Algebraic Methods for Incremental Maintenance and Updates of Views within XML Databases

By

Martin Hugh Goodfellow

A thesis submitted to the University of Strathclyde
for the degree of Doctor of Philosophy
Department of Computer and Information Sciences
October 2014

Examiner's Copy

Typeset in $\LaTeX\,2_\varepsilon$.

Except where acknowledged in the customary manner, the material presented in this thesis is, to the best of my knowledge, original and has not been submitted in whole or part for a degree in any university.

Martin Hugh Goodfellow

Acknowledgements

Firstly, I am very grateful for the continuous support I have received from my supervisor Dr. John Wilson. Without his help and patience this thesis would not have been possible. I am also grateful for my EPSRC stipend which funded me throughout my PhD, along with other financial support from the department and other sources which allowed me to attend conferences and summer schools.

I would also like to thank Prof. Angela Bonifati and Dr. Ioana Manolescu who supervised me throughout my view maintenance work, during my internship at Inria Saclay in Paris. I would also like to thank Angela for continuing to help supervise me with my view maintenance work and some of my view update work up to the end of my PhD.

I would also like to acknowledge Dr. Ela Pustulka who was my initial PhD supervisor before moving to Switzerland. I would like to thank Ela for getting me started on the PhD process and continuing to offer support throughout.

Finally, I would like to thank all my family and friends who supported me throughout the highs and lows of my PhD.

List of Publications

- A. Bonifati, M. H. Goodfellow, I. Manolescu, D. Sileo *Algebraic Incremental Maintenance of XML Views.* ACM Transactions on Database Systems (TODS), Volume 38 Issue 3, August 2013

- A. Bonifati, M. H. Goodfellow, I. Manolescu, D. Sileo *Algebraic Incremental Maintenance of XML Views.* (Proceedings of the 14th International Conference on Extending Database Technology (EDBT/ICDT 2011))

- M. H. Goodfellow, J. Wilson, E. Hunt *Composition of Biochemical Networks using Domain Knowledge Poster.* Nature Precedings 2010

- M. H. Goodfellow, J. Wilson, E. Hunt *Biochemical Network Matching and Composition.* (Proceedings of the 2010 EDBT/ICDT Workshops)

- M. H. Goodfellow, E. Hunt, D. McCafferty *reSearch: Enhancing Information Retrieval with Images.* Technical Report, University of Strathclyde, 2009

Abstract

Within XML data management the performance of queries has been improved by using materialised views. However, modifications to XML documents must be reflected to these views. This is known as the view maintenance problem. Conversely updates to the view must be reflected on the XML source documents. This is the view update problem. Fully recalculating these views or documents to reflect these changes is inefficient. To address this, a number of distinct methods are reported in the literature that address either incremental view maintenance or update. This thesis develops a consistent incremental algebraic approach to view maintenance and view update using generic operators. This approach further differs from related work in that it supports views with multiple returned nodes. Generally the data sets to be incrementally maintained are smaller for the view update case. Therefore, it was necessary to investigate the circumstances in which converting view maintenance into view update gave better performance. Finally, dynamic reasoning on updates was considered to determine whether it improved the performance of the proposed view maintenance and view update methods. The system was implemented using features of XML stores and XML query evaluation engines including structural identifiers for XML and structural join algorithms. Methods for incrementally handling the view maintenance and view update problem are presented and the benefits of these methods over existing algorithms are established by means of experiments. These experiments also depict the benefit of translating view maintenance updates into view updates, where applicable, and the benefits of dynamic reasoning. The main contribution of this thesis is the development of similar incremental algebraic methods which provide a consistent solution to the view maintenance and view update problems. The originality of these methods is their ability to handle statement-level updates using generic operators and views returning data from multiple nodes.

Contents

List of Figures

List of Tables

1 Introduction

EXtensible Markup Language (XML) data management has developed into an important research area [2] [3] [4] [5] [6] [7] as many systems now support XML along with the standards for querying it, XPath [8] and XQuery [9]. As the amount and size of XML data increased, the need for improving query evaluation arose as it did in structured data management. Auxiliary data structures, such as materialised views, which store the result of frequently asked queries, provide a basis for this improvement. The problem with these data structures is that they must be kept consistent with changes made to the database. For large data sets it is detrimental to the performance improvements a structure provides to fully recompute it. Therefore, incremental maintenance methods are of interest. The challenge of providing these is known as the view maintenance problem. Conversely, directly updating views is also a problem as it requires the translation of the update to the source document. This is known as the view update problem.

The following example puts these problems in context. For an XML database containing Systems Biology Markup Language (SBML) (http://sbml.org/) models (XML based markup language for representing biological processes), materialised views are defined in the database to improve query performance time. Users have full access to models for which they are the author. However, they have restricted access to other users' models. This is due to the models containing private details, i.e., contact details or unfinished model parts. Should a user wish to update a model for which they are the author then this will be an instance of the view maintenance problem. However, should they update a model, i.e., to make a correction, for a model for which they are not the author, then this will be an instance of the view update problem. The thesis to be explored in this dissertation is that incremental methods can be used for efficiently maintaining views and conversely, similar methods can be used to solve the view update problem. The ideas are conceptualised using consistent algebraic approaches.

1.1 Motivation

XML data management is now well established. The W3C's XPath [9] and XQuery [8] standards for querying XML documents are now supported by many commercial and open-source systems. Performance challenges have been raised by the complexity of XPath and XQuery and of the XML data itself. One performance improvement is materialised views (or caches), which store precomputed query results, generally of frequently asked queries. These can then improve query efficiency by being used to rewrite queries [10] [11] [12] [13] [14]. Query evaluation performance has been improved by up to several orders of magnitude using such techniques.

More recently, XQuery Update [15] has been proposed by the W3C, an update extension to the XQuery language. Data management platforms are gradually providing support for XQuery Update. When using materialised views, updates to the database could potentially require updates to the view(s).

Work in this area has led to the inverse problem known as the view update problem. The view update problem involves updates on the views themselves which then requires the translation of updates to the source document(s).

1.2 XML

XML is a metalanguage used to define markup languages. It was developed to allow a general representation to store and transport data. XML can be seen as a tree structure. A tree is a hierarchical data structure consisting of nodes. A node contains data along with relationships to other nodes, i.e., in the context of XML, an XML element, attribute or some text, along with references to its parent or child nodes, if they exist. Each node can have any number of children but only one parent. The only exception to this is the root node which has no parent. Fragments of the tree are referred to as subtrees. A subtree consists of any node in a tree, with the exception of the root node, and all its descendants. This thesis treats XML documents as ordered labelled trees. An ordered labelled tree is a rooted tree that assigns a label to each node and defines an ordering for each node's children.

1.3 XML Databases

An XML database is a database that stores XML documents. Two types of XML databases exist, XML-enabled databases and native XML databases. XML-enabled databases are relational or object-oriented databases that have been extended to store XML data. These databases require translation methods between the XML documents and the structure used by the underlying database, i.e., rows and columns for the relational model. Native XML databases store and index native XML documents without any modifications. XML-enabled databases may only store part of some or all documents, whereas native XML databases store the entire documents.

XML databases encounter similar problems to relational databases, i.e., how to query the data. As a result of XML databases being relatively new, in comparison to relational databases, a lot of the problems that have been successfully solved for relational databases are still challenges for XML databases. The problems this thesis is concerned with have been solved for the relational scenario [16] but remain unsolved in the context of XML data storage.

1.3.1 Views

A relational database view is a virtual table that maps to the result of a query. Views are generally used as caches that store the results of frequently asked queries. For example, they could simplify queries by factoring out a common subexpression. However, views also serve many other purposes:

- to implement access control for the purpose of security
- where there is a large amount of data, users may only want to deal with a small part of it
- where structure is decided by a database administrator, rather than the user, the latter may want to restructure it to better suit their needs
- hiding the conceptual complexities of the entire database from the user
- enhancing logical data independence
- allowing a user to hide data which is of no interest to them, thus simplifying the user interface

A materialised view is the same as a view, with the only difference being that it is stored in a real table.

1.4 View Maintenance

The use of materialised views introduces the view maintenance problem. When an update occurs on the database it may affect the contents of the view. This introduces two problems. First, determining if the result of a view should change due to the update. Second, if the view is affected, how to efficiently update the view to reflect the update. This can be handled either incrementally or by recalculating the entire materialised view. The second problem is the view maintenance problem which is explored in this thesis.

1.5 View Update

The inverse problem, which this thesis also addresses, is the view update problem. Database users may only have access to the materialised views. If updates are allowed on the views then it will be necessary to propagate these to the underlying document(s), therefore, the view update problem will be encountered. This differs from view maintenance in that all view updates will affect a document(s). The problem lies in how to update the view, in cases of missing information, and how to efficiently update the document(s) affected.

1.6 Hypothesis and Research Questions

The hypothesis that is proposed in this work is that algebraic incremental methods can be used to provide a single consistent solution for the problems of view maintenance and view update.

Alternative strategies are possible for handling practical view maintenance and view update problems. Updates[1] can be handled at the node-level, i.e., inserting a node at a time. However, despite this being conceptually simple, updates in real scenarios often involve multiple nodes. These statement-level updates allow the insertion/deletion of subtrees. Algorithms that only support node-level updates require multiple runs of the algorithm to support a subtree. Previous work presents an algebraic approach to this problem [6]. However, the approach described by these authors is not a general solution but is tied to the internals of the system in which it is implemented. In other works [17] [18] [19], multiple returned nodes could not be handled within views. However, views with multiple return nodes could result in efficient multiple-view rewritings [20]. Therefore, an algebraic solution, for views supporting multiple returned nodes, using generic operators is of interest.

Typically, views are smaller in size than documents. Therefore, where possible, it is expected that translating view maintenance updates into view updates (see Section 6.3) will give better performance times.

Finally, performing dynamic reasoning on XML updates, by implementing pruning rules

[1]Refers to both view maintenance and view update, unless otherwise stated.

to minimise the number of updates to be performed, it is expected that the performance of the methods can be further improved.

The aim of this research is to explore algebraic incremental methods to solve the view maintenance and view update problems using statement-level updates and dynamic reasoning by means of pruning rules. The research questions that this thesis aims to answer are:

- **RQ1**: Can a similar incremental algebraic approach to view maintenance and view update be developed with generic operators?
- **RQ2**: Does converting view maintenance into view update, where possible, give better performance times?
- **RQ3**: Does dynamic reasoning improve the performance time for view maintenance and view update[2]?

The main contribution of this work is to provide consistent solutions to the view maintenance and view update problems using mutually similar incremental algebraic methods. The originality of these methods is their ability to handle *statement-level updates* using generic operators and views returning data from multiple nodes.

1.7 Thesis Structure

The rest of this thesis is structured as follows: Chapter 2 provides an introduction to the area and discusses related work. Background technologies and implementation details required to understand the work in this thesis are presented in Chapter 3. The view maintenance and view update methods are described in Chapter 4. Chapter 5 presents the experimental evaluation results, which are then discussed in Chapter 6. Finally, Chapter 7 concludes.

[2]Each target node has to be handled separately to perform dynamic reasoning

2

Literature Review

This chapter introduces XML and as a result of its extensive use, the need for XML query languages. With the introduction of XQuery and XPath (explained in Section 3.2) the non-incremental evaluation of queries written in these languages is discussed. For frequently queried XML documents, indexing is explored. For frequently asked queries the idea of incremental evaluation is considered. Incremental evaluation has been used in other areas, such as parsing and XML schema validation. Incremental XPath evaluation can also be viewed as a generalisation of XPath evaluation on XML streams. This leads on to the concept of views and materialised views. These concepts have been used to improve query efficiency, based on query rewritings, by acting as a cache for frequently asked queries. The problem of how to effectively maintain materialised views within a dynamic database is then introduced. This problem has been extensively explored in the relational setting and is discussed along with XML publishing and commercial DBMS support. This leads on to a review of current view maintenance research for XML databases. The problem of materialised view update is then explored for both the relational and XML settings. Finally, the capabilities of commercial DBMSs with respect to view update is discussed.

2.1 XML

The EXtensible Markup Language (XML) [21] was developed in the late 90s to solve "the problem of universal data interchange between dissimilar systems" (Charles F. GoldFarb) [22]. It was developed to allow structured documents to be used over the Internet. Its main purpose with respect to databases was to allow a general representation to store and transport data. A general representation was required so that the structure was not lost when the data was exported/passed between databases. XML provides a specification for defining markup languages, based on Standard Generalised Markup Language (SGML). Since its introduction, many languages have been developed which are based on it. Some common examples are EXtensible HyperText Markup Language (XHTML) and Really Simple Syndication (RSS). XHTML is an XML redesign of HyperText Markup Language (HTML), which is used for describing web pages. This language can be used to force the documents to be well formed and marked up correctly, whereas RSS is used to display, in a standardised format, data from frequently updated data sources, for example, headlines from a news website.

As the amount of XML data increased, the need for XML based databases emerged. There are two approaches for XML based databases: XML-enabled and Native XML (NXD). XML-enabled databases map XML to a traditional database with all input and output being expressed in XML. For example, MS SQL Server [23]. Alternatively, NXD uses XML documents as the fundamental unit of storage. For example, EMC xDB [24] and Mark Logic Server [25].

2.2 XPath and XQuery Evaluation (non-incremental)

With the increasing amount of XML data and the emergence of XML databases, a need for the ability to query this data became apparent. XPath [9] expresses the query as a path and allows the selection of nodes or attributes. XQuery [8] is a superset of XPath to support powerful querying similar to SQL with the biggest difference being the ability to handle FLWOR - For, Let, Where, Order by, Return - expressions. FLWOR expressions can be seen as similar to SQL Select-From-Where expressions for joining multiple tables. The main difference being that XML data is hierarchical, not tabular. XPath query evaluation is a fundamental algorithmic problem regarding XPath. This has been a widely researched topic [26]. The two main approaches to solving this problem are to make use of dynamic programming or finite state automata [27].

Gottlob et al. [28] were the first to handle full XPath 1.0 or even relatively large fragments of it. This method uses dynamic programming due to XPath evaluation consisting of overlapping subproblems. By splitting evaluation into smaller subproblems this overlap is exploited by ensuring each subproblem is computed only once. The algorithm presented for full XPath runs in low-degree polynomial time, in the worst case, with respect to the size of the query and the data. Additional algorithms were defined for handling the fragment of XPath that the authors defined as Core XPath, i.e., simple manipulation of sets of nodes (does not consider string or arithmetic operations), and XPatterns, which is an extended version of EXtensible Stylesheet Language Transformations (XSLT) that handles all XPath axes. An axis defines the tree relationship between nodes, e.g., the child axis. These algorithms run in linear time with respect to the size of the query and the data.

A linear time algorithm, with respect to document size, was presented by Bojanczyk et al. [29], which can handle XPath with tests on attribute values using equality. This work is a generalisation of the automata-theoretic framework[1] for node tests using attribute values but only supports equality. This was an improvement on the linear-time algorithm for Core XPath in [28] as the latter could not handle attribute values. However, the linear time data complexity comes at a cost of a multiplicative constant that is exponential in the query. The complexity is $O(2^{|\varphi|}.|t|)$, where φ is a node selecting XPath query and t is an XML document on which the query is evaluated. Therefore, the complexity is linear when φ is fixed. This algorithm was improved in [30] to be polynomial in the size of the query.

[1]using finite-state tree automata to represent queries

However, the updated algorithm cannot handle the Kleene star[2] in path expressions. This was an addition the original offered which is an extension of XPath. A third algorithm was introduced in [2] with $O(|t|log|t|)$ time complexity for a document size $|t|$. This maintained the polynomial combined complexity as well as being able to handle the fragment of XPath in the original algorithm. All the algorithms can handle boolean, unary and binary queries. The algorithms work for the restricted set of XPath 1.0 used in the paper. However, outside of this they are likely to fail.

The tree homeomorphism problem (determining if a match exists between a tree pattern/query and a data tree) was handled by Götz et al. [31]. The presented algorithm solves this problem as well as being general enough to handle the tree homeomorphism matching problem (finding the matches from the tree homeomorphism problem) while retaining the same complexity. At the time of development it was the only algorithm that could guarantee a space bound that was not reliant on the size of the data tree, the only affecting factors were the depth and branching factor. Tree pattern matching was shown to be LOGSPACE-complete when only descendant axes were used.

Evaluating XQuery would require many nested iterations if a naive implementation of the semantics was used, due to its structure (see Example 2.2.1). An implementation like this would be inefficient for complex nested queries with multiway joins [32]. Therefore, generally queries are compiled into algebraic plans which are similar to those used in the relational setting. As a result of this, standard optimisations such as unnestings can be used. These optimisations can increase query efficiency by up to several orders of magnitude [33].

Example 2.2.1.
for $a in doc("auction.xml")/site/people/person/@id
    $b in doc("auction.xml")/site/open_auctions/open_auction/bidder/personref/@person
where $a=$b
return ⟨personID⟩ $a ⟨/personID⟩

This query returns the IDs of all people who have a bid on an open auction. Several

[2]In the context of path expressions meaning "zero or more".

iterations are required to evaluate this result. First the document must be searched for a and b. Next they must be compared to identify those that are equal. Finally, a new element is created and returned for each match.

2.3 Indexing XML Documents

A related research area is the indexing of XML for frequently queried documents. Without an index these queries could be inefficient due to the requirement for frequent scans of a document to locate the required node(s). The key to fast query processing is an ability to directly identify nodes. The indexing of XML documents typically relies on labelling schemes. Documents can be defined as static, i.e, unchanging over time, or dynamic, i.e., changing with time. A problem with dynamic documents is how to correctly and efficiently update the label set as a result of situations such as the addition of a new XML subtree as a child of an existing leaf in the document. Various dynamic labelling schemes exist [3, 34, 35], however, some of them sacrifice the query performance and increase the labelling costs in order to achieve dynamic labelling. This is a reasonable cost to bear in databases of frequently changing documents. However, update patterns can vary. In some cases only a subset of documents change frequently, with this subset changing over time.

The Dewey method of node labelling [36] works well in the static case and is known to be compact and efficient. Dewey labels store the path from the document root to the current node, as well as support parent-child, ancestor-descendant, document order, and sibling relationships. It has been widely used in XML query processing [37] [38] [36]. It is particularly helpful in XML keyword query processing [39] [40] which requires calculating the Lowest Common Ancestor (LCA) for a set of nodes. Each Dewey ID stores path information making it suitable for this scenario. Examples showing the Dewey method can be seen in Section 3.1.1. An alternative method of labelling is the containment labelling scheme [41] [42][3]. Despite being popular it does not have the same capabilities of Dewey IDs, i.e., checking sibling relationships. An example of the containment labelling scheme is

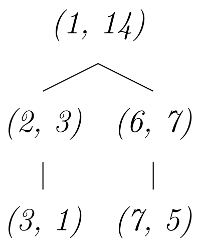
[3]These two works are similar but developed independently of one another. Therefore, they are both credited for this work.

shown in example 2.3.1.

Example 2.3.1. [4] *The containment labelling scheme makes use of an extension of preorder traversal that includes a range of descendants, i.e., all nodes are labelled with an order and a size - <order, size>. The rules for this scheme are as follows:*

- $order(x) < order(y)$ *where x is a parent of y.*
- $order(x) + size(x) \geq order(y) + size(y)$ *where x is a parent of y.*
- $order(x) + size(x) < order(y)$ *where x is the predecessor to its sibling y*[5].

An example of a document labelled using the containment labelling scheme can be seen below:



With this scheme it is not possible to determine sibling relationships, i.e., according to the second rule $(2, 3)$ *could be a sibling of* $(7, 5)$.

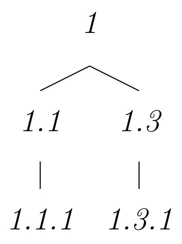
The Dewey labelling scheme was developed for static XML documents. It does not offer the same effectiveness for dynamic documents, as a result of the potentially high cost of relabelling after a document has been updated. Dynamic labelling schemes have been developed based on Dewey, such as ORDPATHs [34]. However, this scheme only utilises odd numbers at initial labelling. The reason for this is that insertions between nodes are handled using a 'careting in' method which inserts an even component that lies between the final components of the left and right siblings, followed by a new odd component. This new odd component is to keep the labelling scheme consistent, as well as to be able to handle the insertion of whole subtrees using only one even component. This works as the method ignores even numbers with respect to increasing the level of a node. Inserting as a first or

[4]based on [42]

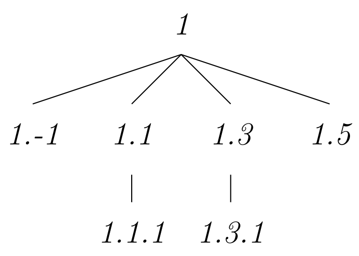
[5]In other words, x and y are on the same level but x comes before y.

last child is the simple scenario which involves subtracting two from the last component of the first child/adding two to the last component of the last child, respectively, with negative numbers being permitted. Despite supporting insertions without relabelling it comes at a cost. Ignoring even numbers results in less compact labels and the complexity of ORDPATH label processing is increased due to the 'careting in' method. XML documents labelled using ORDPATHs can be seen in Example 2.3.2. Dynamic DEwey IDs (DDEs) [3] efficiently handle the labelling problem for both static and dynamic documents. For static documents the original Dewey labelling system [36] is used. Alternatively, for dynamic documents, DDEs can handle updates without the need for relabelling. Insertions as first and last child are handled similarly to ORDPATHs, except one is added/subtracted. For insertions between nodes the labels of the two siblings are added together. The same work introduces Compact DDEs (CDDEs) which are optimised for frequent insertions. The authors have shown that both schemes can efficiently handle insertions repeatedly occurring at the same place (skewed insertions) which was a problem for previous labelling schemes.

