \( \leq n \), we now show the lemma also holds for nodes of depth \( n+1 \).

Given \( cnq \in cn.CTQPTNodes \), if \( cnq \) is the root node in the PDT, then we can use the similar argument to the base case and show (b) holds. If \( cnq \) is non-root node and assume \( p \) is the parent node of \( cnq \). Assume \( cnp \) is an ancestor node of \( cn \) and \( p \in cnp.CTQPTNodes \), then \( cn \in PE(cnq, T(H(t+1, pathLists)).root) \) implies that \( cnp \in PE(p, T(H(t+1, pathLists)).root) \). By I.H., we know that \( cnp.InPdt = true \). Further \( cn \in PE(cnq, T(H(t+1, pathLists)).root) \) also implies that \( cn \in CE(cnq, T(H(t+1, pathLists)).root) \), hence by Lemma 4.7.7, \( \forall i \in cnq.DescendantMap[i] = 1 \). Therefore by lines 4-5 in Figure 4.32, \( cnq.InPdt \) will be set to true.

Hence (b) holds in the base case.

Induction hypothesis: Assume the lemma holds for loop \( \# \leq t \), we now show the lemma also holds for loop \( \# t+1 \).

Given \( cn \in CT((t+1)−).LeftMostPath \), \( cnq \in cn.CTQPTNodes \), if \( cn \in CT(t−).LeftMostPath \) and \( id(cn) \in PE(cnq, T(H(t, pathLists)).pathLists) \), the by I.H., we know that \( cnq.InPdt = true \) at the loop \( \# t \). Since we never change it from true to false, the lemma holds.

Otherwise \( cnq.InPdt \) is set to true at the loop \( \# t+1 \). We can use the similar induction on the depth of the nodes as in the base case to show the lemma holds.

\( \square \)

Proof of Lemma 4.7.10

Proof. (a) It is easy to prove (a) by lines 11 in Figure 4.32 using Lemma 4.7.7.
(b) We prove (b) by considering different cases corresponding to when the node is created in the pdt cache and when the parent list is updated. At the loop # t, given \( cn \in CT(t+1), x \in cn.PdtCache \), if \( x \) is just created in \( cn.PdtCache \), then by definition of ParentList, we know that if \( \forall q \in x.PDTQPTNodes, \forall p \in q.ParentList, \text{Qualified}(id(p), H(t, pathLists)) = \text{false} \) implies \( \text{Qualified}(id, H(t, pathLists)) = \text{false} \).

Otherwise \( x \) is created at loop \( x \leq t \) and is updated. We can show the lemma by inductons on the number of update times. The base case is just shown. Inductively, we assume the (b) holds for the case where ParentList(\( x \)) is updated \( n \) times. Now ParentList(\( x \)) is updated again by line 28 in Figure 4.32. Assume \( q \) is replaced by \( q.ParentList \), by definition we know that \( \forall qp \in q.ParentList, \text{Qualified}(id(qp), H(t, pathLists)) = \text{false} \) implies \( \text{Qualified}(id(q), H(t, pathLists)) = \text{false} \). Further, we know that by I.H., \( \text{Qualified}(id(q), H(t, pathLists)) = \text{false} \) implies that \( \text{Qualified}(x.id, H(t, pathLists)) = \text{false} \). Since we assume \( \forall qp \in q.ParentList, \text{Qualified}(id(qp), H(t, pathLists)) = \text{false} \), we can conclude that \( \text{Qualified}(x.id, H(t, pathLists)) = \text{false} \).

(c) can also be shown in a similar fashion as in (b).

\[ \square \]

**Proof of Lemma 4.7.11**

**Proof.** We can prove this lemma by induction on \( t \).

**Base case:** \( t = 0 \). In this case, all ids in \( H(t, pathLists) \) are in \( CT(t) \) and therefore the lemma is vacuously true.

**Induction Hypothesis:** Assume the lemma holds for \( t \leq n \), we need to show the lemma holds for \( n+1 \).

We show an equivalent statement as follows.
\textit{pid} \notin \text{CT}(n+1) \land \textit{pid} \notin \text{pdtCache}(\text{CT}(n+1)) \land \text{Qualified}(\textit{pid}, \text{Q}, \text{KW}, \text{H}(n+1, \text{pathLists})) = \text{false} \Rightarrow \nexists L \in \text{Comp}(\text{H}(n+1, \text{pathLists})), \text{Qualified}(\textit{pid}, \text{Q}, \text{KW}, L) = \text{true}.

There are two cases to consider depending on whether \textit{pid} is in Prefix(H(n, pathLists)).