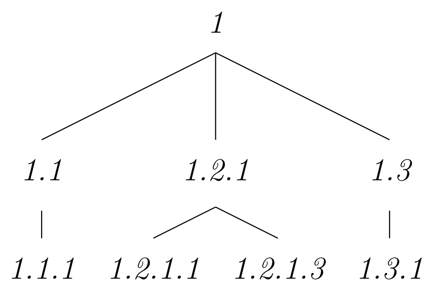
Example 2.3.2. *ORDPATHs are similar to Dewey except they only use odd numbers within the labels. This can be seen below:*



Labels for insertion of a rightmost child are simple to determine and just require adding the next available odd number on to the parent node's label. Leftmost insertions are handled using negative odd numbers. Examples of these types of insertions can be seen below.



The 'careting in' method is used to insert nodes between two siblings. An even component between the final components of the sibling labels is inserted followed by an odd number, starting with 1. An example of the 'careting in' method can be seen below.



Above a node is inserted between 1.1 and 1.3. Therefore, the available even component is taken between these values, 1.2, followed by the first available odd number, i.e., 1, to keep the labelling consistent. Other nodes are labelled as normal.

Prime labelling scheme [43] was developed to handle insertions without the need for relabelling. Each node is identified by a unique prime number and labelled using the product of this number and its parent's label. Paths can then be identified by factorising these labels. The performance impact derives from determining document order. The document order is derived using a Simultaneous Congruence (SC) value. This value is calculated by solving the set of simultaneous equations which represent the tree. This order number can then be determined using the expression: $SC\_Value \ mod \ node\_label$. However, these SC values can become large for big XML documents. A list of SC values is used to counter this where each value handles the document order of five nodes. However, this has a knock on effect on storage and maintenance for large XML documents which require large lists. Typically an average insertion or deletion requires approximately half of the values to be recalculated. This has to be performed by the computationally expensive Euler's quotient function [44].

These encoding schemes are an alternative approach for handling dynamic XML documents [44] [45] [46]. They allow the handling of dynamic updates without relabelling by transforming the labels into another format. Encoding approaches have shown better performance for frequently updated XML documents [47]. Several problems do however exist for the approach. Additional labelling cost is incurred as a result of translating labels into dynamic formats. The performance is also affected by the frequency of each document's updates. Generally each database will have a combination of documents with varying levels of update frequency (with some possibly being static) which can potentially change over time. Static labelling schemes are more efficient for static documents in comparison to dynamic approaches. The problem is how to decide which documents are static and which are dynamic. Simply applying a single encoding scheme for all documents would be inefficient due to the extra encoding cost, except for the situation where all the documents were consistently dynamic. The final problem is when there is a mix of label formats in the database due to different labelling schemes being used for static and dynamic documents. This makes updating and querying complex as different query and storage methods are required. The ideal labelling scheme should be able to minimise the growth rate and the update cost, regardless of the updates.

The majority of previously mentioned works do not support the reuse of deleted labels. Without the reuse of deleted labels the growth rate of the labels increases. EBSL [48] and other schemes [49] [50] [47] [51] provide the ability to reuse deleted labels.

The indexing of XML for frequently queried documents improves query efficiency. Within these documents there generally exists a set of frequently asked queries. Efficiency can be further improved by storing these queries. However, this also introduces the requirement to maintain them, ideally incrementally.

2.4 Incremental Evaluation

In many databases there is a collection of frequently asked queries. For example, an employee's monthly pay cheque may be generated by a database query. Assuming the employees pay is calculated by the number of hours they work multiplied by their hourly pay and nothing else on the cheque changes each month then it would be beneficial to only calculate the modified part of the cheque. This introduces the idea of incremental evaluation, i.e., only evaluating the parts which have changed. This idea has been explored in other areas which will be detailed in this section.

2.4.1 Incremental Parsing

Incremental parsing is required for incremental program compilation. Research has focussed on LR parsing (bottom up parsing which builds a rightmost derivation of the input) [52] [53] [54] [55] [56] and LL parsing (top down parsing which builds a leftmost derivation) [57] [58]. This is a consequence of popular grammars used to define programming languages requiring to be parsed in one of these ways, e.g., LR(0), LR(1), LL(1), LALR(1), and LL(1). The techniques operate by producing a parse tree from parsing the input text, which is updated when updating the input text. Incremental parsing tries to discover the smallest fragments of the parse tree that are affected by updates to the input text.

2.4.2 Incremental Evaluation of XML Schemas

In addition to incremental parsing, incremental evaluation is applicable in other areas, such as the validation of XML schemas. It is utilised by some XML editors, e.g., XMLMind [59] and XMLSpy [60]. Validation of static XML documents is the simplest instance of XML validation.

Barbosa et al. [61] consider the incremental validation of XML documents under XML Schema and DTD definitions. Statement-level updates are considered as well as ID and IDREF attributes. Typically XML documents are considered as trees, however, due to the inclusion of node identifiers (ID attributes) and references to identifiers (IDREF attributes) the trees become graphs. This entailed the authors splitting the method into two parts: checking the structural constraints as defined by the DTD and also checking the attribute constraints. Checking the structural constraints entails parsing, whereas checking the attribute constraints requires checking the ID attributes are unique, as specified in the DTD, within the document and also that the IDREF references actually point to something. Both these stages can be evaluated in $O(nlogn)$ time and linear space. A large fragment of XMLSchema [62] can also be evaluated using this algorithm.

Balmin et al. [4] presents algorithms closely related to [61] which show the problem can be solved in logarithmic time for fixed DTDs and an abstraction of XML schemas called specialised DTDs[6] [63] can be solved in $O(mlog^2n)$. A notable difference is that [61] assumes database access is $O(logn)$ as opposed to constant time assumed in [4]. This allows consideration of the complexity of the algorithms with respect to the number of database accesses rather than time. In contrast [61] handles restricted DTDS and XML schema specifications in constant time and arbitrary DTDs in linear worst case time. It also handles statement-level updates whereas [4] can only handle node-level, but both handle similar update operations (with [4] not having the capability of allowing attribute constraints). However, [4] can perform renaming operations. In [61] this is only possible by expressing them as deletions followed by insertions.

To solve the problem of checking XML validation with respect to a DTD, a place to begin is how to verify that a regular expression is satisfied by a word. The work in [4] is based upon the algorithm for the incremental validation of strings. Algorithms, which support multiple update transactions, are presented to handle the incremental validation of DTDs, XML Schemas and specialised DTDs. Specialised DTDs are more complex as an update to a single node can result in global effects for the typing of the tree. Conversely, for DTDs an updated node only needs to be validated locally as it is only dependent on its parent and children, and XML Schemas can only affect descendant types. The algorithm for XML schema can not handle the validation incrementally for the renaming of internal nodes. However, this can be handled by the specialised DTDs algorithm. A constant time incremental validation algorithm for 'local' DTDs is also presented. These are DTDs that use regular expressions. This allows membership to be checked locally after an update. This is performed by checking the substrings within a bounded distance from the update position. By disallowing the renaming of internal nodes the need for an auxiliary structure is avoided and incremental validation can be achieved by maintaining a list of counters. The results show that DTDs are the simplest to handle followed by XML schemas and finally specialised DTDs.

[6]Element type can depend on all the node labels of the XML tree.

2.4.3 XPath Evaluation on XML Streams

XPath evaluation on XML streams can be perceived as a generalisation of incremental XPath evaluation. An XML stream is a flow of XML elements. Such evaluation algorithms are a popular choice where XML documents are transferred between systems. They can also perform efficiently over stored XML data as a result of maintaining a predictable access pattern.

There has been a lot of work performed in the area of XPath evaluation over XML streams [64] [65] [66] [67] [68] [69] [5] [70] [71]. These operate by performing a one-pass sequential scan of the XML document and keeping small important fragments in memory for later use. However, despite improvements in space and time complexity and on the fragment of XPath handled, these works required a prohibitively large amount of memory for some query types. These methods make use of finite-state automata. In this context, the majority of memory used is for the storing of large transition tables (definitions of the next state based on the current state and its inputs) and for buffering document fragments. This is due to the limitation of the data stream model. Bar-Yossef et al. [72] presented a theoretical study of memory requirement lower bounds (instance data complexity) and explored whether previous memory limitations were unavoidable or just a result of the proposed algorithms. Their algorithm from [5], which avoids the use of finite-state automata, is modified to use a more sophisticated method for handling the global data structures. This method reduces the space complexity to close to the optimum, as defined by this paper. By avoiding the use of transducers[7] or automata, the requirement for storing large transition tables is avoided. Instead an alternative approach is used which is based on finite-state automata theory but overcomes its problems.

A side-effect of using finite-state automata is the exponential complexity caused by deterministic automata being used to represent non-deterministic automata. This is because for the worst case, the number of states is exponential in the query size [72]. The cited paper shows this is avoidable for a large fragment of XPath defined as "Redundancy-free XPath", which disallows queries with redundant data where inclusion or removal does not affect the

[7]similar to an automaton except it has an input and output tape as opposed to just a single tape

meaning of the query.

Related work [73] has studied the memory lower bounds for evaluating XPath queries over streams of indexed XML data. Algorithms are available that can perform multiple sequential scans of the XML documents [74] as opposed to a single scan as used in previously mentioned works. This approach shows that there are tradeoffs between the number of scans needed and the space for some XPath queries.

2.5 Views

Query evaluation efficiency can be improved by making use of materialised views. These views act as a cache and can store the query results of frequently asked queries. Efficiency is then improved by query rewriting (where applicable) using these views, which is faster than using the base relations or documents only.

2.5.1 View Maintenance

Materialised views introduced the problem of how to effectively maintain them within a dynamic database. The naive approach of full recomputation is undesirable as the benefit the views provided for query efficiency would be affected significantly if they had to be recomputed everytime the database was modified. Therefore, incremental methods are of considerable interest.

Relational and Object-Oriented Incremental View Maintenance

Incremental view maintenance is a well established research topic within the relational and object-relational settings. Such work [75] has included determining whether views expressed in a language L1 can be maintained by an algorithm represented by a query in language L2. The goal of this research was to discover a language L2 such that it was well supported by the database system and could be quickly evaluated.

Early work in this area focussed on detecting updates that would not affect the view and developing algorithms for propagating "deltas" to the view based on the source updates. A "delta" can be described as a simple transaction which contains insertions and/or deletions of tuples. Most of these algorithms treated the view definition as a mathematical formula and determined an expression to represent the required changes to the view.

View maintenance techniques for relational databases can be split into two categories, those that use full information and those that use partial information. Full information refers to methods where the materialised view and all the base relations are available. Partial information defines the case where the materialised view and only a subset of the relations involved in the view are available.

In the context of full information methods, algebraic differencing is an approach that has been explored as a basis for view maintenance [76] [77]. This involves determining the difference between two algebraic expressions, one representing the original view and another the updated view. From this a relational expression can be defined which expresses the required change to a Select Project Join (SPJ) view while avoiding redundant computation. A correction to the minimality result (the minimal relational expressions that need to be re-computed) [77] is presented in [78], which also extends the method to multiset difference and supports multiset algebra with aggregations. For each view, two expressions are computed, one to handle insertions and one to handle deletions.

Another approach taken is that of counting algorithms. The general idea behind these is that for each tuple in the view the number of its derivations is stored as additional information. BGDEN [79] handles recursive views and makes use of counts for a subset of derivations to give even finite counts to every tuple, even if tuple derivations are infinite. The counting algorithm [80] handles outer join views and views defined with union, negation and aggregation that can have duplicates. Each view tuple stores its number of derivations (derivation count). This count can be calculated with negligible effect on the view evaluation time for SQL views. When the base relations are modified a set of changes for the view relations is produced. This represents insertions as positive counts to be added and deletions as negative counts to be subtracted. If a tuple's count reaches 0 it is removed. It is optimal as it only computes the view tuples that are inserted or deleted. This algorithm can only handle non-recursive views unless the number of derivations for each tuple is finite [81] [82]. However, despite this condition, the cost of computation can be significantly increased by the computation of counts.

Datalog [83] views are used in all view maintenance work that can support recursive views. Datalog is a declarative logic programming language that is more expressive than SQL as it allows recursion. The paper [80] also presents the Delete and Rederive (DRed) algorithm which can also support SQL views. These views can support recursion, union, and stratified negation and aggregation, i.e., negation and aggregation outside recursion. However, they are restricted in their inability to handle duplicates. The algorithm operates by overestimating the derived tuples to be deleted. This is an overestimate because a tuple is included if a base relation update affects any of its derivations. Then tuples with other derivations in the new database are removed from the overestimate. The partially updated view is used along with the base relation insertions to find the new tuples to be inserted. Despite some other methods being more efficient, at the time of development, none of them supported as large a class of views. The DRed algorithm is extended in [84] to handle non-traditional views, in this case views that can contain non-ground tuples, i.e., tuples that can contain variables.

Propagation/Filtration (PF) [85] algorithms provide an approach that is similar to DRed except that base relation changes are performed on a relation by relation basis using a similar algorithm to DRed, i.e., one base relation change results in computing one derived relation[8] change, performed in a loop to handle all the relations. However, the overestimate and pruning stages are performed after each iteration. This avoids propagating tuples in earlier overestimates that do not change. However, it also results in some tuples being rederived multiple times and fragmenting computation. Rederivations are reduced in [80] using memoing (saving intermediate results) and by exploiting the stratification to reduce rederivations. Between DRed and PF there is no single best solution to relational view maintenance. They outperform each other for specific view definitions. DRed is always better for non-recursive views [16].

Other counting algorithms exist for non-recursive views. [86] is similar to the counting algorithm [80] but only maintains SPJ views and handles insertions and deletions separately. Derivation counts are used for select, project and equijoin views in [87] by means of a data

[8]Materialised views are derived relations which consist of other derived relations and stored relations.

structure with pointers from a tuple to other tuples derived from it. Finally, [88] makes use of materialised views which support selections and one join which they name “View Caches”. These views only store tuple identifiers which have to be joined to get the view tuples.

Methods using production rules have also been developed. The Ceri-Widom algorithm [89] maintains non-recursive SQL views without duplicates, aggregation and negation and can not handle view attributes that identify base relation keys through functional dependencies. SQL queries to perform view maintenance are calculated by the algorithm and called from within production rules.

An alternative approach is taken by the Kuchenhoff algorithm [90] which can support recursive views. A set of rules are presented to be applied to consecutive database states to determine the difference for a stratified recursive program - only contains operations within recursion that can be sensibly defined, i.e., not negation or aggregation. Only stratified recursion is considered as it is safe. This is the same restriction the SQL-99 [91] standard makes. These rules are similar to those of [80] with the exception of the use of the delete/prune/insert technique (also used in [85]). In comparison with the DRed algorithm the duplicate derivations, guaranteed not to affect the view, are discarded for positive rules later on in the method compared to DRed.

The Urpi-Olive algorithm [92], which can support recursive views, makes use of a set of transition rules using existentially quantified subexpressions (subexpressions that are known to exist). The rules define how a modification to a relation translates into a modification for a derived relation. They are defined on stratified Datalog views and expressed in Datalog rules. Pruning can be performed as the quantified subexpressions can be removed in certain circumstances as they may go through negation. However, the update model is more useful for non-recursive views because keys are derived as updates are modelled directly.

Other methods, for handling non-recursive and recursive views, make use of transitive closures - a graph’s reachability matrix between all its nodes, i.e., what nodes have a path between them. [93] handles recursive views by means of defined non-recursive programs to perform the update after insertions into the base relation. [94] handles the transitive closure of certain graph types using a non-recursive program to perform the update after insertions and deletions.

Work has also been performed on the view maintenance of non-recursive full outer join views [95]. From the view definition, a right and left outer join with the updates, is defined to perform the view maintenance for insertions and deletions. These, however, can not compute all view changes, like SPJ views, as insertion side-effects of a successful join can not be handled.

Another area of view maintenance within relational databases is methods which use partial information. In this scenario the view may not always be able to be maintained due to lack of information. For example, only the materialised view may be available. In this case updates would only be able to reason based on the data in the view. Methods within this area first check whether a view can be maintained before performing view maintenance. An initial topic of work in this area was for the "irrelevant update" or "query independent of update" problem. This entailed detecting if an update would affect a view. If the view is affected an additional algorithm is required to perform the view maintenance, if it is possible. [86] [96] present a method for the former problem for SPJ views, whereas [97] handles Datalog views. [98] extends on methods for Datalog views by handling arithmetic inequalities and negated base relations.

If a materialised view can be maintained with only the view and key constraints, it is called self-maintainable [95]. [95] presents results for insertions, deletions and modifications for the self-maintainability of SPJ and outer-join views. This work shows that for insertions, most SPJ views can not be self maintained. However, they are generally self-maintainable for deletions and modifications. In [99] a self-maintenance algorithm for SPJ views is described. However, this is database instance specific and can only handle insertions and deletions. It was subsequently corrected and extended [100]. [96] presented autonomously computable views which can be maintained for a set modification for all database instances using only the materialised view. These SPJ views do not contain self-joins or outer-joins and can handle insertions, deletions and modifications.

The partial-reference maintenance problem occurs when access is only available to the materialised view and a subset of the base relations. Chronicle views [101] are views over a relation that permits insertions and consists of an ordered sequence of tuples. A method is presented to handle the chronicle view maintenance problem. This involves maintaining the view when insertions are performed on the chronicle. The complete chronicle may be inaccessible because it is so large that only a fragment may be stored in the database.

In [102] [100] partial-reference maintenance algorithms for specific database and modification instances are presented. The given materialised view and/or base relations are checked against conditions to determine if the data is capable of maintaining the view.

Publishing Relational Data to XML

Using a relational DBMS to publish relational data to XML documents is known as XML publishing. This is an important problem as many applications use XML as an exchange format where their data is stored in relational databases. Therefore, XML can be used for a materialised view of non-XML data. XML publishing can be achieved by the use of middleware [103] [104] or provided directly by DBMS support [105]. Scenarios where this could occur include mediation, archiving and website management.

XPERANTO [104] is a middleware system which produces XML documents from object-relational databases. It can support both object-relational and flat-relational structures. The XML can be queried and (re)structured with the XML query language XML-QL [106]. Much like other publishing systems XPERANTO translates the XML queries into SQL, converts the results into XML and returns the XML documents. This differs from other similar systems such as SilkRoute [107] and [108] as it uses "pure XML", meaning users/developers need no knowledge of SQL. The middleware provides efficient query evaluation by passing as much work as possible to the DBMS [108]. Finally, a notable property is that it allows seamless querying over relational data and metadata.

Fernandez et al. [103] present a different method of connecting relational data to XML views. XML views are defined in the declarative query language, RXL, of the SilkRoute [107] relational to middleware system. The middleware handles a view query by translating it to several smaller SQL queries on the relational database. It then integrates the results and adds the appropriate XML tags. This work focuses on the materialisation of large RXL views - to support warehousing or data export applications - and the best way to select the SQL queries when there is no control over the DBMS. This lack of control is due to the algorithm operating in middleware, therefore, DBMS-specific heuristics cannot be utilised.

An algorithm is presented which determines an optimal set of SQL queries from an RXL view. This applies greedy optimisation to the view generation process. This greedy algorithm can produce SQL queries which are near optimal by communicating with the target query optimiser to get query cost estimates. This method can also work with other relational-to-XML systems, i.e., Oracle XML SQL Utility, IBM DB2 XML Extender, Microsoft SQL Server 2000, as the view tree representation can represent their view definition languages.

The view maintenance problem of published XML documents with focus on XML produced by schema-directed XML publishing middleware is explored in [109]. Schema-directed publishing is XML publishing where the output XML view must adhere to a schema. In schema-directed publishing, a mapping between the XML and the database may not be definable by a simple query. Attribute Translation Grammars (ATGs) [110] can be used for schema-directed XML publishing. An ATG consists of a DTD and a set of semantic rules used to produce an XML document. These semantic rules are represented as SQL queries and define what is required from the database to generate a specific view. Two approaches for incremental evaluation of ATGs are explored in this work. These are the reduction approach and the bud-cut approach. The reduction approach passes as much work as possible to the DBMS. It relies on translating ATGs to SQL 99 queries and a relational encoding of XML trees. A disadvantage of this approach is that high-end DBMS features (such as incremental SQL 99 view maintenance) are required which at the time of development were not available within commercial databases. This differs from other XML publishing work [103] [104] which aims to pass to the DBMS the XML publishing work as opposed to incremental work, i.e., view update. The XML publishing work is handled within the middleware. The bud-cut approach uses the DBMS for simple queries (similar to distributed relational database query processing [111]) and uses the middleware to perform the majority of the work. It improves on previous work by using a caching strategy to minimise unnecessary computations, i.e., it only computes each new subtree in the view once regardless of how many times it appears in the view. The old XML subtrees are also maximally reused which works best when only a fragment of the XML is contained in the view. Effective optimisation techniques not supported by the DBMS are used, e.g., query merging [103]. Partial results can be returned during computation and the remaining computation can even be performed lazily [105]. Finally, for the bud-cut approach, the DBMS is not required to materialise the view or support incremental view updates.

Experiments show that this approach can effectively maintain XML views. XML integration [112] - generalisation of ATGs - could be handled by an extension to the bud-cut approach to handle multiple data sources. XML views from other systems, e.g., [103] [104] can also be handled.

XML Publishing and View Maintenance Techniques in Commercial DBMSs

Some query standards and languages, e.g, SQL [91], and commercial DBMSs, e.g., IBM DB2 [113], allow users to specify XML views of relations (XML publishing). However, XML view maintenance is very restricted or not yet supported.