\textit{Case 1: pid} \in \text{Prefix}(H(n, pathLists)) \text{ First, by Lemma 4.7.6, } \text{Qualified}(\textit{pid}, \text{Q}, \text{KW}, \text{H}(n+1, \text{pathLists})) = \text{false} \text{ implies } \text{Qualified}(\textit{pid}, \text{Q}, \text{KW}, \text{H}(n, \text{pathLists})) = \text{false}. \text{ Then we have two different cases to consider.}

\textit{Case 1.1: pid} \notin \text{CT}(n) \land \textit{pid} \notin \text{pdtCache}(\text{CT}(n)) \text{ In this case, we can use I.H. and infer that } \nexists L \in \text{Comp}(\text{H}(n, \text{pathLists})), \text{Qualified}(\textit{pid}, \text{Q}, \text{KW}, L) = \text{true}. \text{ This leads to the conclusion } \nexists L \in \text{Comp}(\text{H}(n+1, \text{pathLists})), \text{Qualified}(\textit{pid}, \text{Q}, \text{KW}, L) = \text{true} \text{ because } \text{Comp}(\text{H}(n+1, \text{pathLists})) \subseteq \text{Comp}(\text{H}(n+1, \text{pathLists})).

\textit{Case 1.2: pid} \in \text{CT}(n) \lor \textit{pid} \in \text{pdtCache}(\text{CT}(n)) \text{ In this case, since } \textit{pid} \notin \text{CT}(n+1) \land \textit{pid} \notin \text{pdtCache}(\text{CT}(n+1)), \text{we need to discuss when pid is removed at loop } \# \ n+1.

First assume \textit{pid} \in \text{CT}(n) \text{ and assume at loop } \# \ t, \text{ pid is never temporarily copied to any pdt cache. Intuitively, this case indicates that pid does not satisfy the descendant restrictions.}

By the algorithm there exists a node \textit{pn} in the left most path of \text{CT}(n+) \text{ and } \textit{pn.id} = \textit{pid}. \text{ By the algorithm } \text{pn must be removed by line 34 in Figure 4.32. By } \textit{pid} \notin \text{CT}(n+1), \text{ we can infer that } \forall q \in \textit{pn.CTQPTNodes}, \exists \textit{ch} \in \text{MC}(q.CTQPTNode), \textit{q.DescendantMap}[\textit{ch}] = 0. \text{ Further, we remove } \textit{pn} \text{ only when } \textit{pn.HasChild} = \text{false}. \text{ Also, by line 14 in Figure 4.32, we have already added next minimum ids corresponding to the left most path. Also, for all paths in the lists, the next minimum IDs are greater than their respective IDs in the current CT because they are ordered ID lists. This implies that } \forall L \in \text{Comp}(\text{H}(n+1, \text{pathLists})), \text{if } l \in L \text{ and } l.QPTNode = \textit{ch}, \text{ and if lid is the next id in } l, \text{ then}
we know that Prefix(l.QPTNode, lid, pn.QPTNode) is greater than pn.id, and hence q. DescendantMap[ch] will never be set to be 1. Therefore by Lemma 4.7.7, we know that \( \forall L \in \text{Comp}(H(n+1, pathLists)) \), \( \text{pid} \notin \text{CE}(pn.CTQPTNode, T(L).\text{root}) \), and hence Qualified(pид, Q, KW, L) = false.

Second, if \( \text{pid} \in \text{pdtCache}(CT(n)) \) or \( \text{pid} \) is in \( CT(n) \) but was later copied to \( \text{pdtCache} \) of some nodes when we are at loop \# \( n+1 \). Assume \( pn \) is the node in the \( \text{pdtCache} \) s.t. \( pn.\text{id} = \text{pid} \), and assume \( pn \in cn.\text{pdtCache} \). For simplification, we only consider the case where \( pn \) is removed when we process \( pn \). Intuitively, this case indicates that \( \text{pid} \) does not satisfy the ancestor restrictions.

This is handled by line 26 in Figure 4.32. Therefore we know that before we remove \( pn \), \( pn.\text{ParentList} = \{cn\} \) and \( \exists \text{ch} \in \text{MC}(cn) \), \( cn.\text{DescendantMap}[\text{ch}] = 0 \). By Lemma 4.7.7 and using the same argument as in the first case, we know that \( \forall L \in \text{Comp}(H(n+1, pathLists)) \), Qualified(cn.id, Q, KW, L) = false. Hence \( \forall L \in \text{Comp}(H(n+1, pathLists)) \), pn does not satisfy the ancestor restrictions, and therefore \( \forall L \in \text{Comp}(H(n+1, pathLists)) \), Qualified(pид, Q, KW, L) = false.