SQL [91] provides functionality for XML views. However, it does not allow XPath queries on the views to be recursive or any update operations on the views. XML views are represented by using SQL to encapsulate the relational tables. IBM DB2 [113] supports view maintenance, i.e., it can propagate updates on the relations to simple XML views.

2.5.2 View Maintenance for XML Databases

Following the success of materialised views within relational databases they were also introduced within XML databases. Due to the different structure of these databases the previously developed methods for view maintenance were not suitable. This was due to XML data being hierarchical in contrast to relational data which is flat. Therefore, this reintroduced the problem of view maintenance.

A method to handle both hierarchical semi-structured and relational databases with view constraints is introduced in [114]. This work utilises an unordered data model and supports a restricted query language, which maintains the distributivity of queries over updates, in order to simplify update translation but at the expense of the expressiveness. One such limitation is that queries must be monotonic with respect to updates.

Algebraic methods have also been developed for XML view maintenance [115]. Updates are translated into operators in a tree algebra, XAT, making use of auxiliary data as required. An initial algorithm, for view maintenance, did not handle ordering and stored all intermediate data in a cache. Ordering was included in later work [116] by means of node identity being encoded by a labelling scheme.

Another algebraic method to maintain non-recursive XQuery views is presented in [6]. The main part of this work is an update translator for translating source updates into view updates. This operates by making use of the source update, query, and annotation hints[9] generated from the source. Other parts are an update compiler and a query instrumentor. The update compiler translates update expressions to Galax[10] query plans. The query instrumentor rewrites a source query to compute the required annotation hints for update translation. This method makes use of statement-level updates. An interesting feature that was added is the ability to control the quantity of annotation hints as an external system parameter. These annotation hints are intermediate data in the translation process, as some operators may compute and discard this data which is required in later stages. The greater the amount of annotation hints, the greater the possibility of incremental view maintenance. However, this has the disadvantage of a growing annotation file. If less annotation hints are available, most cases result in recomputation but annotation files are small. This parameter gives programmers the ability to balance these tradeoffs as in some cases keeping no annotations for specific operators might be a reasonable approach for certain queries. The translation of XQuery into algebraic plans is described in [117] [118].

The update method makes use of the tree algebra of Galax despite the view syntax being XQuery. This is due to XQuery being complex and monolithic as a result of being stored in monolithic FLWOR blocks and supporting complex features such as conditionals, iteration, navigation, variable binding, selection, grouping, and reordering. Conversely, Galax is simple as it breaks down these complicated features into a more primitive representation; it is

[9]These hints vary depending on the operator which they are being stored for. For example, for the conditional operator it could store its encoded boolean value.

[10]open-source XQuery implementation

orthogonal, as a consequence allowing easy identification of the operators that are easy to maintain and those that are not; and composable, which simplifies the process of extending the system to handle additional built in functions and algebraic operators. Compositionality also allows simple proofs of correctness. The benefits of working with a primitive representation are: the update translation algorithm is more efficient (i.e., it can be performed as a recursive function on algebraic queries) and since it includes relational algebra operators it is easy to see the relationship to previous work in the relational setting. These benefits have been noticed by other view maintenance works [78] [16]. Despite the language being simple it can still represent the effect of an update on any data model value. These simple updates are only manipulated internally but it can handle source updates in any formalism, e.g., XQuery! [119] (an extension to XQuery that handles update operations with side-effects) or the W3C's XQuery Update Facility [15]. More expressive update languages could be used. However, this would require modifying the system. The update would have to be performed on the source to find fixed path "atomic" updates that would have to be encoded in the described update language.

This Galax method is similar to the Rainbow method presented in [115] [116]. However, the Rainbow method's labelling scheme simplifies some operator's translation rules due to the ability to discover the value an update has affected. Additional annotations are required for order changing operators. Therefore, a lot of the same positional data that is contained within the Galax method's annotation scheme is recorded in the Rainbow method's labelling scheme. The Galax method uses a functional data model without the requirement for additional metadata. Conversely, the Rainbow method uses node IDs stored in the source data. Despite working well, IDs may not always be available. For example, if the source and view were on different hosts, the IDs would need to be sent across the network. The Galax method can tune and selectively omit annotations, whereas the Rainbow method can not do this without affecting the semantics of the method. Finally, the Galax method carries algebraic plans for replacements and insertions as opposed to data model values in the Rainbow method. Therefore, when propagating an update, a source query is required after each replacement to keep it correct. The Galax method only requires the algebraic plan to be rewritten.

Other non-algebraic methods have also been developed, including an approach to solve the boolean version of XPath incremental view maintenance [17]. This involves determining whether an XPath expression is still satisfied following an update. The incremental updates are performed by making use of an auxiliary data structure. This data structure consists of a record for each node in the document. Each record contains the set of query nodes which match the node; the number of children that satisfy a query node; the set of query nodes that are satisfied in a descendant; and the number of children with descendants that satisfy a query node. The method can handle insertion (next sibling or first child), relabelling of single nodes and deletions of subtrees. This was the first work to consider incremental XPath evaluation worst case complexity bounds. A method to support next and following sibling axes is presented which structures the XML as a string. However, this has the side-effect of not being able to support disjunction and negation.

Maintenance of XPath queries over a richer dialect than just boolean is studied in [120]. This method handles node-level updates and uses a two stage approach. First identifying the directly affected nodes, then calculating the indirectly affected nodes. This is extended [121] to handle the scenario of when the database and the view store are decoupled and the update has to be propagated using less information. The information used is that which can be expressed using standard XPath interfaces [122], i.e., the update, view definition and view contents. Using less information, incremental maintenance is not always possible. Therefore, the work aims to reduce full recomputation by identifying cases where updates do not affect the view and where views can be maintained using only the update. The XPath dialect and node-at-a-time approach stay the same.

2.5.3 Incremental View Maintenance for ActiveXML

A related area of work is the incremental view maintenance of Active XML (AXML) [123]. Such XML documents may be populated by information from various web components. This is a common scenario in many web applications, i.e., monitoring distributed autonomous systems [124] and mashup systems [125]. This work [126] [127] looks at the complexity of determining whether there is an update sequence for an active XML document that satisfies a query. An efficient model is introduced to represent these interactions. The main part of this model is a complex stream processor, referred to as the axlog widget which uses AXML and Datalog [83]. This is a view over an AXML document which contains an AXML document which uses streams of updates that are used to communicate with the rest of the model. Input streams define document updates. Whereas output streams specify the updates required to maintain the view for a query defined on the document. The authors mostly deal with insertions but deletions are also addressed. The queries used are tree-pattern queries with value joins. The complete axlog system consists of a set of widgets distributed on multiple peers which communicate via streams of updates.

An algorithm is presented to efficiently compute the output streams for the widgets which is a view maintenance problem. The actual process of incremental maintenance can be handled directly using existing techniques for Datalog, Differential [128] and MagicSet [129] (similarly, QSQ [130]), for incremental computation and query optimisation, respectively. Other known techniques provide a benefit, including constraint databases [131], incremental Datalog evaluation [132] and XML filtering - using YFilter [67]. The algorithm uses a notion of relevance which is more suitable to the dynamic setting than MagicSet. This is based on satisfiability of a fact and provenance of data. This notion can be used to prune data at the Datalog program and at the stream source which saves on processing and communication respectively. By using this notion the algorithm is more optimistic than MagicSet. Pruning is not performed on data just because it is not currently relevant. Satisfiability determines the data which might be relevant in the future. MagicSet is only interested in data which is currently relevant.

2.5.4 View Update

The majority of database systems have the functionality to limit the fragment of the database that is visible to the user. This is useful for security reasons as well as simplifying the users viewing and interaction with the data by removing data which is uninteresting or irrelevant to them. A problem arises when users attempt to update this data as the updates have to be propagated to the source document(s). This is known as the view update problem, and represents the inverse of the view maintenance problem where the source document is updated and the updates propagated to the view.

2.5.5 View Updates for Relational Databases

The main concept behind view update is the translation of view updates to base table updates. This translation is formalised in [133]. However, translation procedures will not always produce correct translations. If a translation exists, it may not be unique. This could result in equivalent translations resulting in different modifications to the database. Therefore, conditions were also derived for which these translations will be correct. These conditions were first derived in relation to schema and view extensions. Subsequently they were derived syntactically using functional dependencies, keys and subset constraints. The authors show that only under tight conditions, unique and correct translations could be produced. This work is extended from relational view theory to a general network model [134].

An important concept in relation to view update is a view complement. A view compliment is the data required to reconstruct the base relations using itself and the view - this may include some overlap with data in the view. There can be multiple view compliments, with each one representing a different update policy. In [135] if a constant compliment exists (one that is unaffected by the view update) then a view update can be translated into base relation updates. The smaller the view complement the bigger the set of updates that are translatable [136]. However, [135] discovered that finding a minimal view compliment is NP-complete and the construction of an update translator given a complement view was a mainly undetermined problem at the time. The translation was based on the method proposed in [137]. However, it was only tested in a simple scenario. Only database schemas containing a single relation were considered where integrity constraints were mostly functional dependencies [138] [139]. The views were then simply projections of the relation.

A restricted class of SPJ views is handled in [140]. These views can only support Boyce-Codd Normal Form relations. Additionally, attribute comparisons can not be included in selection conditions; a join view must be a single tree with each node a relation; and the joining of base tables can only be performed on keys and have to satisfy foreign keys. Five criteria are presented that must be adhered to for translations to be acceptable. These criteria are:

“No database side-effects”
“Only ‘one step’ changes”
“Minimal change: no unnecessary changes”
“Minimal change: replacements cannot be simplified”
“Minimal change: no delete-insert pairs” [140]

Translators that meet these criteria for the set of SPJ views are detailed. Additionally, a collection of all templates for translations which satisfy these criteria are presented. Some updates on views containing joins can not avoid side-effects from translation and only these views are permitted to have update translators with side-effects.

View Update for XML Views

The majority of work on view update for XML deals with the case of the base data being stored in a relational database. This is a difficult problem as it involves handling two different query paradigms and data models. The Round-trip XML View Update Problem (RXU), a subcase of XQuery view update, is handled by Rainfall [141], a decomposition based update translator for virtual views. RXU refers to the combination of loading XML data into a relational database and using XML publishing to obtain the XML views. By splitting XML into relations, Rainfall shows that view update operations can always be translated into relational updates for RXU. The Rainbow XML data management system [142] [143] was used to implement the presented methods for update decomposition, translation and propagation. However, the algorithms and concepts are general and can therefore be used by other systems.

As previously discussed, many methods have been developed for relational view update. Methods for XML view update generally developed alternative approaches. The possibility of mapping XML view updates onto existing relational view update methods when XML is stored in a relational database has been explored [144]. This reduces the problem to an application of relational view update. XML views and updates are mapped to a set of relational views and updates. This mapping allows any method for view update for relational databases to be used to complete the view update process. It focuses on XML views which allow nesting, composed attributes, heterogeneous sets, grouping, repeated elements and text elements with attributes and represents them as query trees. A method for nested relational algebra (NRA) views [145] is similar but far less general.

UFilter [146] is a framework for solving the XML view update translatability problem for arbitrary XML views - translating updates on the view to updates on the base data without side-effects on the view. Constraints are used for schema-level and data-level checking. The schema-level checks are update validation and schema-driven translatability reasoning. Update validation checks if the update is valid with respect to the schema. Schema-driven translatability reasoning checks for possible side-effects. Data-level checking is only performed when necessary due to it being expensive, in comparison to schema-level checks, as it requires checking the base data. This validation is the data-level translatability check which locates conflicts with the base data. Only updates which pass all these checks are translated into SQL queries.

The view update problem for arbitrary virtual XML views over relational databases is tackled in [147]. This entails translating an update into SQL queries without side-effects, if possible. Clean source[11] [133] was used to define the schema conditions in which a relational view is updatable when it is over a single table. This work extends this concept for XML, which is used to determine the correctness of a translation mapping. Updates are classified as untranslatable, conditionally translatable and unconditionally translatable. These classifications depend on the view and the update's features: update granularity for the view, view construction properties and duplication types contained within the view. The algorithm handles the updates depending on their classification: untranslatable updates will

[11]source data of a view that can be updated without resulting in view side-effects

be discarded; more conditions will be requested for conditionally translated updates; and unconditional translatable updates are translated. The main concern for the translatability algorithm is conflict between the relational foreign key constraints and the view hierarchy. The algorithm is efficient as it does not require the database contents, only the view and database schema and runs in polynomial time. The algorithm is extended [148] to support views with duplication. These works differ from [146] in that conflicts can still arise with the base data as only schema reasoning is used.

The view update problem for potentially recursive XML views populated from relational data is also studied in [7]. XML views are represented as mappings defined by DTDs which are stored in relations as directed acyclic graphs (DAGs). XML is a tree structure, however, DAGs are used in the representation as a recursive view could result in infinitely many relational views for a tree structure (e.g., [144]). Another encoding scheme which handles recursion is presented in [149]. This method can handle recursive views - defined in terms of itself, directly or indirectly - and recursive XPath queries. Algorithms are presented for relational view insertions and deletions.

The method presented can handle XML views obtained via schema-directed XML publishing. XML publishing is defined as translating relational data into XML. The work mainly deals with Attribute Translation Grammars (ATGs) [109] however it can support any system that uses Select Project Join (SPJ) queries to define XML views (such as SilkRoute [107], XPERANTO [104]).

Relational views are defined for the Directed Acyclic Graph (DAG) representation of the XML view in such a way that the size of the XML view limits the number of relational views. This holds even in the case of recursive XML views. This is in contrast to other encoding schemes, e.g., [144], which could result in an infinite number of relational views for recursive XML views. Updates to an XML view are handled as follows: updates which are single path expressions at the XML level are translated to group updates (multiple SQL queries each representing a single path) on the relational view level and then translated to updates on the published relational database.

Algorithms are presented to translate XML view updates to updates on the relational representation with the ability to handle recursive updates and recursively defined XML views. Another algorithm is presented to evaluate XPath queries on DAGs with the potential to contain complex filters. This makes use of index structures: reachability matrix for handling ancestor-descendant relationships and a topological order for handling filters and computing and maintaing the reachability matrix. This algorithm also has the ability to detect side-effects. An algorithm is also presented to incrementally maintain these indexes. This takes place in the background in parallel with relational updates. This is so the XML view updates are not affected. To implement these, the update semantics of XML views represented by relations were enhanced in order to take account of XML update side-effects. On the relational side, algorithms are presented for processing group deletions and processing group insertions on SPJ views.

The presented methods were the first that offered support for recursive XML updates and complex filters on XML views which are compressed and potentially recursive, without passing responsibility to the relational DBMS. The problem with previous methods relying on the DBMS was that the required functionality was mostly unavailable. The presented methods can support XML view updates within the immediate reach of the majority of XML publishing systems. Side-effects are handled by the user. Should one be detected the user can decide to quit or continue in accordance with semantics they have defined.

The presented method works in three stages. The update is validated with respect to the DTD, this update is then translated into an update on the DAG (stored in relational views) to maintain it and finally the update on the views is translated into updates to be applied to the relational database. Detected side-effects are handled as defined.

The presented algorithm for evaluating an XPath query on a DAG assumes it is stored in edge relations. Previous work focussed on trees rather than DAGs. Existing algorithms for the evaluation of path queries on DAGs [150] [151] could not be used as they do not support complex filters or maintenance of the indexes. There is also work present in the area of path evaluation for semi-structured data - general graphs - but they treat DAGs the same as cyclic graphs which might not be efficient. Additionally they do not consider XML view updates using XPath queries.

The algorithms scale linearly with the relational database size. XPath evaluation dominates the insertion and deletion time. This performance time increases for deletions with the number of edges generated by the XPath queries as they all have to be examined by the delete algorithm. The benefit of incremental maintenance increases with the data size.

View Update Techniques in Commercial DBMSs

Similarly to view maintenance support in commercial DBMSs, view update support is also very restricted or not yet available. SQL server [152] provides XML publishing and view update functionality. However, for view update, it only supports a restricted set of views. The only joins allowed are key-foreign key and recursive views and updates (recursive XPath) are not supported. An updategram is used to support view updates, this stores the changes to the XML after the update. The updategram is then used to convert the update into SQL statements to update the base tables.

2.6 Conclusions

This chapter focussed on previous work covering XML indexing, query evaluation, incremental methods and views, all of which technologies contribute to the work in the remainder of this thesis. It finished by describing the most closely related work to this thesis, view maintenance for relational and XML databases and view update for relational databases. The main contribution of this thesis is similar incremental algebraic methods which provide a consistent solution to the view maintenance and view update problems. The originality of these methods lies in their ability to handle statement-level updates using generic operators and views returning data from multiple nodes. The next chapter describes the background technologies and implementation details required to understand the view maintenance and view update methods presented in Chapter 4.

3 Preliminaries

This chapter discusses the background technologies and implementation details required to explain the methods presented in Chapter 4 and the associated experimental work. EXtensible Markup Language (XML) and related technologies: XPath, XQuery and XQuery Update are discussed. Following this, views and their specifications are explained. Finally, Views in Peer-to-Peer (ViP2P), helper data structures, functions and operators are detailed.

3.1 EXtensible Markup Language (XML)

This work deals with EXtensible Markup Language (XML) documents. XML is used to describe semi-structured data and was developed to allow the transfer of information between different systems/applications. In this work XML documents are seen as ordered labelled trees consisting of the following nodes (the labels are taken from a finite set of XML node

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<university>
    <department name = "CIS">
        <student>
            <name>Martin Goodfellow</name>
            <age>28</age>
            <email>martin.goodfellow@strath.ac.uk</email>
            <course>PhD</course>
        </student>
            ...
    </department>
        ...
</university>
```

FIGURE 3.1: Students XML Document

labels, $\mathcal{L}$):

- element - represents an object and consists of a label
- attribute - provides additional information about an element and consists of a label
- text - consists of a string representing a value

An XML document can be seen in Figure 3.1. This shows a fragment of a document which stores the students for each department in a university.

3.1.1 IDs

For the purpose of this work, each node must have a unique identifier (ID). This is given based on an encoding scheme and is represented by a compact, unique string. The encoding scheme used was Dynamic DEwey (DDE) [3]. The selection was based on the following properties:

- The structural relationship, if one exists, between two nodes can be determined. This is a requirement for the developed view maintenance and view update methods.
- From a node ID, the IDs of all the node's ancestors can be determined.
- Relabelling is not required when an update occurs.
- A compact method can be used for encoding.

An XML document using the encoding scheme can be seen in Figure 3.2. Each ID is a sequence of steps, where each step holds the relative position of each ancestor of the node, labelled from the root. In order to improve the developed view maintenance and view update methods (see Sections 4.1.1 and 4.2.3) DDE was extended to include the label path. This allowed the use of more powerful pruning methods at the expense of making the ID scheme less compact. An XML document using the extended scheme can be seen in Figure 3.3.

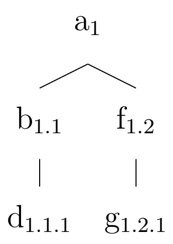


FIGURE 3.2: XML document using DDE

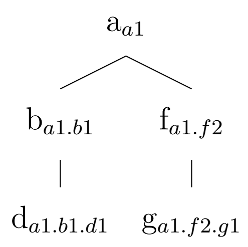


FIGURE 3.3: XML document using extended DDE

3.2 XML Query Languages

As the volume of XML data increased the need to query the data emerged. The World Wide Web Consortium (W3C) first introduced XPath [9] for path queries, followed by the more expressive XQuery [8].

3.2.1 XPath

XPath allows the selection of nodes or attributes in an XML document that match a given query, where the query is expressed as a path. It supports parent-child and ancestor descendant relationships; predicates; selecting unknown nodes; selecting several paths (using an OR operator); several axes; and numerous operators. For example, in the document in Figure 3.4, the g node could be selected using the XPath query $/a/f/g$. XPath is used in the view maintenance and update methods to identify the target nodes for the updates.

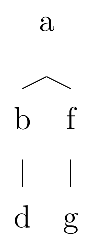


Figure 3.4: XML document

3.2.2 XQuery

XQuery is a superset of XPath and was developed to allow powerful querying of XML, similar to SQL for relational databases. The biggest difference is the ability to handle FLWOR (For, Let, Where, Order by, Return) expressions. Example 3.2.1 shows an XQuery query using FLWOR.

Example 3.2.1. *Let the XML document "students.xml" be as seen in Figure 3.1. The following query returns the names of all students over 21 in alphabetical order.*

```
for $x in doc ("students.xml")/university//student
where $x/age > 21
order by $x/name
return $x/name
```

XQuery Update

XQuery Update [15] is an extension of the XQuery language which can be used to update XML documents or data. It supports five basic operations:

- insert
- delete
- replace node
- replace contents/value of a node
- rename

The following example shows the insertion of a new student into a document.

Example 3.2.2. *Let the XML document "students.xml" be as seen in Figure 3.1. The following query inserts a new student, John Smith, into the CIS department.*