**Case 2: pid \( \in \text{Prefix}(H(n+1, pathLists)) - \text{Prefix}(H(n, pathLists)) \)** Note that in the algorithm, we first add ids in \( H(n+1, pathLists) - H(n, pathLists) \) (line 14 in Figure 4.32) and then process the left most path, therefore if \( CT(n') \) is the intermediate candidate tree after we add new ids to \( CT(n) \), then \( \text{pid} \in CT(n') \) and we can use the same argument in Case 1.2 to show that the lemma holds. The full proof is skipped here.

□

**Proves of correctness of PrepareList**

By Lemma 4.7.5, we know that once we exit the loop and the candidate tree becomes empty, all qualified ids w.r.t to pathLists are captured in PDT and PDT only contains...
qualified ids w.r.t pathLists. In other words, if GenPDT is the PDT that is produced upon termination of the loop, then \( \text{GenPDT} = \text{PDT}(Q, \text{KW}, \{ Q.\text{root} \Rightarrow \text{T(pathLists).root} \}) \).

We just need the following final lemma to show the Theorem 4.7.4 is true.

We first show two supporting lemmas.

Given a QPT \( Q \), a node \( n \in Q \), we say \( \text{RootToLeaf}(n, Q) \) is the path starting from the root node of \( Q \) and ends on \( n \), then we can show the following lemma.

**Lemma 4.7.15 (PathIndex).** Given a set of keyword \( \text{KW} \), a QPT \( Q \), an XML database \( D \), an environment \( \delta \in \text{UE}(D, Q) \), if \( \delta(Q.\text{root}) = d \), then \( \forall q \in \text{QPT}, \forall \ n \in D, n \in PE(q, \{ Q.\text{root} \Rightarrow d \}) \Rightarrow \text{id}(n) \in \delta.\text{PathIndex.LookUp}(\text{RootToLeaf}(q, Q)) \).

**Proof.** We prove the lemma by inductions on the depth of \( q \).

*Base case: depth = 0* In this case, \( q \) is the root node of \( Q \) and \( \text{RootToLeaf}(q, Q) = \{ q \} \). Hence \( \delta.\text{PathIndex.LookUp}(\text{RootToLeaf}(q, Q)) = \{ \text{id}(n) | \text{tag}(n) = q.\text{tag} \land \forall \ p \in q.\text{Predicates}, \text{satisfies}(n, q) \} \). Therefore \( \delta.\text{PathIndex.LookUp}(\text{RootToLeaf}(q, Q)) \) is a superset of \( PE(q, \{ Q.\text{root} \Rightarrow d \}) \). Hence the lemma holds.

*Induction hypothesis:* Assume the lemma holds for \( q \) of depth \( \leq d \), we need to show the lemma for \( q \) of depth \( d + 1 \).

Assume \( pq \) is the parent of \( q \). We now show the case where \( (pq, q, '/', \text{ann}) \in Q \), and the case where \( (pq, q, '//', \text{ann}) \in Q \) can be shown similarly.

By definition, \( \forall n \in PE(q, \{ Q.\text{root} \Rightarrow d \}), \exists np \in PE(q, \{ Q.\text{root} \Rightarrow d \}), \text{parent}(np, n) \). By I.H., we can infer that \( \forall n \in PE(q, \{ Q.\text{root} \Rightarrow d \}), \exists \text{pid} \in \delta.\text{PathIndex.LookUp}(\text{RootToLeaf}(np, Q)), \text{parent}(\text{pid}, \text{id}(n)) \). Therefore we can infer that \( \forall n \in PE(q, \{ Q.\text{root} \Rightarrow d \}), \text{id}(n) \in \delta.\text{PathIndex.LookUp}(\text{RootToLeaf}(q, Q)) \).

Hence the lemma holds. \( \square \)

**Lemma 4.7.16 (CandidateElements).** Given a set of keyword \( \text{KW} \), a QPT \( Q \), an XML
database\(D\), an environment \(\delta \in UE(D, Q)\), if \(\delta(Q.\text{root}) = d\) and \((\text{pathLists}, \text{invLists}) = \text{PrepareList}(Q, d.\text{PathIndex}, d.\text{InvIndex}, KW)\), \(\forall q \in Q, \forall n \in D, n \in PE(q, \{Q.\text{root} \Rightarrow d\}) \Rightarrow n \in CE(q, \{Q.\text{root} \Rightarrow T(\text{pathLists}).\text{root}\})\).

**Proof.** We prove the lemma by induction on the depth of \(q\).

**Base case:** \(q\) is the leaf node. In this case, \(\forall n \in PE(q, \{Q.\text{root} \Rightarrow d\})\). Since \(q\) does not have children nodes, we will issue \(d.\text{PathIndex}\text{LookUp}(\text{RootToLeaf}(q, Q))\). Therefore by Lemma 4.7.15, we can infer that \(\text{id}(n) \in \text{pathLists}\). Hence by the definition, \(n \in CE(q, \{Q.\text{root} \Rightarrow T(\text{pathLists}).\text{root}\})\).

**Induction hypothesis:** Assume the lemma holds for \(q\) of depth \(\geq d\), we need to show the lemma for \(q\) of depth \(d - 1\).

If \(MC(q) = \emptyset\), then by the algorithm we will issue \(d.\text{PathIndex}.\text{LookUp}(\text{RootToLeaf}(q))\). Hence similar to the base case we can show the lemma holds. Otherwise by definition of PDT and \(\forall n \in PE(q, \{Q.\text{root} \Rightarrow d\}), \forall cq \in MC(q), (q, cq, '/' , m) \in Q \Rightarrow \exists nc \in PE(cq, \{Q.\text{root} \Rightarrow d\}), \text{parent}(n, nc) \land (q, cq, '/' , m) \in Q \Rightarrow \exists nc \in PE(cq, \{Q.\text{root} \Rightarrow d\}), \text{anc}(n, nc)\).

Hence by I.H., we know that \(\forall n \in PE(q, \{Q.\text{root} \Rightarrow d\}), \forall cq \in MC(q), (q, cq, '/' , m) \in Q \Rightarrow \exists nc \in CE(cq, \{Q.\text{root} \Rightarrow T(\text{pathLists}).\text{root}\}), \text{parent}(n, nc) \land (q, cq, '/' , m) \in Q \Rightarrow \exists nc \in CE(cq, \{Q.\text{root} \Rightarrow T(\text{pathLists}).\text{root}\}), \text{anc}(n, nc)\).

Further, by the definition of Dewey ID, we know that \(\forall n \in PE(q, \{Q.\text{root} \Rightarrow d\}), \text{id}(n) \in T(d), \forall cq \in MC(q), (q, cq, '/' , m) \in Q \Rightarrow \exists nc \in CE(cq, \{Q.\text{root} \Rightarrow T(\text{pathLists}).\text{root}\}), \text{parent}(n, nc) \land (q, cq, '/' , m) \in Q \Rightarrow \exists nc \in CE(cq, \{Q.\text{root} \Rightarrow T(\text{pathLists}).\text{root}\}), \text{anc}(n, nc)\).

Therefore \(n \in CE(q, \{Q.\text{root} \Rightarrow T(\text{pathLists}).\text{root}\})\).

Hence the lemma holds. \(\square\)
Lemma 4.7.17 (PrepareList). Given a set of keyword KW, a QPT Q, an XML database D, an environment $\delta \in UE(D, Q)$, if $\delta(Q, root) = d$ and $(\text{pathLists}, \text{invLists}) = \text{PrepareList}(Q, d.\text{PathIndex}, d.\text{InvIndex}, KW)$, $\forall q \in Q, \forall n \in D, n \in PE(q, \{Q, root \Rightarrow d\})$ $\Leftrightarrow n \in PE(q, \{Q, root \Rightarrow T(\text{pathLists}).\text{root}\})$.

Proof. “$\Rightarrow$”

We prove this direction by induction on the depth of q.

Base case: depth = 0 In this case, q is the root node of the QPT. By definition, $PE(q, \{Q, root \Rightarrow d\}) = CE(q, \{Q, root \Rightarrow d\})$, and $PE(q, \{Q, root \Rightarrow T(\text{pathLists}).\text{root}\}) = CE(q, \{Q, root \Rightarrow T(\text{pathLists}).\text{root}\})$. By Lemma 4.7.16, we know $PE(q, \{Q, root \Rightarrow d\}) \subseteq CE(q, \{Q, root \Rightarrow T(\text{pathLists}).\text{root}\})$, and hence $PE(q, \{Q, root \Rightarrow d\}) \subseteq PE(q, \{Q, root \Rightarrow T(\text{pathLists}).\text{root}\})$. Therefore the lemma holds in the base case.

Induction Hypothesis: Assume the lemma holds for q of depth $\leq d$, now we show the lemma also holds for q of depth d+1.

Assume p is the parent node of q in Q. We show the case where (p, q, ‘/’, ann) $\in$ Q, the case where (p, q, ‘//’, ann) $\in$ Q can be shown similarly.

First, by Lemma 4.7.16, $\forall n \in PE(q, \{Q, root \Rightarrow d\}), n \in CE(q, \{Q, root \Rightarrow T(\text{pathLists}).\text{root}\})$. Then by definition, we know that $\forall n \in PE(q, \{Q, root \Rightarrow d\}), \exists np \in PE(p, PE(q, \{Q, root \Rightarrow d\}), parent(np, n))$. Hence by I.H. on nq, we know that $\forall n \in PE(q, \{Q, root \Rightarrow d\}), \exists np \in PE(p, \{Q, root \Rightarrow T(\text{pathLists}).\text{root}\}) parent(np, n)$. Hence $\forall n \in PE(q, \{Q, root \Rightarrow d\}), n \in CE(q, \{Q, root \Rightarrow T(\text{pathLists}).\text{root}\}) \land \exists np \in PE(p, \{Q, root \Rightarrow T(\text{pathLists}).\text{root}\}) parent(np, n)$.