```
insert nodes
<student>
  <name>John Smith</name>
  <age>19</age>
  <email>john.smith@strath.ac.uk</email>
  <course>BSc (Hons) Software Engineering</course>
</student>
into doc("students.xml")/university/department[@name="CIS"]
```

The developed methods for view maintenance and view update make use of XML for the documents, XPath to identify the target nodes for the update and XQuery for the views. The methods handle a subset of XQuery Update statements.

3.3 Views

A database view is a virtual table containing the result of a query. Conversely, a materialised view stores the result in a persistent table. The work described in this thesis is focussed on manipulation of materialised views.

The views considered in this work are expressed using a subset of XQuery. This conjunctive dialect is presented in Table 3.1.

- $\mathcal{XP}$ represents the XPath$^{\{/,//,*,[]\}}$ language.
- $absVar$ is an absolute variable declaration. The variable x_i is bound to the path expression $p \in \mathcal{XP}$ evaluated on the document at URI uri starting from the root.
- $relVar$ is a relative variable declaration. It is non-terminal. The variable x_i is bound to the path expression $p \in \mathcal{XP}$ evaluated starting from the bindings of a previously introduced variable x_j. Where is optional and is a conjunction over predicates. Each predicate compares the string value of a variable x_i with a constant c.
- return builds a new element labelled l with some children labelled l_i ($l, l_i \in \mathcal{L}$), for each tuple of bindings of the for variables. Each child element can contain one of the following pieces of information related to the current binding of a variable x_k, declared in the for clause:
  - x_k denotes the full subtree rooted at the binding of x_k.
  - id(x_k) denotes the ID of the node to which x_k is bound (see Section 3.1.1).
  - string(x_k) is the string value of the binding.

The full subtree, or content, of a node contains all its descendants - element, attribute, and text - whereas the string value only contains the text descendants. For example, for the

1	$q :=$	(let $absVar$ return)? for $(absVar,)?$ $relVar$ $(relVar\ ,)^*$ (where $pred$ (and $pred)^*)?$ return ret	
2	$absVar :=$	x_i in doc(uri) $/p$	where $p \in \mathcal{XP}$
3	$relVar :=$	x_i in x_j $/p$	where x_j introduced before x_i
4	$pred :=$	string(x_i) $= c$	
5	$ret :=$	$\langle l \rangle$ $elem^*$ $\langle /l \rangle$	
6	$elem :=$	$\langle l_i \rangle \{$ $(x_k/p$ | id(x_k) | string(x_k)) $\} \langle /l_i \rangle$	where $p \in \mathcal{XP}$

Table 3.1: Grammar for XML materialised views

XML document in Figure 3.4, the content of *b* is <b><d/></b>, whereas the string value is null, as *b* has no text descendants. Therefore, the string value is generally smaller but holds less information. By storing these extra pieces of information the number of queries that can be rewritten using a view increases. The IDs do not hold any of the content but enable joins to combine tree patterns (XML queries). The view dialect allows significant flexibility by allowing any subset of ID, value and content to be returned for any of the variables.

3.3.1 XML Access Modules (XAMs)

XML Access Modules (XAMs) [1] can represent an XML storage, index or materialised view. In general terms, XAMs are persistent data structures storing parts of an XML document. For the purpose of this work they are used to model views.

The subset of XAM specifications used can be defined using the grammar in Figure 3.5, where bold font represents constants. Each node can carry the label of an XML element or attribute and can be annotated with the following specifications:

- ID - A Dynamic DEwey (DDE) ID [3] is stored. The **u** states that it is an updatable

$$NS ::= N^+$$
$$N ::= name\ \ IDSpec?\ \ TagSpec?\ \ ValSpec?\ \ ContSpec?$$
$$IDSpec ::= \textbf{ID}\ \ \textbf{u}$$
$$TagSpec ::= (\textbf{Tag})\ \mid\ [\textbf{Tag=}c]$$
$$ValSpec ::= (\textbf{Val})\ \mid\ [\textbf{Val=}c]$$
$$ContSpec ::= \textbf{Cont}$$
$$ES ::= E*$$
$$E ::= name_1(/\,|\,//)\ \ \textbf{j}\ \ name_2$$

FIGURE 3.5: XAM Grammar [1]

ID[1], other ID types are supported by XAMs but are not used in this work. This can be stored for all view elements and attributes.

- Tag - **Tag** denotes that the element tag or attribute name is stored for that node. Alternatively, [**Tag=**c] declares that only elements matching this predicate are stored.

- Cont - The XML of the subtree rooted at this element or attribute is stored. Formally, this is the full (serialised) representation of the element or attribute.

- Val - This is defined similarly to Tag with the exception that it refers to an element or attribute's value. It is obtained by concatenating all its text descendants in document order, i.e, the cont of the element or attribute with the XML tags removed.

ES is the edge specification. The relationships supported are parent-child (/) and ancestor-descendant (//). The **j** states that it is a join edge. Other XAM edges are available but are not used in this work.

An example XAM can be seen in Figure 3.6. The tuples in this view store data from all a elements with an f child, where the f child has a g child, i.e, all subtrees matching //a/f/g. The IDs are stored for all elements and for g elements the val and cont is also stored.

[1]This work extended XAMs to support updatable IDs, expressed as DDEs.

1: ID u [Tag=a]

2: ID u [Tag=f]

3: ID u [Tag=g] Val Cont

;

1,2 / j

2,3 / j

Figure 3.6: XAM

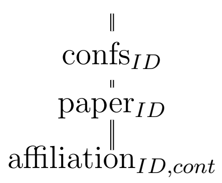


Figure 3.7: Sample tree pattern.

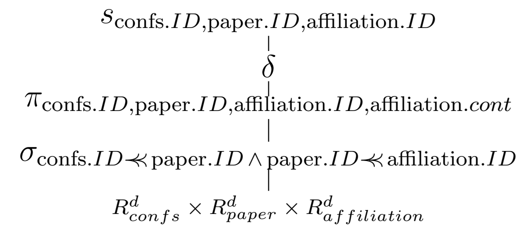

$s_{\text{confs}.ID,\text{paper}.ID,\text{affiliation}.ID}$
δ
$\pi_{\text{confs}.ID,\text{paper}.ID,\text{affiliation}.ID,\text{affiliation}.cont}$
$\sigma_{\text{confs}.ID \prec\!\!\prec \text{paper}.ID \wedge \text{paper}.ID \prec\!\!\prec \text{affiliation}.ID}$
$R^d_{confs} \times R^d_{paper} \times R^d_{affiliation}$

Figure 3.8: Algebraic semantics of the tree pattern in Figure 3.7.

3.3.2 Algebraic Tree Pattern Semantics

View semantics are defined using tree embeddings [153]. Equivalent semantics are used which are presented by an algebra [154].

Given a document d and label a, the list of tuples of the form $(ID, val, cont)$ obtained from all the a-labelled nodes in d is denoted by R^d_a. This is referred to as the virtual canonical relation of a in d. R^d_a's tuples are stored in the order of appearance of the corresponding nodes in d. A parent comparison operator, $\prec$, and ancestor comparison operator, $\prec\!\!\prec$, are also introduced. These return true if the left-hand ID is the parent or ancestor, respectively, of the right-hand ID. The algebra discussed is only logical and no assumptions are made on the physical implementation of $\prec$ and $\prec\!\!\prec$.

Let $\mathcal{A}$ be the algebra consisting of the following operators: (1) the n-ary cartesian product $\times$; (2) selection, denoted σ_{pred}, where $pred$ is a conjunction of predicates of the form $a \odot \underline{c}$ or $a \odot b$, a and b are attribute names, $\underline{c}$ is some constant, and $\odot$ is a binary operator among $\{=, \prec, \prec\!\!\prec\}$; (3) projection, denoted π_{cols}; (4) duplicate elimination, denoted δ; (5) sort, denoted s_{cols}. Joins are also used, defined, as usual, as selections over $\times$.

The algebraic semantics of the tree pattern in Figure 3.7 is the algebraic expression in Figure 3.8. From the bottom up, there is an R_a atom per query node labelled a, connected through $\times$ operators. The selection σ enforces (i) all value constraints on the nodes, and (ii) all structural $\prec$ or $\prec\!\!\prec$ relationships between query nodes. The projection retains the attributes projected by query nodes, e.g., confs.ID, paper.ID, affiliation.ID and affiliation.-$cont$. After duplicate elimination (δ), the tuples are sorted in the order dictated by the IDs of the bindings of all nodes.

3.3.3 Views in Peer-to-Peer (ViP2P)

The view maintenance and view update methods described in this thesis were implemented within ViP2P[2]. This is an open source system that offers good view management support so was suitable for this research. It is a peer-to-peer XML database based on distributed hash table (DHT) indices and exploits materialised views independently published by the peers in the P2P network, to answer an interesting dialect of tree pattern queries. When a query arrives at a peer it uses the views available in the network to rewrite the query in order to evaluate it. The main architecture of a peer can be seen in Figure 3.9. It consists of data storage which stores the tuples contained in the views located on this peer. It also stores modules to manage each of the files that can arrive at a peer. The work presented in this thesis added the update management module to the ViP2P system which handles both view maintenance and view update.

[2]http://vip2p.saclay.inria.fr/

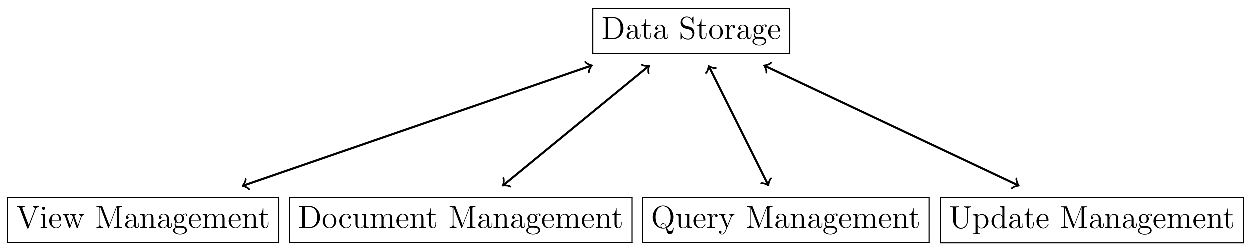


FIGURE 3.9: ViP2P Peer Architecture

3.4 Helper functions and operators

The view maintenance and view update methods, presented in Chapter 4, assume the availability of a number of helper functions and operators. These have been incorporated into the produced methods.

Let u be an update (insertion or deletion), and $pul(u)$ be the pending update list [15] resulting from u on a document d. The pending update list represents the updates to be performed on the document. Thus:

- if u is an insertion, $pul(u) = \{(n_1, t_1), (n_2, t_2), \ldots, (n_k, t_k)\}$, is a list of pairs consisting of an XML element, n_i, the target of the update, which also contains the ID of the last child of the XML element (if it has children) and a subtree t_i to be copied as a child of n_i.

- if u is a deletion, $pul(u) = \{n_1, n_2, \ldots, n_k\}$ is the list of the nodes to be removed.

Let vu be a view update (insertion or deletion), and $pul(vu)$ be the pending view update list resulting from vu on a view v. The pending update list represents the updates to be performed on the view. Thus:

- if vu is a view insert, $pul(vu) = \{(vn_1, t_1), (vn_2, t_2), \ldots, (vn_k, t_k)\}$, is a list of pairs consisting of view node ID information vn_i (this includes the ID of the document to which the node belongs, as well as the node's ID within the document and the ID of its last child, if it has children), and a subtree t_i to be copied as a child of vn_i.

- if vu is a deletion, $pul(vu) = \{vn_1, vn_2, \ldots, vn_k\}$ is the list of the tuples to be removed.

The following functions are used in the view maintenance and view update methods:

- **compute-pul(u, d)** is a function which from an update u and a document d, computes its pending update list $pul(u)$.

- **compute-pul(vu, v, d)** is a function which from a view update vu, a view v and documents d[3], computes $vu's$ pending view update list $pul(vu)$ for insertions.

- **compute-pul(vu, v)** is a function which from a view update vu and a view v computes $vu's$ pending view update list $pul(vu)$ for deletions. This is different from insertions as deletions do not need to assign new IDs, so the ID of the last child of target nodes is irrelevant. All the required information is contained in the view.

- **apply-insert(n, t)** is a function which, given a node n and a tree t, copies t into a new tree t', inserts t' as a new child of n and returns t'. Importantly, the tree t' also includes the IDs assigned to the copied t nodes in their new context (in d).

- **apply-insert(vt, t)** is a function which, given a view tuple vt and a tree t, copies t into a new tree t', inserts t' as a new child of vt (viewing vt in tree form) and returns t'. Importantly, the tree t' also includes the IDs assigned to the copied t nodes in their new context (in v).

- **extr-pattern(n_{ID}, p, t)** is a function which, given a target node ID and the ID of its last child n_{ID}, a tree pattern p and an XML tree t, evaluates p on t and returns the corresponding set of tuples. The IDs for the tuples are assigned as children of the target node ID. The ID of its last child (if it has children) is used to ensure the new IDs are unique.

- **extr-pattern(IDs, vd)** is a function which, given IDs of nodes to be deleted from the source IDs and a view definition vd returns the corresponding set of tuples to be deleted from the view. This operates by only returning tuples which are contained in vd.

[3]These are required if the ID of the last child is not available from a cont attribute. See Section 4.2.3.

- **extr-pattern(**vn_{ID}**,** p**,** t**)** is a function which, given a document ID, target node ID and the ID of its last child vn_{ID}, a tree pattern p and an XML tree t, evaluates p on t and returns the corresponding set of tuples. The IDs for the tuples are assigned as children of the target node ID. The ID of its last child (if it has children) is used to ensure the new IDs are unique. The document ID is required as tuples can exist from multiple documents.

- **operators** assumed to be available include physical structural joins [155]. Structural joins are joins which are defined on structural relationships as opposed to primary key-foreign key relationships. These operators implement the Stack Tree Ancestor and Stack Tree Descendant algorithms. Based on Dynamic Dewey IDs [3], Path Filter is used to determine if a node ID is on a path and Path Navigate for discovering parent IDs of nodes based on their IDs.

3.5 Term evaluation based on lattice

Views can be seen as a join expression and new tuples as well as tuples to be deleted as Δs, see Sections 4.1.1 and 4.1.2 for more information. To maintain the view v, auxiliary data structures, organised as a *view lattice*, are utilised. This lattice is an AND-OR graph. Figures 3.10, 3.11 and 3.12 depict the lattices corresponding to the views $//a_{ID}//b_{ID}//c_{ID}//d_{ID}$, $//a_{ID}[//b_{ID}][//c_{ID}]//d_{ID}$ and $//a_{ID}[//b_{ID}//c_{ID}]//d_{ID}$. The boxed nodes are explained in Definition 3.5.1. This section explains the lattice in the context of insertions, however, deletion lattices are handled similarly.

The lattice of v is a directed acyclic graph (DAG) which can contain three different kinds of nodes:

- (i) *pattern-labelled nodes* - The root node of the lattice is labelled by the pattern v. Every other pattern-labelled node is labelled by a distinct subtree pattern of v. For readability, in Figures 3.10, 3.11 and 3.12, the joins are removed from all pattern labelled nodes.

- (ii) OR-nodes labelled $\vee$

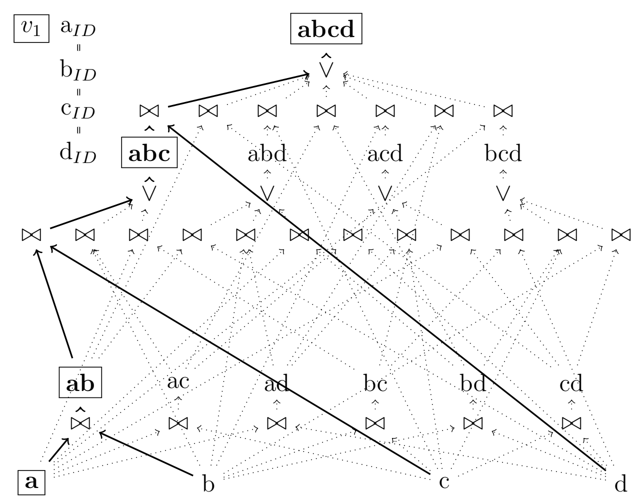


FIGURE 3.10: Subpattern lattice and snowcaps for the view v_1.

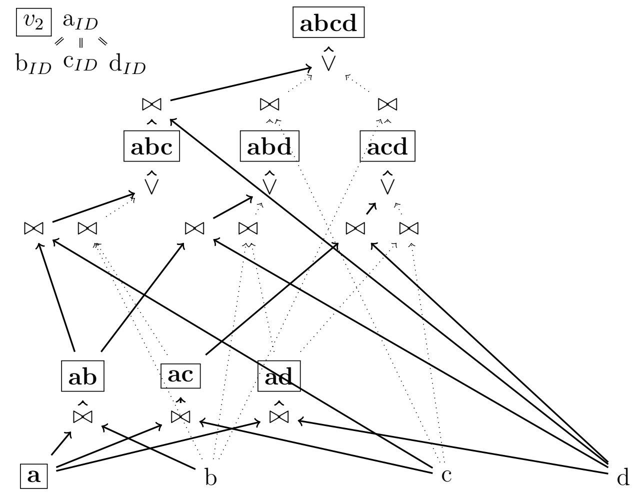


FIGURE 3.11: Subpattern lattice and snowcaps for the view v_2.

- (*iii*) Join nodes labelled $\bowtie$.

A DAG is used for this structure as a child can have multiple parents.

The computation of a pattern-labelled node can be determined by its child. Each pattern-labelled node has one child which is either a join ($\bowtie$) or an OR ($\vee$) node. A join defines how to compute the pattern-labelled node. For example, the pattern-labelled node *ab* in Figure 3.10 can be computed by the join of children *a* and *b*. An OR states that there are several ways to compute a pattern-labelled node. Its children are a set of join nodes that define these alternatives. For example, in Figure 3.11 *abc* can be computed by either the

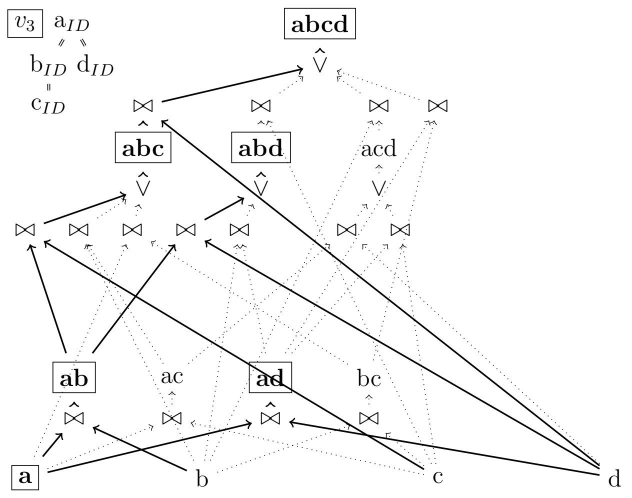


FIGURE 3.12: Subpattern lattice and snowcaps for the view v_3.

join of ab and c or the join of ac and b. When faced with a choice of how to compute the tuples for a node, the first join node was selected. The chosen approach to maintain a node is represented in the Figures by the bold arrows.

Materialising (and maintaining) all pattern-labelled lattice nodes is enough to maintain v after an insertion or deletion. For each union term t to be added to v:

- Let t_R be the $\bowtie$ subexpression(s) of t containing only R_a occurrences. Then, t_R corresponds exactly to some materialised lattice node(s).
- Let t_{Δ^+} be the $\bowtie$ subexpression(s) of t containing only Δ^+ occurrences. Then, t_{Δ^+} can be computed using the new tuples to be inserted, i.e., the Δs.

By making use of a lattice, the problem of maintaining a view can be reduced to maintaining the subpatterns of a view. This is a potential solution as maintaining the lattice leaves is trivial (e.g., replace R_a by $R_a \cup \Delta_a^+$) along with the propagation of the updates to the lattice root. However, a lot of space and time is required to materialise and maintain all these lattice nodes. Fortunately, only a subset of these nodes have to be handled:

Definition 3.5.1 (Snowcap)**.** *Let v be a tree pattern. The* snowcap *of v is any non-empty subtree t of v such that: for each node $n \in v$ that also appears in t, the parent of n in v also* *appears in t.*[4]

Snowcaps are displayed as boxed nodes in Figure 3.10. For example *abc* is a snowcap as the parent of each node is included in the tree pattern, the b parent of c, the a parent of b and since a is the root it does not a have a parent and the tree pattern is therefore a snowcap. Conversely, *bcd* is not a snowcap as both d and $c's$ parent are included in the tree pattern but $b's$ is not and it is not the root. A snowcap copies the root of the pattern and then goes down to some length on all paths, only including a node n if it includes its parent. This name comes from the way snow covers mountains, from the top down.

After the update expression has been computed using the view definition and the new tuples/tuples to be deleted, the expression can be pruned. This pruning is performed based on what subexpressions can be determined to be empty. The pruning methods developed are detailed in Chapter 4. The first pruning method (Proposition 4.1.4) takes XML update semantics into account which state that new children can be added to a document but not new parents. Therefore, expressions which attempt this can be ignored. It can now be stated:

Proposition 3.5.2. *Let v be a view, u an insertion and t a term resulting from u. The term t survives the first pruning (Proposition 4.1.4) if and only if t_R (the subexpression of t which does not contain Δ^+ symbols) is the algebraic semantics of a pattern v_{t_R}, which is a snowcap in v's lattice.*

Proof. "If" direction: if t_R corresponds to a snowcap in v's lattice, then t is not eliminated by Proposition 4.1.4. Since v_{t_R} is a node in v's lattice, it corresponds to a subset N_{t_R} of v's nodes. The term t can be written as a join between t_R and the Δ^+ tables for all the v nodes that are not in N_{t_R}, where the join predicates are derived from the structure of v. Let n_1 be a v node and n_2 a / or //-child of n_1. Since v_{t_R} is a snowcap, exactly one of the following must hold:

- n_1 and n_2 are both in v_{t_R}, therefore R_{n_1} and R_{n_2} are both in t_R and t. In this case, n_1 and n_2 do not lead to t being pruned by Proposition 4.1.4.

[4]This can also be viewed as an upwards closed set of the tree.

- n_1 is in v_{t_R}, thus R_{n_1} is in t_R (and t), while n_2 is not in v_{t_R} and thus $\Delta^+_{n_2}$ appears in t. The t subexpression $R_{n_1}\Delta^+_{n_2}$ does not trigger the pruning of Proposition 4.1.4 either.
- neither n_1 nor n_2 are in v_{t_R}, thus they do not appear in t_R and t features $\Delta^+_{n_1}\Delta^+_{n_2}$ as a subexpression, which again cannot trigger the condition of Proposition 4.1.4.

"Only if" direction: If t survives the pruning of Proposition 4.1.4, then t_R corresponds to a snowcap in v's lattice. If t includes no Δ^+ table, then $t_R = t$ coincides with the lattice's topmost node, which is a snowcap. Now assume that t includes at least one Δ^+ table, corresponding to the view node n. Moreover, since t was not pruned by Proposition 4.1.4, it follows that for any node n' immediately under n in v, t also contains the Δ^+ table corresponding to n'; repeating this reasoning shows that for t' contains $\Delta^+_{n'}$ for any descendant n' of n, one of the two cases below must hold:

1. For any ancestor m of n in v, t includes Δ^+_m. Thus, t includes the Δ^+ corresponding to v's root, therefore t includes Δ^+ tables for all v nodes, thus $t_R = \emptyset$, which verifies the conclusion.
2. There exists a lowermost ancestor m of n such that t does not contain Δ^+_m (equivalently, such that R_m appears in t_R). Then it is easy to see that for any ancestor m' of m, $R_{m'}$ appears in t_R; in other words, all nodes from m upward to v's root must appear in t_R.

Generalising the above reasoning to all v nodes n such that Δ^+_n appears in t, it is determined that for any node m in v_{t_R}, all ancestors of m are in v_{t_R}. In other words, v_{t_R} is a snowcap. □

To give an example of this proposition, consider the view v_3 in Figure 3.12. For an insertion u to add tuples to v_3, one or several of the following cases must hold:

- (*i*) u adds a d child to element(s) matching the path $//a//b//c$. View maintenance is performed by joining the snowcap abc with Δ^+_d.
- (*ii*) u adds a c child to element(s) matching $//a//b$, and this c child has at least one d descendant (join the snowcap ab with $\Delta^+_c\Delta^+_d$).

- (*iii*) u adds a b child to element(s) matching $//a$, and this b child has at least one $//c//d$ descendant (join the snowcap a with $\Delta_b^+\Delta_c^+\Delta_d^+$).
- (*iv*) u adds matches to the full $//a[//b//c]//d$ view path. The lattice is not required for this case.

By showing all the scenarios that can occur when adding new tuples to a view, in this example, it can be seen that the snowcaps are necessary and sufficient to maintain the view. This can be generalised into the following proposition:

Proposition 3.5.3. *Each snowcap can be maintained based only on other snowcaps, the lattice leaves and the Δ^+ relations extracted during an update.*

Proof. The proof is by induction on the number of nodes, k, of the snowcap tree pattern. If $k = 1$, the snowcap is a leaf, and it can be maintained by adding to it the corresponding Δ^+ tuples. If $k > 1$, by the induction hypothesis, any snowcap of $k - 1$ nodes can be maintained based on other (smaller) snowcaps. Moreover, there exists a snowcap s_{k-1} such that (i) s_{k-1} has all the view nodes appearing in s_k but one, (ii) there is a $\bowtie$ node in the lattice pointing to the $\vee$ node which points to s_k, and such that the s_{k-1} node points to the $\bowtie$. This means that s_k can be incrementally maintained by joining the tuples added to s_{k-1} (as part of its own incremental maintenance) with those from the leaf node that is in s_k but not in s_{k-1}. $\square$

Observe that since v is a snowcap itself, it trivially follows that maintaining the snowcaps is sufficient to maintain v.

Snowcaps are sufficient to maintain a view. However, it is not always necessary to store every snowcap. For example, in Figure 3.11, *abcd* can be maintained by *abc*, *ab* and *a* along with the leaves *b*, *c*, and *d*. The snowcaps *abd*, *acd*, *ab*, and *ad* are not required.

The proof of proposition 3.5.3 actually shows that each snowcap can be maintained by joining one smaller snowcap with a leaf. The most efficient implementation depends on the data set statistics: the number of tuples for each subpattern in the lattice; the size and number of the Δ^+ tables corresponding to the various view nodes (which can be seen as reflecting the nature of the update, since some elements may be added in greater numbers than others); the join order; join processing costs; etc. Experiments were performed to compare two alternative implementations: maintaining a minimal set of snowcaps and leaves, versus maintaining only the lattice leaves with internal joins being recomputed on the fly. The results are presented in Chapter 5.

Having established the preliminaries necessary to explain the methods, Chapter 4 presents this material and examines its application in view maintenance and view update.

4 Methods

This chapter details the methods created to handle the view maintenance and view update problems in the context of the XML data model. Initially, the methods for view maintenance and view update are discussed, followed by an optimisation using dynamic reasoning.

4.1 View Maintenance

This section describes the developed solution to the view maintenance problem. View maintenance is the problem of maintaining materialised views as a consequence of changes to the underlying database. In more formal terms view maintenance can be described as follows: Let v be a view and u an update on a document d. View maintenance involves transforming v into v' so as to reflect the effect of u on d. This can be seen in Figure 4.1. The remainder of this section will describe algorithms for transforming v into v' so as to reflect the effect of

$$\begin{tikzcd}
v(d) \arrow[r, dashed, "\text{update propagation}"] & v(d') \\
d \arrow[u, dotted, "\text{view evaluation}"] \arrow[r, "\text{document update}"] & d' \arrow[u, dotted, "\text{view evaluation}"']
\end{tikzcd}$$

Figure 4.1: View Maintenance.

u on d.

4.1.1 Insertions

This section will discuss the propagation of insertions. The following example illustrates a simple insertion scenario.

Example 4.1.1. *Consider the view $//a_{ID}//d_{ID}$ and the XML document d shown in Figure 4.2, where the ID of each node is shown as a subscript. Consider an update u_1 adding the XML fragment $\langle d/\rangle$ into $//a$. When evaluated on the document, this results in entering a new d child of a, $d3$. Therefore, this produces a new tuple $(a1, a1.d3)$ which must be added to the view.*

The developed method for insertions, which is discussed in the remainder of this section, can be explained in simple terms as follows: first the new tuples to be inserted in the view are computed from the source document and the XML fragment to be inserted. Then the algebraic expression is obtained from the view to be maintained, based on its definition. This expression can then be modified, using the new tuples to be inserted, to represent the update expression required to maintain the affected view. To improve efficiency, a number

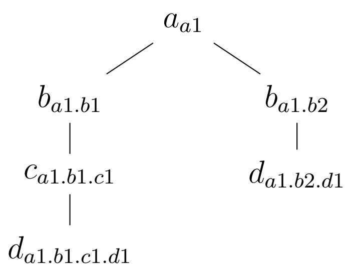


Figure 4.2: Sample XML Document.

of pruning rules are utilised to remove unnecessary terms from this update expression. This expression can then be evaluated to maintain the view and finally the source document is updated.

The rest of this section is structured as follows: First the general method is outlined, then a set of pruning methods are introduced, which reduces the computations required for view maintenance. Finally, the algorithms are introduced for handling insertions.

Method

The approach developed, relies on the algebraic view semantics, as follows: Assume that the nodes of the view v carry the node names $a_1, a_2, \ldots, a_k \in A_l$, where $k \in \mathbb{N}$ and A_l is a tag alphabet (set of tags), $A_l = \{a, b, c, \ldots\}$. Then, v can be written as:

$$v = e_v(\sigma_{a_1}(R_{a_1}) \bowtie \sigma_{a_2}(R_{a_2}) \bowtie \ldots \bowtie \sigma_{a_{k-1}}(R_{a_{k-1}}) \bowtie \sigma_{a_k}(R_{a_k}))$$

where R_{a_i} defines the virtual canonical relation scan of a_i (as defined in Section 3.3.2) and for each a_i, σ_{a_i} is a selection operator. If the view node a_i has a value predicate then this is its logical condition, otherwise it is true as the predicate is on the tag which we know to be true by the definition (only labels with the specified tag are stored within the canonical relation scan). The algebraic expression, e_v, includes the projections, sorts, and duplicate eliminations in the algebraic semantics of v. The joins $\bowtie$ correspond to the specific structural relationship predicates connecting the a_i nodes in the view v.

The nodes added by u to d are referred to as new nodes. For any node label l, Δ_l^+ is defined as the ordered collection of tuples of the form $(n.ID, n.val, n.cont)$ (assuming all the attributes are stored for the node) for all n nodes the update adds to the document. The IDs of the new nodes are computed as a side-effect of querying the document for the target node(s), whereas their values and contents can be extracted directly from the XML fragment to be inserted. The effect of u on d can be expressed, in the context of Δ^+ relations, as follows: for each node label l occurring in v, replace R_l^d by $R_l^{d'} = R_l \cup \Delta_l^+$.

After the update, if there are delta relations for all the view node labels, the tuples of the view v will become:

$$v' = v(d') = e_v(\sigma_{a_1}(R_{a_1} \cup \Delta^+_{a_1}) \bowtie \sigma_{a_2}(R_{a_2} \cup \Delta^+_{a_2}) \bowtie \ldots \bowtie \\ \sigma_{a_{k-1}}(R_{a_{k-1}} \cup \Delta^+_{a_{k-1}}) \bowtie \sigma_{a_k}(R_{a_k} \cup \Delta^+_{a_k}))$$

Thus, v' can be determined by evaluating the expression e_v over the result of the join expression. The developed method to compute v' is:

- (i) compute e_v's input, i.e., the join expression
- (ii) apply the unchanged algebraic expression e_v on the result

Since (ii) is relatively simple after computing the join expression, e_v is ignored in the remaining examples and task (i) becomes the focus with the aim of efficiently and incrementally maintaining the join expression.

Distributing the joins over unions in the expression above produces a single union of 2^Δ *terms* - where Δ is the number of deltas - each of which is a join expression. The join expressions containing no Δ^+ relations corresponds to the original view v. Therefore, the remaining $2^\Delta - 1$ union terms must be evaluated to propagate u.

Example 4.1.2. *Let d be a document and u_1 be an update that inserts in d the following XML fragment:*

$$xml_1 = \langle a\rangle\langle b/\rangle\langle b\rangle\langle c/\rangle\langle /b\rangle\langle /a\rangle$$

Let a_1, b_1, b_2 and c_1 be the XML elements inserted in d by u_1 and consider the view $v_1 = //a_{id}//b_{id}//c_{id}$. The Δ^+ relations, i.e., the set of tuples corresponding to u_1, are:

Δ^+_a	Δ^+_b	Δ^+_c
$(a_1.id)$	$(b_1.id)$ $(b_2.id)$	$(c_1.id)$

After u_1 is applied, v_1 should become:

$$\begin{aligned} & v_1 \quad \cup (R_a \bowtie_{a \prec\!\!\prec b} R_b \bowtie_{b \prec\!\!\prec c} \Delta^+_c) \cup \\ & (R_a \bowtie_{a \prec\!\!\prec b} \Delta^+_b \bowtie_{b \prec\!\!\prec c} R_c) \cup (R_a \bowtie_{a \prec\!\!\prec b} \Delta^+_b \bowtie_{b \prec\!\!\prec c} \Delta^+_c) \cup \\ & (\Delta^+_a \bowtie_{a \prec\!\!\prec b} R_b \bowtie_{b \prec\!\!\prec c} R_c) \cup (\Delta^+_a \bowtie_{a \prec\!\!\prec b} R_b \bowtie_{b \prec\!\!\prec c} \Delta^+_c) \cup \\ & (\Delta^+_a \bowtie_{a \prec\!\!\prec b} \Delta^+_b \bowtie_{b \prec\!\!\prec c} R_c) \cup (\Delta^+_a \bowtie_{a \prec\!\!\prec b} \Delta^+_b \bowtie_{b \prec\!\!\prec c} \Delta^+_c) \end{aligned}$$

In the remaining examples the join predicates will be omitted from the union terms for conciseness and to improve readability. Therefore, the expression to compute the new tuples of v_1, after the insertion in Example 4.1.2, can be written as:

$$v_1 \cup R_a R_b \Delta_c^+ \cup R_a \Delta_b^+ R_c \cup R_a \Delta_b^+ \Delta_c^+ \cup \Delta_a^+ R_b R_c \cup \Delta_a^+ R_b \Delta_c^+ \cup \Delta_a^+ \Delta_b^+ R_c \cup \Delta_a^+ \Delta_b^+ \Delta_c^+$$

Term pruning

Several observations lead to identifying when union terms are guaranteed to have empty results. These union terms are pruned as their evaluation is not necessary in order to propagate the insertion to the view. The evaluation workload can be reduced significantly by making use of pruning which can be generalised by the following propositions.

Pruning by the update semantics It can be defined that some terms will always have empty results from the semantics of XQuery Update [15]. This is because XQuery Update allows the addition of new children to existing nodes, but not new parents. This is demonstrated by the following example.

Example 4.1.3. *Consider the insertion u_1 from Example 4.1.2. A newly added a node cannot have as a child a b node which belonged to document d before u_1 was applied. Thus, $\Delta_a^+ R_b$ is empty, therefore the terms $\Delta_a^+ R_b R_c$ and $\Delta_a^+ R_b \Delta_c^+$ to be added to v_1 in order to maintain it are guaranteed to produce an empty result. Similarly, no b element in Δ_b^+ can have descendants in R_c, therefore $\Delta_b^+ c$ is also empty, and so are $R_a R_b \Delta_c^+$ and $\Delta_a^+ R_b \Delta_c^+$. Thus, to compute v_1', it suffices to add to v the results of evaluating the terms:*

$$R_a R_b \Delta_c^+ \cup R_a \Delta_b^+ \Delta_c^+ \cup \Delta_a^+ \Delta_b^+ \Delta_c^+$$

This is generalised by the following proposition.

Proposition 4.1.4. *Let v be a view of k nodes, and n_1, n_2 be v nodes such that n_2 is a (/ or //) child of n_1[1]. Let R_{n_1}, respectively, R_{n_2} be the tuples corresponding to n_1, respectively, n_2 in the algebraic semantics of v, i.e., $R_{n_1} \bowtie R_{n_2}$ is a subexpression of v. Let u be an arbitrary*

[1]e.g, n_1/n_2 or $n_1//n_2$ is a subexpression of the view expression.

insertion, and t be one of the $2^{\Delta} - 1$ terms to be added to v in order to propagate the effect of u. If t contains as a subexpression $\Delta^+_{n_1} R_{n_2}$, then t produces an empty result.

Observe that Proposition 4.1.4 does not depend on the the contents of the Δ relations. It only relies on the view syntax, allowing it to be performed very quickly. Therefore, the following, focuses only on the terms that survive this pruning.

Inserted data-driven pruning Further pruning may be possible from inspecting the XML fragment, as demonstrated by the following example.

Example 4.1.5. *Consider the view v_1 from Example 4.1.2 and the insertion u_2 which adds the following XML fragment:*

$$xml_2 = \langle a \rangle \langle b/ \rangle \langle b/ \rangle \langle /a \rangle$$

The difference with respect to Example 4.1.2 is that xml_2 does not include a c element, i.e., $\Delta^+_c = \emptyset$. This means that all the terms of the expression are empty and thus, v_1 is not affected by u_2.

Value predicates may also impact update propagation, as the following example shows.

Example 4.1.6. *Consider the view $v_2 = //a_{[val=5]}//b_{id}$ and the insertion u_3 adding the following XML fragment:*

$$xml_3 = \langle a \rangle 3 \langle b/ \rangle \langle b/ \rangle \langle /a \rangle$$

In this case, $\Delta^+_a \neq \emptyset$ and $\Delta^+_b \neq \emptyset$, however, $\sigma_b(\Delta^+_b) = \emptyset$ because the new a element does not satisfy the view predicate $[val = 5]$. Thus, $R_a \Delta^+_b$ and $\Delta^+_a \Delta^+_b$, which both involve $\sigma_b(\Delta^+_b)$, are empty. Since by Proposition 4.1.4 term $\Delta^+_a b$ is also empty, v_2 is unaffected by u_3.

This generalises to the following observation.

Proposition 4.1.7. *Let u be an insertion adding the trees $t_1, t_2, \ldots, t_k$ to d, and v be a view. If a node n of v is not matched in any of the trees $t_1, t_2, \ldots, t_k$, all union terms involving Δ^+_n are empty.*

Inserted ID-driven pruning The final pruning criteria reasons on the label paths leading to the insertion points, which are stored in the extended Dynamic Dewey IDs of the nodes.

Example 4.1.8. *Consider the view v_1 from Example 4.1.2 and the insertion u_4, adding the following XML fragment:*

$$xml_4 = \langle b\rangle\langle c/\rangle\langle/b\rangle$$

as a child of a node a, whose ID is a.id. This ID encodes the labels of all the nodes on the path from the root to a. Assume that a.id is inspected and no ancestor labelled b is found above the a node. Then, the new (inserted) c node has only one b ancestor, namely the inserted (new) b node. Thus, the term $R_aR_b\Delta^+_c$ is empty. Therefore, the only term to compute to update v_1 after inserting xml_4 is $R_a\Delta^+_b\Delta^+_c$.

This generalises as follows:

Proposition 4.1.9. *Let u be an insertion adding children to the nodes $p_1, p_2, \ldots, p_k$ in d. Let v be a view, and n_1, n_2 be v nodes such that n_1 is an ancestor of n_2 in v. If for each $i = 1, 2, \ldots, k$, p_i is not labelled n_1 and has no ancestor labelled n_1, then all union terms containing $R_{n_1}\Delta^+_{n_2}$ are empty.*

Another pruning method was developed for when schemas are available. However, since they were not present in this work it was never implemented. It is detailed in Appendix A.

Algorithms

The first update propagation algorithm considers the scenario when an insertion to the XML document either leads to new tuples being added to the view, or does not affect the view. When the view is affected, insertions result in entering new tuples in the view or increasing the derivation count of tuples in the view.

The insertion propagation method is outlined in Algorithm 1, Propagate Insert by New Tuples (PINT). The computation of the Δ^+ relations is performed first and the algorithm for this will be detailed shortly. However, the core of the complexity in Algorithm 1 lies in line 6: the computation of the remaining union terms.

ALGORITHM 1: Propagate Insert by New Tuples **(PINT)**

```
Input: insert update $u$, view $v$, lattice $l$, document $d$
Output: updated view $v'$ to reflect $u$; updated lattice $l'$ to reflect $u$; updated document $d'$
        to reflect $u$
1 Compute the $\Delta^+$ relations corresponding to $u$ (call Algorithm CD+($u$, $d$))
2 Develop the $2^\Delta - 1$ union terms to be added to $v$ in case of insertions, and prune them
  based on XQuery Update semantics (Proposition 4.1.4)
3 Further prune terms based on the $\Delta^+$ relations (Propositions 4.1.7 and 4.1.9)
4 Evaluate the remaining terms and add their results to $v$ (call Algorithm ET-INS($u$, $v$, $\Delta^+$,
  $l$)
5 Update the snowcaps from the bottom up in the lattice
6 Update document
```

ALGORITHM 2: Compute Δ^+ relations (**CD+**)

```
Input: insert update $u$, document $d$
Output: $\Delta^+$ relations
1 $(n_1, t_1), \ldots, (n_k, t_k) \leftarrow$ compute-pul($u, d$)
2 for $n \in n_k$ labelled $l$ do
3     let $p_l$ be the pattern $//l_{attributes}$
4     $\Delta_l^+ \leftarrow \bigcup_{1 \leq i \leq k}$(extr-pattern($n_{id}$, $p_l$, $t_i$))
```

Computing Δ^+ relations Given an insertion u on the document d and the view v, Algorithm 2 **(CD+)** computes the Δ^+ relations. It relies on the function **compute-pul** to compute the update list and **extr-pattern** to extract the Δ^+ relations from the pending updates. These functions are described in Section 3.4.

Evaluate Insertion Algorithm 3 (**ET-INS**) outlines the evaluation of the terms, that survived pruning, resulting from an insertion. Efficient evaluation techniques within the XML query engine are utilised by means of structural joins within the first for loop.

ALGORITHM 3: Evaluate terms resulting from insert (**ET-INS**)