Therefore $n \in PE(q, \{Q, root \Rightarrow T(\text{pathLists}).\text{root}\})$.

“$\Leftarrow$”: This direction follows from Lemma 4.7.6 because $\forall id \in \text{pathLists}, id \in D$. Hence the full proof is skipped.
Chapter 5

Related Work

In this chapter, we review past research that is related to trigger processing and ranked keyword search queries.

5.1 Trigger Processing

In this section, we describe previous work on trigger processing over relational/XML data, incremental maintenance of materialized views, deriving production rules for constraints violation, and XML publish/subscribe systems. All of these topics are relevant to our trigger processing.

5.1.1 Trigger processing over relational/XML data

There have been recent advances in supporting triggers in native XML databases [19, 91]. In [19], authors propose Active XQuery, an extension of XQuery which enables trigger definition and management; they also propose an architecture and algorithms for efficiently processing triggers over XML documents. Our work uses the syntax and semantics specified in [19] to define triggers over XML views. However, unlike our approach, these systems do not support triggers over (unmaterialized or incrementally maintained) XML views, and also do not exploit relational technology.

In contrast to XML triggers, which are specified on XML nodes, SQL(relational) triggers [40, 41] are fired when an event (INSERT, UPDATE, or DELETE) occurs on a specific relational table. Relational triggers have been studied well in the context of active database systems [40, 41], and implemented as a rule system. A rule consists of three parts, an event that causes the rule to be triggered, a condition that is checked
when the rule is triggered, and an action that is executed when the rule is triggered and its condition is true. When an SQL trigger is activated, it has access to the pre- and post-update versions of the affected rows through transition tables which are used in trigger actions.

Many commercial relational databases have also recently added built-in XML support. However, these systems do not support triggers over incrementally maintained (or unmaterialized) XML views, as they lack sophisticated XML data management capabilities. In contrast, our approach supports triggers over XML views using existing relational technology by leveraging SQL triggers.

5.1.2 Incremental maintenance of materialized views

As mentioned in Section 3.3, one of the main technical challenges in supporting triggers over virtual views is determining how updates to the base data translate into updates to the old and new values of rows produced by the view. This problem is similar to the problem of incremental view maintenance in relational/nested-relational/object-oriented/semi-structured databases [3, 4, 47, 48, 27, 61, 65, 95]. We now briefly describe these related work.

View mechanisms and algorithms for materialized view maintenance have been studied extensively in the context of relational model [27, 61, 65]. In this context, authors analyze algorithms and techniques proposed for incrementally maintaining nonrecursive views, outer-join views, and recursive views. In practice, incremental maintenance has been shown to dramatically improve performance for relational views.

Views are much richer in the nested relational model and object-oriented data model and, subsequently, algorithms for querying materialized views are significantly more intricate([57, 82, 101, 80]). There has been recent work on incremental maintenance
of views of semistructured data [3, 47, 48]. In particular, [3] studies the problem of incremental maintenance of views of semi-structured data. The work is based on the Object Exchange Model (OEM) [95] for semistructured data and on the Lorrel query language for OEM [4]. Given a view and a database update, the proposed algorithm produces a set of maintenance statements, evaluates them on the database to yield a set of view updates, and installs the updates in the view.

We now describe how our proposed techniques are related to the work described above. First, we do reuse some algorithms from the view maintenance literature [26, 27] to determine which updates on the base data can cause updates to the view. However, there are several subtle and important distinctions that require the development of new techniques.

First, most existing techniques for incremental view maintenance assume that the view is materialized, and thus focus on computing the new value based on the old value. In contrast, we assume that the view is virtual (hence the old value is not materialized) and thus have to selectively compute the old value and the new value.

Second, nested views introduce a new class of predicates that we call nested predicates, which require special handling to avoid incorrect results (in fact, if we directly use existing techniques, triggers over views with nested predicates will produce wrong results!). One of the contributions of this dissertation is a general algorithm that works with nested predicates.

Third, triggers over views need an efficient way to efficiently check whether the old values and new values, which may be deeply nested, are indeed different so that users do not receive spurious notifications (for example, special care has to be taken for functions such as max, which may return the same result even if the base data has been updated). To address this issue, we introduce a general class of views called injective views for
which this check can be performed very efficiently.