```
Input: insert update $u$, view $v$, delta relations $\Delta^+$, lattice $l$
Output: updated view $v'$ to reflect $u$; updated lattice $l'$ to reflect $u$
1 $\Delta_v^+ \leftarrow \emptyset$ (tuples to be possibly added to $v$)
2 foreach term $t$ surviving pruning do
3     Evaluate $t_{\Delta^+}$ by structural joins over the $\Delta^+$ relations
4     Add to $\Delta_v^+$ the result of joining $t_R$ (snowcap materialised in the lattice) and $t_{\Delta^+}$
5 foreach tuple $t_\Delta \in \Delta_v^+$ do
6     if $t_\Delta \in v$ then
7         increase the derivation count of $t_\Delta$
8     else
9         add $t_\Delta$ to $v$ with a derivation count of 1
```

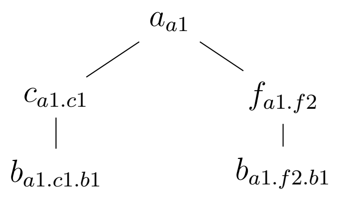


Figure 4.3: Sample XML Document

4.1.2 Deletions

This section will discuss the propagation of deletions. The following example illustrates a simple deletion scenario.

Example 4.1.10. *Consider the view* $//a_{ID}//b_{ID}$ *and the XML document* d *shown in Figure 4.3, where the ID of each node is shown as a subscript. Consider an update* u_1 *deleting* $//c//b$*. When evaluated on the document, this targets the node whose ID is* $a1.c1.b1$*. Therefore, this node must be deleted from the document and the tuple* $(a1, a1.c1.b1)$ *must be deleted from the view.*

The remainder of this section is structured as follows: first the basis of the algebraic delete approach is outlined, namely algebraic deletion expressions. Then term pruning criteria are presented to simplify the deletion expression based on the semantics of XML and XML updates, and the tuples to be removed from the view by the deletion. Finally, the deletion propagation algorithms are provided.

Deletion propagation expression

In the context of deletions, for a view node label a, Δ_a^- is defined as the ordered collection of tuples of the form $(n.id)$ for all nodes n to be deleted from the document, which are labelled a. As a result of a deletion, a view of the form

$$R_{a_1} \bowtie R_{a_2} \bowtie \ldots R_{a_k}$$

needs to be transformed, after the deletion of the elements in $\Delta_{a_1}^-$, $\Delta_{a_2}^-, \ldots, \Delta_{a_k}^-$, into:

$$(R_{a_1} \setminus \Delta_{a_1}^-) \bowtie (R_{a_2} \setminus \Delta_{a_2}^-) \bowtie \ldots \bowtie (R_{a_k} \setminus \Delta_{a_k}^-)$$

This can be expanded based on the known properties of the relational $\bowtie$ (join) and $\setminus$ (set difference) operators. By eliminating the parentheses the following expression can be reached.

$$R_{a_1}R_{a_2}\ldots R_{a_k} \setminus \Delta_{a_1}^- R_{a_2}R_{a_3}\ldots R_{a_k} \setminus R_{a_1}\Delta_{a_2}^- R_3 \ldots R_k \setminus \ldots \setminus R_{a_1}R_{a_2}R_{a_3}\ldots \Delta_{a_k}^-$$
$$\cup\, \Delta_{a_1}^-\Delta_{a_2}^- R_3 \ldots R_{a_k} \,\cup\, R_{a_1}\Delta_{a_2}^-\Delta_{a_3}^- \ldots R_{a_k} \,\ldots\, \Delta_{a_1}^-\Delta_{a_2}^-\Delta_{a_3}^- \ldots \Delta_{a_k}^-$$

The expanded expression above has 2^{Δ} terms. The first term is the view definition v, whereas all the others contain delta relations. In this work the expression is defined as the (expanded) deletion expression of v. Propagating a deletion update u to the view v requires evaluating this expression (minus the view definition) and removing the returned tuples from the view.

Term Pruning

Similarly to the insertion case, to simplify the processing of updates, criteria for term pruning are identified. The first pruning methods rely on the update semantics of XQuery Update [15], whereas the final method relies on the Dynamic Dewey IDs [3]. The first pruning method is presented in the following proposition.

Proposition 4.1.11. *Let v be a view and n_1, n_2 be two nodes in v, such that node n_2 is a / or // child of n_1 in v. Let t be a term in a deletion expression, such that $\Delta^-_{n_1} R_{n_2}$ is a subexpression of t. Then, t has no results (see Proposition 4.1.4).*

This Proposition follows from the semantics of XQuery Update for deletions, as if a node is removed then so are all its descendants. So, if n_1 is deleted then so are its descendants, including n_2. This is a counterpart to Proposition 4.1.4 for insertions.

The second pruning method also relies on XQuery Update semantics and is presented below. This pruning method is specific to deletions.

Proposition 4.1.12. *Let v be a view and t be a term in its deletion expression, including $k > 0$ relations of the form Δ^-: (i) If k is odd, the operand before t is $\setminus$, that is, the tuples of t must be removed from v, whereas if k is even, the operand before t is $\cup$, that is, the deletion expression requires adding these tuples to reflect the deletion; (ii) if k is even, t can be ignored in the deletion expression.*

Proof. Claim (i) follows directly from when joins are distributed over the set difference in the deletion expression. For what concerns claim (ii), an even and positive k must be at least 2. This ensures that in the deletion expression there exists k terms $t_1, t_2, \ldots, t_k$ such that for each $1 \leq i \leq k$, t_i is identical to t except that t_i contains a symbol of the form R_a, where t contains Δ^-_a, for some a. Observe that each of the terms $t_1, t_2, \ldots, t_k$ have $k-1$ Δ^- relations, and $k-1$ is odd, therefore, by claim (i), they all represent tuples to remove from v. Thus, each tuple which t would attempt to add as a result of the deletion u, is deleted at least k times with $k \geq 2$, whereas t would attempt to add it only once. In other words the positive ($\cup$) terms do not actually need to be calculated in the deletion expression at all, since that data will be removed "more times than t can add it". Thus, such positive terms can simply be ignored. $\square$

Example 4.1.13. *Consider a view v_2 defined as $//a_{ID}[//c_{ID}]//b_{ID}$. Elementary development of the deletion expression shows that the delete expression of v (prior to any term pruning) is:*

$$R_aR_bR_c \setminus \Delta^-_a R_bR_c \setminus R_a\Delta^-_b R_c \setminus R_aR_b\Delta^-_c \;\cup\; \underline{\Delta^-_a\Delta^-_b R_c} \;\cup\; \underline{\Delta^-_a R_b\Delta^-_c} \;\cup\; \underline{R_a\Delta^-_b\Delta^-_c} \setminus \Delta^-_a\Delta^-_b\Delta^-_c$$

where the underlined terms are those prefixed with $\cup$, whose tuples should be "added" to reflect a deletion. Let these terms be considered one by one:

- *$\Delta_a^- \Delta_b^- R_c$ tuples are deleted first, by the term $\Delta_a^- R_b R_c$, and second, by the term $R_a \Delta_b^- R_c$;*
- *$\Delta_a^- R_b \Delta_c^-$ tuples are deleted first, by the term $\Delta_a^- R_b R_c$, and second, by the term $R_a R_b \Delta_c^-$;*
- *$R_a \Delta_b^- \Delta_c^-$ tuples are deleted first, by the term $R_a R_b \Delta_c^-$ and second, by the term $R_a \Delta_b^- R_c$.*

Proposition 4.1.12 allows reducing this expression to:

$$R_a R_b R_c \setminus \Delta_a^- \Delta_b^- \Delta_c^- \setminus \Delta_a^- R_b R_c \setminus R_a \Delta_b^- R_c \setminus R_a R_b \Delta_c^-$$

In this expression, data is only removed *from $R_a R_b R_c$, which rejoins the intuition that deletes should not add tuples to the (conjunctive, monotone) views.*

The following example illustrates the cumulated impact of the Propositions 4.1.11 and 4.1.12.

Example 4.1.14. *Consider the view v_2 from Example 4.1.13 and the XML document d shown on the left in Figure 4.4. The tuples of the view v_2 evaluated on d appear in the same figure on the right. Consider the update u_2 deleting $//a/f/c$ from d. This amounts to deleting the d subtree rooted in the node identified by $a1.f2.c1$. The full deletion expression of v_2 under the update u_2 (as in Example 4.1.13) is:*

$$R_a R_b R_c \setminus \Delta_a^- R_b R_c \setminus R_a R_b \Delta_c^- \setminus R_a \Delta_b^- R_c \cup R_a \Delta_b^- \Delta_c^- \cup \Delta_a^- R_b \Delta_c^- \cup \Delta_a^- \Delta_b^- R_c \setminus \Delta_a^- \Delta_c^- \Delta_b^-$$

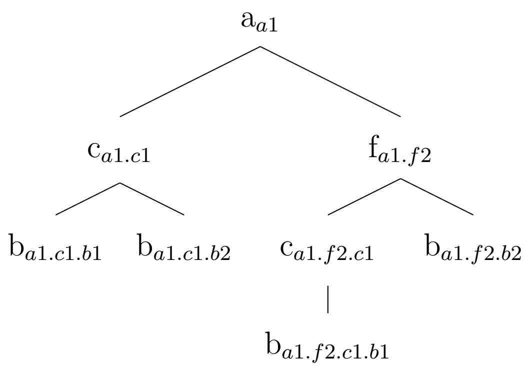
Proposition 4.1.11 eliminates the second, sixth and seventh join terms from this expression, since they are guaranteed to be empty. Thus, the deletion expression is reduced to:

$$R_a R_b R_c \setminus R_a R_b \Delta_c^- \setminus R_a \Delta_b^- R_c \cup R_a \Delta_b^- \Delta_c^- \setminus \Delta_a^- \Delta_c^- \Delta_b^-$$

Proposition 4.1.12 eliminates the positive term, and the deletion expression becomes:

$$R_a R_b R_c \setminus R_a R_b \Delta_c^- \setminus R_a \Delta_b^- R_c \setminus \Delta_a^- \Delta_c^- \Delta_b^-$$

The computation of the pending update list leads to discovering that $\Delta_a^- = \emptyset$, which further simplifies the deletion expression to:



num.	a_{ID}	c_{ID}	b_{ID}
1	$a1$	$a1.c1$	$a1.c1.b1$
2	$a1$	$a1.c1$	$a1.c1.b2$
3	$a1$	$a1.c1$	$a1.f2.c1.b1$
4	$a1$	$a1.c1$	$a1.f2.b2$
5	$a1$	$a1.f2.c1$	$a1.c1.b1$
6	$a1$	$a1.f2.c1$	$a1.c1.b2$
7	$a1$	$a1.f2.c1$	$a1.f2.c1.b1$
8	$a1$	$a1.f2.c1$	$a1.f2.b2$

Figure 4.4: Sample XML document for Example 4.1.14, and view content on this document.

$$R_aR_bR_c \setminus R_aR_b\Delta_c^- \setminus R_a\Delta_b^-R_c$$

The pending update list also defines that $\Delta_b^- = \{(a1.f2.c1.b1)\}$ *and* $\Delta_c^- = \{(a1.f2.c1)\}$*. Thus,* $R_aR_b\Delta_c^-$ *contains the tuples* $5, 6, 7$ *and* 8 *from the view, while* $R_a\Delta_b^-R_c$ *consists of the tuples* 3 *and* 7 *from the view. The update* u_2 *thus reduces* v *to its tuples* $1, 2$ *and* 4*.*

ID-driven pruning can also be performed for the deletion case. Consider the following example:

Example 4.1.15. *Let* v *be the view* $//c_{ID}//b_{ID}$ *and* d *be the XML document shown in Figure 4.3. Consider an update* u_2 *deleting* $//f$*, which in* d *targets the node identified by* $a1.f2$*. As a side-effect of the deletion, the node identified by* $a1.f2.b1$ *is also removed, thus* $\Delta_b^- = \{(a1.f2.b1)\}$*. From the identifier of the single* Δ_b^- *node, it can be seen that this node does not have a* c *ancestor. Therefore, the deletion expression term* $R_c\Delta_b^-$ *is empty.*

This example generalises into:

Proposition 4.1.16. *Let* v *be a view and* n_1, n_2 *be two nodes in* v*, such that* n_2 *is a / or // child of* n_1 *in* v*. Consider an update* u *and let* $\Delta_{n_2}^-$ *be the delta relations corresponding to the label of* n_2*. If for every node* $m \in \Delta_{n_2}^-$*, the ID shows that* m *has no ancestor labelled* n_1*, then all terms in the* v *deletion expression containing* $R_{n_1}\Delta_{n_2}^-$ *are empty.*

This is a counterpart to Proposition 4.1.9.

Deletion propagation algorithms

The general algorithm Propagate Delete by Deleting Tuples (**PDDT**), which propagates a deletion on the XML source document to the view, is outlined in Algorithm 4. These deletions result in removing tuples from the view, if the view is affected. Algorithm **CD-** is not included as it is similar to **CD+**. The only difference is that val and cont are not calculated as only the IDs are required for deletions.

ALGORITHM 4: Propagate Delete by Deleting Tuples **(PDDT)**

```
Input: deletion update $u$, view $v$, lattice $l$, document $d$
Output: updated view $v'$ to reflect $u$, updated lattice $l'$ to reflect $u$, updated document $d'$
        to reflect $u$
1  Compute the $\Delta^-$ relations corresponding to $u$ (call Algorithm CD-($u$, $d$))
2  Expand the deletion expression to $2^\Delta - 1$ terms and prune them based on the data
   (Propositions 4.1.11 and 4.1.12) and then based on the $\Delta^-$ relations (Proposition 4.1.16)
3  $\Delta_v^- \leftarrow \emptyset$ (tuples to be possibly deleted from $v$)
4  foreach term $t$ surviving pruning do
5      Evaluate $t_{\Delta^-}$ by structural joins over the $\Delta^-$ relations
6      Add to $\Delta_v^-$ the result of joining $t_R$ (snowcap materialised in the lattice) and $t_{\Delta^-}$
7  foreach tuple $t_v \in$ view $v$ do
8      if $t_v \in \Delta_v^-$ then
9          remove $t_v$ from $v$
10     else
11         if $t_v$ stores ID, val or cont for a node $n$ (i) appearing in some $\Delta^-$ relation or (ii)
           having an ancestor in some $\Delta^-$ relation then
12             decrease $t_v$ derivation count
13             if $t_v$'s derivation count becomes 0 then
14                 remove $t_v$ from $v$
15 Update the snowcaps from the bottom up in the lattice
16 Update document
```

Algorithm **PDDT** handles tuple deletions and decreasing the derivation count of a tuple which might result in removing it from the view. Both cases are illustrated by the following example.

Example 4.1.17. *To illustrate the first case, consider the view* $//a_{ID}//b$*, the document* d *in Figure 4.3, and an update deleting* $//c//b$*. The view contains two tuples corresponding to node* b*. The deletion removes the* b *node identified by* $a1.c1.b1$*, which belongs to* Δ_v^-*. This case is the same as the one illustrated by Example 4.1.10. Therefore, no check of the derivation count is necessary as* t_v *matches exactly the node in* Δ_v^-*.*

To illustrate the second case, consider the view $//a_{ID}[//b]$*, the document* d *in Figure 4.3, and an update deleting* $//c//b$*. The view contains a single tuple corresponding to node* a*. The tuple has a derivation count of* 2 *due to the two* b *nodes matching the existential view branch. The deletion removes the* b *node identified by* $a1.c1.b1$*, but this still leaves a* b *descendant to the* a *node. Therefore, the update decreases the derivation count of the view tuple* $(a1)$ *by one unit, setting it to* 1*.*

Now consider a second update, deleting $//f//b$*. This will lead to removing the node* $(a1.f2.b1)$*, reducing the derivation count of the corresponding* v *tuple to* 0 *and thus removing the tuple from the view.*

4.1.3 Modifications

A case not considered yet is when updates result in the modification of existing tuples. This section will discuss the propagation of modifications for insertions and deletions.

Modifications Resulting from Insertions

Existing tuples of a view v may have to be modified as a result of an insertion. This is a consequence of an insertion modifying the value or content of an XML node n, whose value (respectively, content) is stored in v. This modification can be the result of an insertion on n or one of its descendants.

Example 4.1.18. *Consider the view* $/a_{ID}/b_{ID}//c_{ID,Cont}$ *and an insertion* u *adding the XML fragment:*

$$\langle extra \rangle \; some \; value \; \langle /extra \rangle$$

into //d//c. In this case, no Δ^+ relation affects the view, thus no new tuples need to be added. However, the insertion u may lead to modifying some of the c.cont values stored by the view, if the intersection of /a/b//c and //d//c is not empty. Assuming this intersection is not empty the XML fragment can be inserted after the last child of the c in c.cont.

If the cont was associated with the a or the b elements then the update would be handled in the exact same way. Additionally, if the same scenario as shown here occurs with a val it will be handled similarly. IDs don't need to be handled in the modification case as they are immutable.

Propagate Insert by Modifying Tuples (**PIMT**), shown in Algorithm 5, handles the problem of modifications. It considers all XML nodes for which the view stores content and/or value. These nodes are then checked to determine if they are affected by the update. If they are, then their value and/or content must be updated to reflect this.

ALGORITHM 5: Propagate Insert by Modifying Tuples **(PIMT)**

```
Input: insert update $u$, view $v$, document $d$
Output: updated view $v'$ to reflect $u$
1 $ut = [(n_1, t_1), \ldots, (n_k, t_k)] \leftarrow$ compute-pul($u$, $d$);
2 $cvn \leftarrow \{n \in v$, $n$ annotated with $cont$ or $val\}$;
3 foreach tuple $t \in v$ do
4     foreach tuple $(n_i, t_i) \in ut$ and $t.n \in cvn$ do
5         if $t.n = n_i$ or $t.n \prec\!\!\prec n_i$ then
6             Update $t.n.cont$ (respectively, $t.n.val$) to reflect the insertion of $t_i$
```

PIMT first determines the pending update list. It then searches for all the content or value annotated view nodes and stores them in the node set *cvn*. The algorithm then checks, for each view tuple t, whether and how each of the t attributes corresponding to the content or value of a *cvn* node must change. It can be checked that the *cont* and/or *val* attribute in tuple t is the same as, or an ancestor of the node n_i, by inspecting its ID. If any of these

$$\begin{tikzcd}
d \arrow[dr] & v \arrow[d] & u \arrow[dl] \\
\boxed{\text{PIMT}} & \boxed{\text{PINT}} \arrow[l] \arrow[r] \arrow[d] & \boxed{\text{ET-INS}} \arrow[d] \\
& v' & \boxed{\text{CD+}}
\end{tikzcd}$$

Figure 4.5: Insert propagation outline.

$$\begin{tikzcd}
d \arrow[dr] & v \arrow[d] & u \arrow[dl] \\
\boxed{\text{PDMT}} \arrow[r] & \boxed{\text{PDDT/PDMT}} \arrow[r] \arrow[d] & \text{CD-} \\
& v' &
\end{tikzcd}$$

Figure 4.6: Delete propagation outline.

cases are true, then the insertion of t_i has to be propagated to the t attribute corresponding to $n.cont$ (respectively, $n.val$).

If cvn is empty, insertions cannot modify view tuples (but only add to the view). If cvn is of size 1 (val or $cont$ is only stored by one node of v) , **PIMT** can be implemented by a single efficient structural join (extended to check ancestor-descendant or equality relationships) between v and the pending update list. The $cont$ and/or val attributes must be changed for the view tuples that join with the pending update list. If cvn contains multiple nodes, then a nested loops join is required as **PIMT** must compare several IDs from each view tuple t with the pending update list.

In practice it is not known in advance whether an insertion adds and/or modifies tuples, therefore both Algorithms **PINT** and **PIMT** have to be run. To improve efficiency a combined algorithm **PINT/MT** is used to avoid repeated computation, i.e., generating the pending update list. **PINT/MT** can be seen in Algorithm 6. The insert propagation method's main steps are shown in Figure 4.5. A description of the symbols used in the figure are shown in Figure 4.7.

Proposition 4.1.19 (Complexity of Algorithms 1 and 5)**.** *Let v be a view of k nodes and u be an insertion resulting in a pending update list PUL. Let R be the largest relations among the leaf nodes in v's lattice. The worst case complexity of Algorithm 1* ***(PINT)*** *is $O(2^k \times k \times max(|PUL|, |R|))$. Moreover, the complexity of Algorithm 5* ***(PIMT)*** *is $O(|v| \times |PUL| \times |cvn|)$, where cvn is the set of content-returning nodes of v.*

Algorithm 3's **(ET-INS)** (called by Algorithm 1) term evaluation dominates the complexity of Algorithm 1 **(PINT)**. The upper bound is obtained by assuming, pessimistically, that no term is pruned; also, it is assumed, pessimistically, that no intermediary lattice

ALGORITHM 6: Propagate Insert by New/Modifying Tuples **(PINT/MT)**