Finally, one other difference between our proposed approach and XML view maintenance is that we exploit relational triggers and query processing, which is important for our target applications. A more detailed description (with examples) of how our techniques related to incremental view maintenance can also be found in Section 3.3.1.

### 5.1.3 Deriving production rules for constraints violation

Our trigger processing techniques involve determining which events on which relational tables can cause the event specified in the XML trigger. This is similar to the problem of identifying events on the base tables that can violate constraints [26] and affect materialized views [27]. In [26], authors propose a general language for expressing integrity constraints and present a framework based on production rules for deriving invalidating operations on base tables that can cause violations of constraints. Later, in [27], authors propose a mechanism where productions rules are automatically generated for incrementally maintenance of materialized views. The generated rules consist of relational operations that update views based on the updates to base tables. We adopt a similar approach to identifying relevant events on base tables.

### 5.1.4 XML publish/subscribe System

Another class of systems that is related to our trigger processing architecture is XML publish/subscribe systems [34, 46, 66, 91, 111]. NiagaraCQ [34] is a large scale continuous query processing system that achieves the scalability at the level of millions of subscribing queries. The key technique proposed in [34] is grouping similar queries. Specifically, NiagaraCQ uses signatures to group similar subscribing queries together, and uses a table of constants extracted from the queries along with a join to evaluate a
group of queries simultaneously. This grouping optimization scheme is adapted in our trigger processing architecture for improved scalability. However, NiagaraCQ, like most of other publish/subscribe systems, does not leverage relational technology with respect to query processing and storage, and does not have access to the database states, while trigger systems support retrieving both pre- and post-update versions of the affected values and thus enables realizing more complex application logics. Further, NiagaraCQ does not consider updates to existing documents, which are key challenges in our trigger problem.

XPush [66] addressed the problem of efficiently evaluating XPath queries against a large number of incoming documents. Similar to [34], [66] proposed optimization techniques based on the concept of deterministic automaton to identify common subexpressions in XPath queries, and common subexpressions will be evaluated only once for all queries sharing them. YFilter [46] aims to provide fast, on-the-fly matching of XML encoded data to a large number of interest specifications, and transformation of the matching XML data based on recipient-specific requirements. Over the past two years, it has been developed into an efficient query processor on streaming data that high-capacity XML message brokering systems can be built on. However, like NiagaraCQ, XPush and YFilter do not consider updates.

Xyleme [91] considers the problem of monitoring the web data consisting of XML and HTML, not only at the page level (e.g., discovery of a new page within a certain semantic domain) but also at the element level (e.g., insertion of a new electronic product in a catalog). However, it only works with native XML engines and is not designed to work with relational engine or exploit SQL triggers.

Recently, in [111] authors present a system that exploits the relational database to efficiently match the documents with queries. However, they do not consider updates to
the pages, and they do not take advantage of relational triggers.

5.2 Ranked Keyword Search Queries

In this section, we describe the previous research advances on ranked keyword search queries over XML documents, ranked relational algebra, XML scoring and indexing, and integrating structure and keyword queries. All of these topics are related to the problem of ranked keyword search queries over virtual XML views.

5.2.1 Ranked keyword search queries over XML documents

Closely related to the problem of ranked keyword search queries over XML views are the problem of ranked keyword search over XML documents.

XRank [64] presents a ranking framework, namely ElemRank, for computing the score of each XML node with respect to query keywords, and also proposes algorithms for efficiently producing ranked results of keyword search queries over hyperlinked XML documents. ElemRank captures the objective importance of an XML element computed using the underlying hyperlinked structure. This is conceptually similar to PageRank [22] except that ElemRank is defined at the granularity of an element and takes the nested structure of XML into account. The proposed query processing algorithms take into account a two-dimensional notion of keyword proximity when computing the ranking for XML keyword search queries. The experimental evaluation shows that the proposed specialized index structures and query evaluation techniques provide significant space savings and performance gains.

There has been other recent work on integrating keyword search with structured XML querying [6, 18, 54]. Schmidt et al. [104] introduce the “meet” operator for XML,
which intended for returning the most specific result. They also present efficient algorithms for computing “meet” using relational-style joins and indices. Christophides et al. [38], and Lee et al. [83] present systems for querying structured documents. However, the above systems do not consider ranking, rank-based query processing algorithms, which are central to our keyword search problem.

The following systems support ranked XML keyword search. XIRQL [56] is an extension of XQL for information retrieval. Myaeng et al. [88] use term-occurrences to compute the ranked results over SGML documents. XXL [109] uses term occurrences and ontological similarity for ranking. DBXplorer [5] and DISCOVER [70] support keyword search over relational databases, but do not support information retrieval style ranking. Further, they are not directly applicable for XML and HTML documents, which cannot always be mapped to a rigid relational schema. BANKS [15], DataSpot [45] and Lore [58] support keyword search over graph-structured data. Some of these systems use hyperlinked structure (BANKS), and simple proximity (BANKS, Lore) for ranking. However, DataSpot [45] does not present any query evaluation algorithms, and Lore [58] can only support keyword searches where the result type is known. BANKS requires that all the data edges fit in memory, which is not feasible for large data sets. Chakrabarti et al. [28] use nested HTML tag and hyperlink information to compute ranks at the granularity of a document. In contrast, our problem require computing rankings at the granularity of an element because XML keyword search queries return elements.

In summary, most of the previous work assumes that either specialized index structures or the actual documents are available. In our problem, however, we are aiming at efficiently processing keyword search queries over non-materialized XML views where both index structures and the document content are not available during runtime. Hence previous techniques cannot be directly applied.
5.2.2 Ranked relational algebra

There has been a lot of recent interest on ranked query operators, such as ranked join and aggregation operators for producing top-k results (e.g., [23, 31, 50, 51, 74, 76]), where the focus is on evaluating complex queries over ranked inputs. Our work is complementary to them in the sense that we focus on identifying the ranked inputs for a given query (using PDTs). We now briefly review previous work in this area.

Top-k query processing is first studied in the middleware scenario or in RDBMS in a “piecemeal” fashion, i.e., focusing on specific operators or sitting outside the core of query engines. In middleware settings, various algorithms are proposed for rank aggregation on a set of objects, by merging multiple ranked lists [50, 51], or scheduling random accesses efficiently [23, 31], with the goal of minimizing number of accesses to objects. Although in a different setting, the works in [23, 31] explore the concept of upper bound scores that inspires us to design algorithms for top-k keyword queries over virtual XML views. A similar sampling approach was applied in [31] to schedule predicates.

In RDBMS, there have been several proposals to support answering top-k queries at application level or outside the core of query engines [32, 62, 63, 69], or for supporting special types of ranking queries [73, 89]. Recently, supporting top-k queries inside the relational query engine, in terms of physical query operators, has been proved to be an efficient approach that treats ranking as a basic database functionality [25, 73, 74, 76]. A stop operator is proposed in [25] to limit the cardinality of intermediate and query result, either conservatively by integrity constraints or aggressively with the risk of restarting the query plan. The order supported by the stop operator is from columns of relations in SQL queries. Aggregation of multiple ranking criteria is not considered.

In [74] a new operator is devised for supporting rank join query, where rank join
predicates coexist with Boolean join predicates. Instead of conducting normal join algorithms on Boolean join predicates, the rank-join operator progressively produces the join results. In [76] the relational query optimizer is further extended to utilize the rank-join operator in generating efficient query plans.

With respect to the approach of extending query algebra, [81] proposes an algebra for capturing the semantic of preference queries. In [98] an algebra is proposed for expressing complex queries over Web relations that are used to model Web repositories. The algebra extension focuses on capturing the semantic of application-specific ranking and order relationships over Web pages and hyperlinks, instead of enabling efficient query processing.

RankSQL [84] is the first piece of work that introduces a systematic and principled framework, by extending relational algebra and query optimizers, to support ranking as first-class construct in relational database systems. It made the first attempt to fully integrate ranking in database systems on both the logical algebra level and the physical implementation level. In particular, RankSQL introduces a rank operator and defines the rank-aware semantics for traditional operators in relational algebra. They use rank-join as one of the rank-aware operators and at the same time supply an algebraic foundation of such support. They propose a dimensional enumeration framework enumerates plans by two dual logical properties to handle both scheduling of rank operators and join order selection. Finally, they propose an algorithm for efficient top-K query processing in the context of rank-aware relational algebra.

As mentioned earlier, our work is complementary to the ones described above. However, there are new challenges when applying the proposed techniques in our context, and we refer the reader to the conclusion for more details.
5.2.3 XML scoring and indexing

One key aspect in our keyword processing architecture is XML scoring and indexing and there has been a large body of work in the information retrieval community on scoring and indexing [5, 13, 14, 64, 68, 71]. However, they make the assumption that the documents being searched are materialized. In this paper, we build upon existing scoring and indexing techniques and extend them for virtual views. There has also been some recent interest on context-sensitive search and ranking [20, 60], where the goal is to restrict the document collection being searched at run-time, and then evaluate and score results based on the restricted collection. In our terminology, this translates to ranked keyword search over simple selection views (e.g., restricting searches on books to just books with year > 1995). However, these techniques do not support more sophisticated views based on operations such as nested FLWOR expressions and joins, which are crucial for defining even simple nested views (as in our running example). Supporting such complex operations requires a more careful analysis of the view query and introduces new challenges with respect to index usage and scoring, which are the main focus of this paper.