```
Input: insertion update $u$, view $v$, lattice $l$, document $d$
Output: updated view $v'$ to reflect $u$, updated lattice $l'$ to reflect $u$, updated document $d'$
        to reflect $u$
1  Compute the $\Delta^+$ relations corresponding to $u$ (call Algorithm CD+($u$, $d$))
2  Develop the $2^\Delta - 1$ union terms to be added to $v$ in case of insertions, and prune them
   based on XQuery Update semantics (Proposition 4.1.4)
3  Further prune terms based on the $\Delta^+$ relations (Propositions 4.1.7 and 4.1.9)
4  $\Delta_v^+ \leftarrow \emptyset$ (tuples to be possibly added to $v$)
5  $cvn \leftarrow \{n \in v$, $n$ annotated with $cont$ or $val\}$;
6  Evaluate the remaining terms and add their results to $v$ (call Algorithm ET-INS($u$, $v$, $\Delta^+$,
   $l$)
7  if $\Delta_v^+ \neq \emptyset$ then
8      foreach tuple $t_\Delta \in \Delta_v^+$ do
9          if $t_\Delta \in v$ then
10             increase the derivation count of $t_v$
11         else
12             add $t_\Delta$ to $v$ with a derivation count of 1
13 if $cvn \neq \emptyset$ then
14     call Algorithm PIMT($u$, $v$, $d$)
15 If needed, update lattice
16 Update document
```

node is materialised, thus joins have to be recomputed from the lattice leaves and the Δ^+ relations. The computation of the Δ^+ relations is assumed to be part of the update process and is thus not considered in the complexity of PINT. Additonally, the size of a Δ^+ relation is bound by $|PUL|$. The inputs to the k-way join of each term is either a Δ^+ relation or a leaf in the lattice (whose size is bound by $|R|$). Finally, it is assumed that evaluating a term in time proportional to the cumulated size of its inputs is possible using efficient join algorithms such as holistic twig join.

Symbol	**Description**
d	Document
v	View
v'	Updated View
u	Update
PINT	Propagate Insert by New Tuples
ET-INS	Evaluate Terms Resulting from Insert
CD+	Compute Δ^+ Relations
CD-	Compute Δ^- Relations
PIMT	Propagate Insert by Modifying Tuples
PDDT	Propagate Delete by Deleting Tuples
PDMT	Propagate Delete by Modifying Tuples

Figure 4.7: Acronyms used in Figures 4.5 and 4.6.

Modifications Resulting from Deletions

Similarly to insertions, deletions can also result in updates that modify view tuples, by altering some *val* or *cont* attribute that the view stores. Such modifications may occur regardless of whether the update deletes tuples and/or modifies derivation counts. However, unlike insertions, this algorithm only has to be run if the query is not linear (i.e, it contains predicates). This is because a linear query will delete all nodes on its path which will empty the view. Therefore, performing any modifications is unnecessary in this case.

An algorithm has been designed, Propagate Delete by Modifying Tuples (**PDMT**), which can be seen as symmetric to Algorithm 5 (focusing on tuple modifications due to insertions). This can be seen in Algorithm 7. Modifications, deletions and derivation count updates can all be a consequence of the same update. Therefore, a general algorithm for handling all these possibilities is provided and used. Propagate Delete by Deleting/Modifying Tuples **(PDD/MT)** is outlined in Algorithm 8. This combines the main steps of **PDDT** and **PDMT** to avoid repeated computation, i.e, the computation of the pending update list. Figure 4.6 outlines the main steps of the delete propagation method - Figure 4.7 defines the acronyms.

ALGORITHM 7: Propagate Delete by Modifying Tuples **(PDMT)**

```
Input: deletion update $u$, view $v$, document $d$
Output: updated view $v'$ to reflect $u$
1 $ut = [n_1, \ldots, n_k] \leftarrow$ compute-pul($u$, $d$);
2 $cvn \leftarrow \{n \in v$, $n$ annotated with $cont$ or $val\}$;
3 foreach tuple $t \in v$ do
4     foreach tuple $(n_i, t_i) \in ut$ and $t.n \in cvn$ do
5         if $t.n \lll n_i$ then
6             Update $t.n.cont$ (respectively, $t.n.val$) to reflect the deletion of $t_i$
```

Proposition 4.1.20 (Complexity of Algorithms 4 and 8)**.** *Let v be a view of k nodes, $|v|$ denote the number of tuples in v, and u be a deletion resulting in the pending update list PUL. Let R be the largest relation among the leaf nodes in v's lattice. The complexity of Algorithm 4 (PDDT) is $O(2^k \times k \times max(|PUL|, |R|) + |v| \times s(\Delta^-))$, where $s(\Delta^-)$ is the cost to search for a tuple in one of the Δ^- relations, containing the ID of a node appearing in the view, or of an ancestor of such a node (check performed in Algorithm 4). As for **PINT** and **PIMT**, $2^k \times k \times max(|PUL|, |R|)$ is a bound for the cost of evaluating all the deletion terms which have survived pruning. The second term $|v| \times s(\Delta^-)$ corresponds to the PDDT loop over v tuples. The complexity of Algorithm 8 **(PDDT/MT)** is $O(2^k \times k \times max(|PUL|, |R|) + |v|^2 \times s(\Delta^-))$.*

The overall purpose of the work described in this thesis is to develop incremental methods for solving the view maintenance and view update problems using statement-level updates and dynamic reasoning by means of pruning rules. Section 4.1 has explained the methods involved in incremental view maintenance. In the next section (4.2), attention is turned to the view update problem.

ALGORITHM 8: Propagate Delete by Deleting/Modifying Tuples **(PDDT/MT)**

```
Input: deletion update $u$, view $v$, lattice $l$, document $d$
Output: updated view $v'$ to reflect $u$, updated lattice $l'$ to reflect $u$, updated document $d'$
        to reflect $u$
1  Compute the $\Delta^-$ relations corresponding to $u$ (call Algorithm CD-($u$, $d$))
2  Develop the $2^\Delta - 1$ set difference terms to be deleted from $v$ in case of deletions and prune
   terms based on the data and $\Delta^-$ relations (Propositions 4.1.11, 4.1.12 and 4.1.16)
3  $\Delta_v^- \leftarrow \emptyset$ (tuples to be possibly deleted from $v$)
4  $cvn \leftarrow \{n \in v$, $n$ annotated with $cont$ or $val\}$;
5  foreach term t surviving pruning do
6      Evaluate $t_{\Delta^-}$ by structural joins over the $\Delta^-$ relations
7      Add to $\Delta_v^-$ the result of joining $t_R$ (snowcap materialised in the lattice) and $t_{\Delta^-}$
8  if $\Delta_v^- \neq \emptyset$ then
9      foreach tuple $t_v \in$ view $v$ do
10         if $t_v \in \Delta_v^-$ then
11             decrease $t_v$ derivation count
12         if $t_v$'s derivation count becomes 0 then
13             remove $t_v$ from $v$
14             if $t_v.n \in cvn$ then
15                 remove $t_v.n$ from $cvn$
16 if $cvn \neq \emptyset$ then
17     call Algorithm PDMT($u$, $v$, $d$)
18 If needed, update lattice
19 Update document
```

4.2 View Update

This section will describe the view update problem in the context of the XML data model and the method developed to handle it. View update can be described as follows: Let d be a document and vu an insert on a view v. View update involves transforming d into d' so as to reflect the effect of vu on v. This can be seen in Figure 4.8. A generalisation of this problem is translating update programs: given a view definition v and a view update program f_v, provide a source update program f, s.t., for all documents d, $v(f,(d)) = f_v(v(d))$. That is, applying f on the source document d and then computing the view gives the same result as applying f_v on the view of d. The remainder of this section will describe algorithms for performing vu on v and transforming d into d' to reflect this.

$$\begin{tikzcd}
v(d) \arrow[r, "\text{view update}"] & v(d') \\
d \arrow[u, dotted, "\text{view evaluation}"] \arrow[r, dashed, "\text{update propagation}"] & d' \arrow[u, dotted, "\text{view evaluation}"']
\end{tikzcd}$$

Figure 4.8: View Update.

4.2.1 View Maintenance applied to View Update

A subset of updates can be handled by the view maintenance method explained in Section 4.1. The only modification required is where the Δs originate from. The problem lies in performing the source update. An update can be handled if the view contains no predicates.

Example 4.2.1. *Consider the view $v_1 = //a//b_{id}[c_{id}]//d_{id}//e_{id}$ and the update vu_1 that inserts in v_1 at $//a//b$ the following XML fragment:*

$$xml_1 = \langle d\rangle\langle e/\rangle\langle/d\rangle$$

Then if the same source insertion method from view maintenance is used, it will target all a nodes with a b descendant.

However, if the view was $v_1 = //a//b_{id}//d_{id}//e_{id}$, then the source insertion method for view maintenance would work as there are no predicates.

The same problem would arise if the update was a deletion. This is generalised by the following proposition.

Proposition 4.2.2. *If the view definition contains no predicates, then the source update is supported by the view maintenance method. Therefore, updates can be handled if there are no predicates contained in the view.*

The handling of source updates is discussed in Section 4.2.3.

4.2.2 IDs

Assigning IDs to tuples to be added to a view is more complex within the view update problem. An ID has to be assigned to each new tuple which is unique and correct for the insertion position. The ID of the target node is required for this purpose.

Due to the developed extension of Dynamic Dewey IDs [3], which store the label paths as well as the ID paths, the ID can be determined. Therefore, the view can be queried to discover the ID of the target node(s). The ID of the new tuple is assigned by adding the child number at the end of the target node's ID. The IDs of other nodes, should a tree be inserted, are assigned similarly, e.g., for the node 1.1 with no children an inserted node would have the ID 1.1.1. In order to determine the child number, the current number of children the target node has must be queried.

For deletions the handling of IDs is a lot simpler. Only the IDs of the tuples to be deleted are required which can be taken directly from the view.

4.2.3 Insertions

The rest of this section is structured as follows: first the general approach is outlined by means of examples, then a set of pruning methods is introduced which reduces the computations required for view update. Finally, the algorithms are introduced for handling insertions.

Method

The approach is similar to the view maintenance method. The difference lies in the implementation. For clarity between view maintenance (Section 4.1) and view update, delta relations will be represented by $v\Delta$ in the view update case, as $v\Delta$s come from the view and not the document. For further details on the view maintenance approach, see Section 4.1.1.

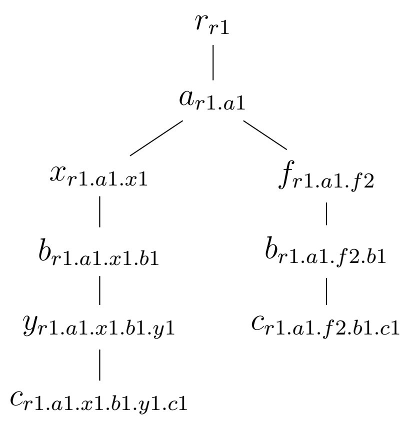


FIGURE 4.9: XML Document

The following example shows how view update is handled where the insertion point in the view is an intermediate node.

Example 4.2.3. *Consider the view $v_2 = //a_{id}//b_{id}//c_{id}$ and the view update vu_2 which inserts in v_2 at $//a$ the following XML fragment:*

$$xml_2 = \langle b \rangle \langle c/ \rangle \langle /b \rangle$$

The source document d can be seen in Figure 4.9. As there is only one a in the source there will only be one b and one c $v\Delta$ tuple. Let b_1 and c_1 be the XML elements inserted in v_2 by vu_2. The $v\Delta^+$ relations corresponding to vu_2 are:

$v\Delta_b^+$	$v\Delta_c^+$
$(b_1.id)$	$(c_1.id)$

The IDs are assigned based on the approach in Section 4.2.2. From a source query it can be seen that a already has two existing children. Therefore, the ID will be calculated based on a's ID and the number of children, 2. Alternatively, should cont be stored for the a this could be queried to determine the number of existing children. The val and cont attributes are derived directly from the XML fragment, should they be stored. However, they are not required for v_2.

After vu_2 is applied, v_2 should become:

$$\begin{array}{ll}
v_2 & \cup\,(R_a \bowtie_{a \prec\!\!\prec b} R_b \bowtie_{b \prec\!\!\prec c} v\Delta_c^+) \cup \\
(R_a \bowtie_{a \prec\!\!\prec b} v\Delta_b^+ \bowtie_{b \prec\!\!\prec c} R_c) & \cup\,(R_a \bowtie_{a \prec\!\!\prec b} v\Delta_b^+ \bowtie_{b \prec\!\!\prec c} v\Delta_c^+) \cup \\
(v\Delta_a^+ \bowtie_{a \prec\!\!\prec b} R_b \bowtie_{b \prec\!\!\prec c} R_c) & \cup\,(v\Delta_a^+ \bowtie_{a \prec\!\!\prec b} R_b \bowtie_{b \prec\!\!\prec c} v\Delta_c^+) \cup \\
(v\Delta_a^+ \bowtie_{a \prec\!\!\prec b} v\Delta_b^+ \bowtie_{b \prec\!\!\prec c} R_c) & \cup\,(v\Delta_a^+ \bowtie_{a \prec\!\!\prec b} v\Delta_b^+ \bowtie_{b \prec\!\!\prec c} v\Delta_c^+)
\end{array}$$

vu_2 must then be translated to a source update and performed on d.

As the label path is stored within each ID, the insertion point of the new XML fragment within the document can be determined. For this XML fragment the IDs will be:
b_1: r1.a1.b3
c_1: r1.a1.b3.c1

The IDs of all the tuples are not required to insert in the source, only the root of the subtree that is being inserted. In the case of ID r1.a1.b3, the target node where the insertion has to occur, r1.a1, is known. Therefore, the subtree is inserted as descendants of this node.

Therefore, d will become:

$$\begin{tikzcd}
 & r_{r1} \arrow[d, dash] & \\
 & a_{r1.a1} \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
x_{r1.a1.x1} \arrow[d, dash] & f_{r1.a1.f2} \arrow[d, dash] & b_{r1.a1.b3} \arrow[d, dash] \\
b_{r1.a1.x1.b1} \arrow[d, dash] & b_{r1.a1.f2.b1} \arrow[d, dash] & c_{r1.a1.b3.c1} \\
y_{r1.a1.x1.b1.y1} \arrow[d, dash] & c_{r1.a1.f2.b1.c1} & \\
c_{r1.a1.x1.b1.y1.c1} & &
\end{tikzcd}$$

The following is another example which shows a similar update to Example 4.2.3, where the insertion point in the view has changed to a leaf.

Example 4.2.4. *Consider the view v_2 as defined in Example 4.2.3, and the view update vu_3 that inserts in v_2 at //a//b the following XML fragment:*

$$xml_3 = \langle c/\rangle$$

The source document d can be seen in Figure 4.9. As two b nodes have a ancestors this will mean there are two c tuples. Let c_1 and c_2 be the XML elements inserted in v_2 by vu_3. The $v\Delta^+$ relations corresponding to vu_3 are:

$v\Delta^+_c$
$(c_1.id)$
$(c_2.id)$

The IDs are assigned based on the approach in Section 4.2.2. There are two bs in the view, as can be seen from d. Therefore, the XML fragment will be inserted as children of both these nodes. From a source query it can be seen that they both have one existing child each. Therefore, the IDs for each XML fragment will be calculated based on the respective ID of b and the number of children, 1.

After vu_3 is applied, v_2 should become:

$$\begin{array}{ll} v_2 & \cup\,(R_a \bowtie_{a\prec\!\prec b} R_b \bowtie_{b\prec\!\prec c} \Delta^+_c)\,\cup \\ (R_a \bowtie_{a\prec\!\prec b} \Delta^+_b \bowtie_{b\prec\!\prec c} R_c) & \cup\,(R_a \bowtie_{a\prec\!\prec b} \Delta^+_b \bowtie_{b\prec\!\prec c} \Delta^+_c)\,\cup \\ (\Delta^+_a \bowtie_{a\prec\!\prec b} R_b \bowtie_{b\prec\!\prec c} R_c) & \cup\,(\Delta^+_a \bowtie_{a\prec\!\prec b} R_b \bowtie_{b\prec\!\prec c} \Delta^+_c)\,\cup \\ (\Delta^+_a \bowtie_{a\prec\!\prec b} \Delta^+_b \bowtie_{b\prec\!\prec c} R_c) & \cup\,(\Delta^+_a \bowtie_{a\prec\!\prec b} \Delta^+_b \bowtie_{b\prec\!\prec c} \Delta^+_c) \end{array}$$

vu_3 must then be translated to a source update and performed on d.

As the label path is stored within each ID, the insertion points of the new XML fragment within the document can be determined. For this XML fragment the IDs will be:
c_1: r1.a1.x1.b1.c2
c_2: r1.a1.f2.b1.c2

The IDs of all the tuples are not required to insert in the source, only the root of the subtree that is being inserted. In the case of IDs $r1.a1.x1.b1.c2$ and $r1.a1.f2.b1.c2$, the target nodes where the insertion has to occur, $r1.a1.x1.b1$ and $r1.a1.f2.b1$, is known. Therefore, the subtree is inserted as descendants of these nodes.

Therefore, d will become:

$$\begin{tikzcd}
 & & & r_{r1} \arrow[d, dash] & & & \\
 & & & a_{r1.a1} \arrow[dll, dash] \arrow[drr, dash] & & & \\
 & x_{r1.a1.x1} \arrow[d, dash] & & & & f_{r1.a1.f2} \arrow[d, dash] & \\
 & b_{r1.a1.x1.b1} \arrow[dl, dash] \arrow[dr, dash] & & & & b_{r1.a1.f2.b1} \arrow[dl, dash] \arrow[dr, dash] & \\
y_{r1.a1.x1.b1.y1} \arrow[d, dash] & & c_{r1.a1.x1.b1.c2} & & c_{r1.a1.f2.b1.c1} & & c_{r1.a1.f2.b1.c2} \\
c_{r1.a1.x1.b1.y1.c1} & & & & & &
\end{tikzcd}$$

Term Pruning

The pruning techniques that are used for insertions in view maintenance can be applied to view update insertion expressions. These can be used directly as the only difference with the update expression is the origin of the $v\Delta$ relations. Proposition 4.1.4 can be used directly to prune by the update semantics; Proposition 4.1.7 can be used for inserted data-driven pruning and Proposition 4.1.9 can be used for inserted ID-driven pruning.

Propagating view insert by view nodes

This first view update algorithm considers the case when an insertion on a view leads to new tuples being added to the view, and/or increasing the derivation count of existing view tuples. The insertion will then have to be reflected on the source document.

Algorithm 9 outlines the propagation procedure and is symmetric to Algorithm 1 for view maintenance. The computation of the $v\Delta^+$ relations will be detailed shortly. Again, as in Section 4.1.1, the core of the complexity lies in line 4.

Computing $v\Delta^+$ relations Given a view insertion vu on the view v, Algorithm 10 computes the $v\Delta^+$ relations.

Evaluate insertion Algorithm 11 (**ET-VINS**) outlines the evaluation of non-pruned terms resulting from insertions. This is symmetric to Algorithm 3 for view maintenance.

ALGORITHM 9: Propagate View Insert by New Nodes **(PVINN)**

```
Input: insert view update $vu$, view $v$, lattice $l$, documents $d$
Output: updated view $v'$ to reflect $vu$; updated lattice $l'$ to reflect $vu$; updated documents
        $d'$ to reflect $vu$
1 Compute the $v\Delta^+$ relations corresponding to $vu$(call Algorithm CVD+($vu$, $v$, $d$))
2 Develop the $2^{v\Delta} - 1$ union terms to be added to $v$ in case of insertions, and prune them
  based on XQuery Update semantics (Proposition 4.1.4)
3 Further prune terms based on the $v\Delta^+$ relations (Propositions 4.1.9)
4 Evaluate the remaining terms and add their results to $v$ (call Algorithm ET-VINS($vu$, $v$,
  $v\Delta^+$, $l$))
5 Update the snowcaps from the bottom up in the lattice
6 Update documents (call Algorithm US($vu$, $v$, $d$))
```

ALGORITHM 10: Compute $v\Delta^+$ relations (**CVD+**)

```
Input: view update $vu$, view $v$, documents $d$
Output: $v\Delta^+$ relations
1 $(vn_1, t_1), \ldots, (vn_k, t_k) \leftarrow$ compute-pul($vu$, $v$, $d$)
2 for $vn \in vn_k$ labelled $l$ do
3     let $p_l$ be the pattern $//l_{attributes}$
4     $v\Delta_l^+ \leftarrow \bigcup_{1 \le i \le k}$(extr-pattern($vn_{id}$, $p_l$, $t_i$))
```

Updating the source Algorithm 12 handles the source update. It works by adding the predicates from the view, if any exist, to the XPath query which defines the location of the target node(s) for the update. The purpose of this is so that only the target nodes from the view are updated in the source. Without including the predicates extra fragments of XML may be inserted. To update the source, this new query is used by the update and performed on each affected document.

Example of source update

Example 4.2.5. *Consider the view $v_3 = //a/b/c[d][e]$ and the view update vu_4 that inserts in v_3 at $//a/b/c$. If the predicates are not added to the update query for the source update*

ALGORITHM 11: Evaluate terms resulting from view insert (**ET-VINS**)

```
Input: view update $vu$, view $v$, delta relations $v\Delta^+$, lattice $l$
Output: updated view $v'$ to reflect $vu$; updated lattice $l'$ to reflect $vu$
1 $v\Delta_v^+ \leftarrow \emptyset$ (tuples to be added to $v$)
2 foreach term $t$ surviving pruning do
3     Evaluate $t_{v\Delta^+}$ by structural joins over the $v\Delta^+$ relations
4     Add to $v\Delta_v^+$ the result of joining $t_R$ (snowcap materialised in the lattice) and $t_{v\Delta^+}$
5 foreach tuple $t_{v\Delta} \in v\Delta_v^+$ do
6     if $t_{v\Delta} \in v$ then
7         increase the derivation count of $t_{v\Delta}$
8     else
9         add $t_{v\Delta}$ to $v$ with a derivation count of 1
```

ALGORITHM 12: Update Source **(US)**