5.2.4 Integrating structure and keyword search queries

The problem of keyword search over views is also related to integrating structure and keyword search queries. GTP [36] with TermJoin [8] were originally designed to integrate structure and keyword search queries. Since it is a general solution, it can also be applied to the problem of keyword search over views. However, there are two key aspects that make GTP with TermJoin less efficient in our context. First, GTP and TermJoin use relatively expensive structural joins to reconstruct the document hierarchy. Second, GTP requires accessing the base data to support value joins, which is again
relatively inefficient. In contrast, our approach uses path indices to efficiently create the
PDT hierarchy and retrieve join values, which leads to an order of magnitude improve-
ment in performance (Section 4.5). Further, since our system uses the regular query
evaluator, we support a larger XQuery grammar (that includes functions) than GTP be-
cause GTP relies on encoding the semantics of the entire query using a tree pattern,
which makes it more difficult to represent a substantial part of XQuery.
Chapter 6

Conclusion

Creating XML views has been shown to be a powerful way to publish relational data and XML documents in the Internet. Many techniques have been proposed for efficiently publishing the base data, and for evaluating queries over virtual XML views. However, emerging Internet applications bring about new requirements on XML views. We focus on two such new requirements in this dissertation. First, users may wish to specify triggers to monitor changes to the view; existing systems, however, do not support support this “active” feature. Second, users often want to issue ranked keyword search queries over XML views, especially when the size of XML views are large. Existing XML view systems, however, do not support such exploratory keyword queries. In this dissertation, we have presented novel techniques that addresses these new challenges.

6.1 Contributions

The first contribution of this dissertation is a complete solution for supporting triggers over XML views. Specifically, the three main contributions are: (1) a system architecture for supporting triggers over nested views of relational data, (2) an algorithm for identifying and computing changes in an nested view based on possibly deeply nested relational updates, and (3) the definition and use of a general class of views called \textit{injective views} for which we can efficiently check whether the old values and new values are different, without having to explicitly compare these (possibly deeply nested) values. We also show that how prior work on scalable trigger processing can be adapted for the nested view problem, and investigate the effects of using incrementally maintained \textit{relational materialized views} for evaluating triggers over nested views.
The second contribution of this dissertation is a system architecture and novel algorithms for efficiently evaluating ranked keyword searches over virtual XML views. The key idea is to use regular indices, including inverted list and XML path indices, that are present on the base data to efficiently evaluate keyword search over views. The indices are used to efficiently identify the portion of the base data that is relevant to the current keyword search query so that only the top ranked results of the view are actually materialized and presented to the user. For this purpose, we propose the notions of Query Pattern Tree (QPT) and Projected Document Tree (PDT) that capture part of the base data that is relevant to the query, and also propose efficient algorithms to produce QPTs and PDTs. We use the 500MB INEX dataset to evaluate the performance of the proposed techniques; the results show that our proposed architecture is indeed an efficient and scalable way to support ranked keyword searches over virtual XML views.

6.2 Directions for Future Work

There are many avenues for future work. In the area of trigger processing, while our proposed techniques have been shown to be effective and scalable, our performance results reveal an opportunity for applying our techniques to optimize view maintenance. The idea is that currently the use of relational materialized views for nested triggers actually results in a performance degradation. This is partly due to a lack of support for materialized views with nested predicates, which are common in nested views. Thus, it is interesting to investigate whether our general algorithm for detecting changes over nested views can be adapted for incrementally maintaining materialized views with nested predicates.

In the area of ranked keyword search queries, while using the regular query evaluator in our proposed system architecture already leads to significant performance improve-
ment over alternative approaches, we could instead use the techniques proposed for ranked query evaluation (e.g., [33, 49, 75]) to further improve the performance of our system. There are, however, new challenges that arise in our context because XQuery views may contain non-monotonic operators such as group-by. For example, when calculating the scores of our example view results (please refer to Chapter 4 for details of this example), extra review elements may increase both the tf values and idf values, and hence the overall score may increase or decrease (non-monotonic). Hence existing optimization techniques based on monotonicity are not directly applicable. Second, our proposed PDT algorithms may be applied to optimize regular queries because the algorithms efficiently generate the relevant pruned data, and only materialize the final results. our context because XQuery views may contain several levels of nesting, aggregations and value joins. Finally, our proposed PDT algorithms might be useful for optimizing regular XML queries because the algorithms efficiently generate the pruned data needed for evaluation, and only materialize the final results.
BIBLIOGRAPHY

[1] WWW consortium, XQuery 1.0: An XML query language.

[2] WWW consortium, XQuery 1.0 and XPath 2.0 formal semantics.