```
Input  : update $vu$, view $v$, documents $docs$
Output: updated documents $docs'$ to reflect $vu$
1 Add predicates from view into update query path
2 foreach document $d \in docs$ do
3     Perform the update using the new update query path
```

then c nodes may be modified which don't contain d and e branches. Therefore, the update query must become $//a/b/c[d][e]$. *Alternatively, if the update was applied to* $//a/b$ *then subsequently it would have to become* $//a/b[c[d][e]]$ *for the source update.*

4.2.4 Deletions

This section will discuss the propagation of deletions for view update. The following example is symmetric to Example 4.1.10 and illustrates a simple deletion scenario.

Example 4.2.6. *Consider the view* $//a_{ID}//b_{ID}$ *and the XML document d shown in Figure 4.3, where the ID of each node is shown as a subscript to the node. Each ID is a sequence of steps, each step holding the label and the relative position of one ancestor of the* *node*[2].

Consider a view update vu_1 deleting the tuple $(a1, a1.c1.b1)$, which targets the node whose ID is a1.c1.b1. Since the view tuple $(a1, a1.c1.b1)$ had a derivation count of 1, the update leads to removing the tuple from the view. Then the corresponding element from the source document needs to be removed. This extends to the case where there are multiple target nodes.

The approach is similar to the view maintenance method for deletions. The difference again lies in the implementation. For further details see Section 4.1.2. The remainder of this section is structured as follows: first the set of pruning methods that can be used are discussed, and then the algorithm for handling deletions is introduced.

Term Pruning

The pruning techniques that are used for deletions in view maintenance can be applied to view update deletion expressions. These can be used directly as the only difference with the update expression is the origin of the $v\Delta$ relations. Propositions 4.1.11 and 4.1.12 can be used directly to prune by the update semantics and Proposition 4.1.16 can be used for inserted ID-driven pruning.

Propagate view delete by deleting nodes

The general algorithm Propagate View Delete by Deleting Nodes (**PVDDN**), which propagates a deletion on the view to the XML source document(s), is outlined in Algorithm 13. This algorithm covers both tuple deletions, and decreasing the derivation count of a tuple while not necessarily removing it from the view. Both these cases require computing and performing deletions on the XML source document(s). The algorithm **CVD-** is not detailed as it is similar to **CVD+**. The only difference is that the documents are not needed as all the required information is available within the view. The following example illustrates the scenario when a deletion results in modifying the derivation count of a tuple but not removing it from the view.

[2]Internally ID representation is much more compact.

Example 4.2.7. *Consider the view* $//a_{ID}[//b]$ *and the document* d *in Figure 4.3, and an update deleting the tuple* $(a1, a1.c1.b1)$*.*

The view contains a single tuple corresponding to node a*. The tuple has a derivation count of 2 due to the two* b *nodes which satisfy the predicate in the view. Therefore, the deletion does not affect the view, since by deleting only the* b *node identified by a1.c1.b1, the* a *element still has a* b *descendant. The only effect of the update is to decrease the derivation count by 1. Then similarly to the previous example we just have to delete the element corresponding to the tuple from the source document.*

4.2.5 Modifications

In a similar manner to view maintenance, modification of existing tuples is a possible side-effect of insertions and deletions. As modifications only affect the views, not the documents, they are handled as in the view maintenance method. For further information see Section 4.1.3.

Section 4.2 has explained the methods involved in incremental view update. In the next section (4.3), attention is turned to dynamic reasoning on updates by means of pruning rules, which has relevance both to view maintenance and view update.

4.3 Optimisation using Lazy Evaluation and Dynamic Reasoning on XML Updates

The previous methods address the problem of incrementally updating materialised views in the presence of statement-level XML updates. These methods can be applied after each update. However, when a sequence of updates is applied to a document, their propagation to the views may be deferred and applied lazily. Updating the views as updates arrive (i.e., immediate evaluation) may be inefficient for incrementally updating views compared with performing bulk updates when they are required (i.e., lazy evaluation).

Recent work [156] has studied how to apply dynamic reasoning on sequences of XML updates - expressed as a PUL (See Section 3.4) - without needing to access the source

ALGORITHM 13: Propagate View Delete by Deleting Nodes **(PVDDN)**

```
Input: view deletion update $vu$, view $v$, lattice $l$, documents $d$
Output: updated view $v'$ to reflect $vu$, updated lattice $l'$ to reflect $vu$, updated documents
        $d'$ to reflect $vu$
1  Compute the $v\Delta^-$ relations corresponding to $vu$ (call Algorithm CVD-($vu$, $v$))
2  Expand the deletion expression to $2^{v\Delta} - 1$ terms and prune them based on the data
   (Propositions 4.1.11 and 4.1.12) and then based on the $v\Delta^-$ relations (Proposition 4.1.16)
3  $v\Delta^-_v \leftarrow \emptyset$ (tuples to be deleted from $v$)
4  foreach term $t$ surviving pruning do
5      Evaluate $t_{v\Delta^-}$ by structural joins over the $v\Delta^-$ relations
6      Add to $v\Delta^-_v$ the result of joining $t_R$ (snowcap materialised in the lattice) and $t_{v\Delta^-}$
7  foreach tuple $t_v \in$ view $v$ do
8      if $t_v \in v\Delta^-_v$ then
9          remove $t_v$ from $v$
10     else
11         if $t_v$ stores ID, val or cont for a node $n$ (i) appearing in some $v\Delta^-$ relation or (ii)
           having an ancestor in some $v\Delta^-$ relation then
12             decrease $t_v$ derivation count
13             if $t_v$'s derivation count becomes 0 then
14                 remove $t_v$ from $v$
15 Update the snowcaps from the bottom up in the lattice
16 Update the documents (call Algorithm Update Source($vu$, $v$, $d$))
```

documents. As the methods for view maintenance and view update relied on PULs it seemed natural to utilise these optimisation methods. Additionally, since the methods do not need to access the source documents, optimisations can be performed remotely from their storage. This allows simple integration with the presented methods as the optimised PULs can be used instead of the originals.

Several techniques can be used for dynamic reasoning. On sequences of updates one can perform a method known as reduction, which collapses similar operations and removes

$$\begin{tikzcd}
v \arrow[dddr] & & d \arrow[dr] & & u \arrow[dl] \\
& & & \boxed{\text{CP}} \arrow[dl] & \\
& & \boxed{\text{OR}} \arrow[dl] & & \\
& \boxed{\text{PINT/PDDT}} \arrow[rr] \arrow[dr] & & \boxed{\text{ET-INS/DEL}} \arrow[d] & \\
& & v' & \boxed{\text{CD+/-}} &
\end{tikzcd}$$

Figure 4.10: Interleaving PINT/PDDT with Optimisation Rules (OR).

useless operations (i.e., operations whose effects are overridden by others). Additionally, depending on whether sequences of updates are to be executed in parallel or sequentially, further methods such as integration of parallel sequences of updates and aggregation of sequential update sequences, can also be applied. This can improve the efficiency of the presented incremental view maintenance and view update methods. In this section, how to handle sequences of updates within this work is discussed, by applying some of the methods envisioned in [156].

4.3.1 Sequencing Updates

The rules presented in [156] are defined for PULs. The interaction between the view maintenance method and the optimisation rules can be seen in Figure 4.10. The PUL is computed from the updates and the document. The optimisation rules are then applied to the PUL. The optimised PUL is then propagated to the view instead of the original. Figure 4.10 shows that the optimised sequences of atomic updates are used in the algorithm **PINT/PIMT** (**PDDT/PDMT**, respectively) and the algorithms which they call. Precisely, each insert/delete in a set of statement-level updates for multiple return nodes is transformed into atomic updates by **CP** (compute-pul) and the optimisation rules **OR** applied to them. The remaining updates are then evaluated as in the view maintenance method. Optimisations for the view update method are handled similarly.

4.3.2 Update Operations

Eleven elementary update operations are handled in [156]. Two fundamental update operations are considered here.

- insert a forest[3] P after the last child of a node v, denoted $ins^{\searrow}(v, P)$
- delete a node v, denoted $del(v)$

4.3.3 Rules

The rules that are applicable to this algebraic incremental view maintenance work are detailed below. Reduction rules allow useless operations to be removed from the update sequence. When multiple PULs have to be evaluated on the same document it would be beneficial to integrate these into a single PUL. However, as conflicts can occur a set of conflict rules are presented. Finally, if a collection of PULs are to be executed sequentially then the aggregation rules can be used.

Reduction Rules

These rules allow a more compact PUL that has the same effect as the original one. In [156] there are 20 reduction rules split over nine stages. Each stage contains a subset of the reduction rules where the order of their execution is irrelevant/undefined. The next stage can not be evaluated until all the rules from the current stage have been performed. However, as currently only a subset of update operations are handled in this work, only some rules from the first two stages are applicable. These stages are O - for removing overridden operations - and I - for collapsing insertions, where possible, i.e., same target node. The rules show how operations can be converted into a single operation. The applicable rules are detailed below.

O1) $\dfrac{op_1{=}op(v,\_) \quad op_2{=}op'(v,\_)}{op_1,\, op_2 \nabla_1 op_2} \begin{array}{c} op \in \{del, ins^{\searrow}\} \\ op' \in \{del\} \end{array}$ O3) $\dfrac{op_1{=}op(v,\_) \quad op_2{=}op'(v',\_)}{op_1,\, op_2 \nabla_1 op_2} \begin{array}{c} op \in \{ins^{\searrow}, del\} \\ op' \in \{del\} \end{array} \; v//_d v'$

I5) $\dfrac{op_1{=}op(v,L_1) \quad op_2{=}op(v,L_2)}{op_1,\, op2 \nabla_1 op(v,[L_1,L_2])} \; c(op) = i$

[3]a collection of XML trees

O1: If there is an insertion or deletion, followed by a deletion on the same target node, then just perform the second deletion.

O3: If there is an insertion or a deletion, followed by a deletion on an ancestor of the first operation, then just perform the second operation.

I5: If there is an insertion, followed by another insertion on the same target node, then combine the insertions into one.

In the above operations, the ∇ operator reduces a sequence of input operations, e.g., $op_1,\ op_2 \nabla_1 op_2$ denotes the reduction of the operations op_1 and op_2 into just op_2. The numerical subscript indicates the stage which the rules belongs to, i.e., ∇_1 belongs to stage one. $n_{//d}n'$, in rule O3, declares that n is a descendant of n'. Finally, $c(op) = i$ declares the class of operations which the rules apply to. For the instance in I5 this is applicable to insertions, more specifically, for this work $c(op) = ins \searrow$.

Conflict Rules

For update sequences to be executed in parallel a method for integrating them has been created. However, conflicts can arise. Therefore, conflict rules have been specified. The rules can pick up the following conflicts: repeated modifications; repeated attribute insertion; local override; and non-local override. However, the only ones that are applicable to this work, based on currently available operations, are detailed below.

$$\text{IO)}\ \frac{t(op_1)=t(op_2) \quad o(op_1)=o(op_2)\,\epsilon\,\{ins^{\searrow}\}}{op_1 \overset{3}{\leftrightarrow} op_2} \qquad \text{LO)}\ \frac{t(op_1)=t(op_2) \quad o(op_1)\,\epsilon\,\{del\}\ o(op_2)\,\epsilon\,\{ins^{\searrow}\}}{op_1 \overset{4}{\rightarrow} op_2}$$

$$\text{NLO)}\ \frac{t(op_2)//_d t(op_1) \quad o(op_1)\,\epsilon\,\{del\}\ o(op_2)\,\epsilon\,\{ins^{\searrow}\}}{op_1 \overset{5}{\rightarrow} op_2}$$

In these rules $t(op)$ represents an operation's target and $o(op)$ represents its name.

IO: This operation (Insertion Order) identifies the case where operations are different depending on execution order.

LO: This operation (Local Overriding) identifies the case when an operation is overridden by another with the same target node. This causes the first operation to not affect the document due to the presence of the second one.

NLO: This operation (Non-Local Overriding) identifies the case when an operation is overridden by another with a different target node. This case arises when the deletion's target node is an ancestor of the insertion's target node.

The conflicts are defined as symmetric or asymmetric. IO is symmetric, as both operations have the same type and target node, as defined by the substitutability symbol $\overset{3}{\leftrightarrow}$. The 3 is the rule order [156] which has been kept the same for compatibility. LO and NLO are both asymmetric as the first operation overrides the second, defined by $op_1 \overset{4}{\rightarrow} op_2$, where again the numerical subscript represents the rule number.

After applying the conflict rules to the update sequence a list of non-conflicting operations are returned along with the identified conflicts. The work allows PUL producers to define conflict resolution policies in order to solve conflicts during PUL integration.

Aggregation Rules

For updates to be executed sequentially a method for aggregating them has been created. Aggregation involves merging multiple sequences. Precisely, given two sequences Δ_1 and Δ_2, their aggregation into a single sequence Δ corresponds to the sequential execution of $\Delta_1; \Delta_2$, where Δ_1 is applied to the original document and Δ_2 is applied to the document updated through Δ_1. Dependencies must be removed before this occurs using the defined aggregation rules. Again, not all these rules apply to this work, the ones that do are detailed below.

$$\text{A1)}\ \frac{op_1{=}op(v{,}L_1) \quad op_2{=}op(v{,}L_2)}{\Delta'_1{\cup}\{op_1{,}op_2\},\ \Delta'_2 \overset{A}{\multimap}_1 \Delta'_1{\cup}\{op(v{,}[L_1{,}L_2])\},\ \Delta'_2}\ c(op) = i$$

$$\text{A2)}\ \frac{op_1{=}op(v{,}L_1) \quad op_2{=}op(v{,}L_2)}{\Delta'_1,\ \Delta'_2{\cup}\{op_1{,}op_2\} \overset{A}{\multimap}_1 \Delta'_1,\ \Delta'_2{\cup}\{op(v{,}[L_1{,}L_2])\}}\ c(op) = i$$

$$\text{D6)}\ \frac{op_1{=}op(v{,}T_1 \cdots T_i \cdots T_n) \quad \Delta_{v}i{=}\{o \in \Delta_2 \mid t(o){=}v'\}}{\Delta'_i{\cup}\{op_1\},\ \Delta_2 \overset{A}{\multimap}_4 \Delta'_1{\cup}\{op(v{,}P)\},\ \Delta_2 \backslash \Delta_{v'}}\ \begin{matrix} v' \in V(T_i), \\ T_i {\models} \Delta_{v'}\ leads\ to\ T'_i \\ P{=}T_1 \cdots T'_i \cdots T_n \end{matrix}$$

Rules A1 and A2 handle the aggregation of operations with the same target node and type, while D6 operates on a different node and type.

A1: If both the insertions are on the same node then aggregate the PULs by adding the second insert to the first PUL, after the first insert.

A2: This is A1 in reverse. Add the insert from the first PUL into the second PUL before the second insert.

D6: This handles the case where operations in the second PUL reference node(s) of a tree which is a parameter of an operation in the first PUL. In this case the operations in the second PUL, referencing the tree, are performed and removed from the second PUL.

The aggregation operator is denoted as $\overset{A}{\multimap}$. Again, the numerical subscript declares the stage which the rule belongs to.

4.3.4 Examples

The following examples show how the reduction, integration and aggregation of PULs are handled. These PULs are represented in the syntax presented in the view update and view maintenance methods, i.e., encoding the IDs using DDE and making them explicit. Example 4.3.1 shows how a PUL can be reduced. Example 4.3.2 shows how two parallel PULs can be integrated. Finally Example 4.3.3 shows how two sequential PULs can be aggregated.

Example 4.3.1. *Let Δ be the PUL specified on the document doc in Figure 4.11 containing the following operations:*
$op_1 = ins \searrow (a1.c1.b1.d1.b1, \langle b\rangle\langle d/\rangle\langle/b\rangle),$
$op_2 = del(a1.c1.b1.d1.b1),$
$op_3 = ins \searrow (a1.c1.b1.d2.b1, \langle b/\rangle),$
$op_4 = del(a1.c1.b1.d2),$
$op_5 = ins \searrow (a1.c1.b1.d3, \langle b/\rangle),$
$op_6 = ins \searrow (a1.c1.b1.d3, \langle d\rangle\langle b/\rangle\langle/d\rangle)$

Let v be the view //b//d//b over doc. The reduced PUL is $\{del(a1.c1.b1.d1.b1), del(a1.c1.b1.d2), ins \searrow (a1.c1.b1.d3, \langle b/\rangle, \langle d\rangle\langle b/\rangle\langle/d\rangle)\}$. *This is because op_1 is ignored due to rule O1; op_3 is ignored due to rule O3; and op_5 and op_6 are combined due to rule I5.*

Example 4.3.2. *Let* $\Delta_1 = \{op_1^1 = ins \searrow (a1.c1.b1.d1, \langle d\rangle\langle b/\rangle\langle/d\rangle), op_1^2 = del(a1.c1.b1.d2), op_1^3 = del(a1.c1.b1.d3)\}$, $\Delta_2 = \{op_2^1 = ins \searrow (a1.c1.b1.d1, \langle b/\rangle), op_2^2 = ins \searrow (a1.c1.b1.d2, \langle b/\rangle), op_2^3 =$

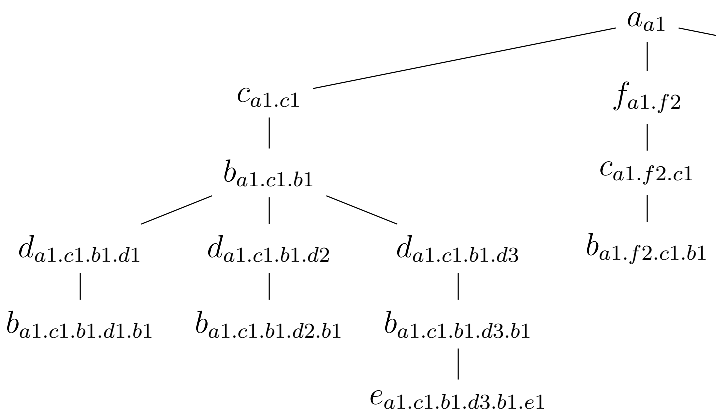


FIGURE 4.11: Sample XML document.

$ins \searrow (a1.c1.b1.d3.b1, \langle b/\rangle)\}$ be the PULs over document doc in Figure 4.11. Let v be the view //b//d//b over doc.

This example can't be aggregated as every operation causes a conflict. op_1^1 and op_2^1 are in conflict due to the insertion order (IO) rule; op_1^2 and op_2^2 are in conflict due to the local overriding (LO) rule; and op_1^3 and op_2^3 are in conflict due to the non-local overriding (NLO) rule. How these conflicts are solved depends on the conflict resolution policies the PUL producers specify. The algorithm will fail if it cannot identify a valid reconciliation, which is a PUL with no conflicts which satisfies the policies of all the PUL producers involved.

Example 4.3.3. *Let $\Delta_1 = \{op_1^1 = ins \searrow (a1.c1.b1.d1.b1, \langle c\rangle\langle b/\rangle\langle /c\rangle)$,
$op_1^2 = ins \searrow (a1.c1.b1.d2, \langle b/\rangle), op_1^3 = ins \searrow (a1.c1.b1.d3, \langle d\rangle\langle b/\rangle\langle /d\rangle)\}$,
$\Delta_2 = \{op_2^1 = ins \searrow (a1.c1.b1.d1.b1, \langle b/\rangle), op_2^2 = ins \searrow (a1.c1.b1.d2, \langle d\rangle\langle b/\rangle\langle /d\rangle)$,
$op_2^3 = ins \searrow (a1.c1.b1.d3.b1, \langle b/\rangle)\}$. Let v be the view //b//d//b over doc.*

*Aggregation $\Delta_1 \mapsto \Delta_2$ is $\{op_1^1 = ins \searrow (a1.c1.b1.d1.b1, \langle c\rangle\langle b/\rangle\langle /c\rangle, \langle b/\rangle)$,
$op_2^2 = (a1.c1.b1.d2, \langle b/\rangle, \langle d\rangle\langle b/\rangle\langle /d\rangle), op_1^3 = ins \searrow (a1.c1.b1.d3, \langle d\rangle\langle b/\rangle\langle b/\rangle\langle /d\rangle)\}$. This is because op_1^1 and op_2^1 have the same target node, so the XML insertion fragments are combined into the one operation due to rule A1; op_1^2 and op_2^2 are combined due to rule A2 (A1 in reverse); and op_1^3 and op_2^3 are combined due to rule D6. The XML fragment for the final insertion operations are combined due to the second fragment being an insertion on a node*

of the first XML fragment to be inserted.

This chapter has described the view maintenance and view update methods. It also detailed the dynamic reasoning rules utilised for PULs. Chapter 5 goes on to describe the approach taken to evaluate these ideas before Chapter 6 discusses the results.

5

Results

Chapter 4 presented the algebraic approach to specification of view maintenance and view update operations and the fundamental ideas behind the implementation of these approaches was presented in Chapter 3. To show the effectiveness of the methods for XML incremental view maintenance and update, a set of performance experiments and their associated results are presented in this chapter. To help with the explanation, a summary of results are presented here. The full set of results can be found in Appendix C. The results are discussed in Chapter 6.

5.1 Technical Details

The algorithms were implemented in Java 6, within the Views in Peer-to-Peer (ViP2P) (http://vip2p.saclay.inria.fr/) platform. ViP2P handles XML documents in peer-to-peer networks using a scalable method based upon materialised views and distributed hash table (DHT) indices. ViP2P provides the implementations of various components used within the algorithms: tree patterns from the $\mathcal{P}$ dialect; DDE implementation[1]; execution engine providing the required operators (projections, selections, joins, etc.); and XML-specific structural joins. Data is stored within BerkeleyDB v4.0.71. For operations on the source document Saxon XQuery processor v9.2.1.1j is used. All experiments were run on a PC with Linux Ubuntu v10.04, with a Pentium 4 3.40GHz CPU and 2GB memory.

5.2 View Maintenance

Testing was based on the approach taken in [157]. The XMark [158] data generator (xmlgen) was used to create benchmark XML documents of various sizes for the experiments. These sizes were: 100KB, 500KB, 1MB and 10MB. Views were defined by selected queries from the (read-only) XMark benchmark queries: Q_1, Q_2, Q_3, Q_4, Q_6, Q_13, Q_17. These queries were selected because they could be supported by XAMs, in some cases with slight modifications. The XPathMark benchmark [159] queries were used to derive a set of updates that inserted dummy elements or deleted the node(s) returned. Where possible the same views and updates as in [157] were used. This was because no benchmark for view maintenance existed when the experiments were performed, so a defined data set, consisting of views and updates from related works was used instead. However, as this work focussed on the independence of the views and updates, the views were not always affected, so view maintenance was not always required, which is the matter of interest in the current work. Therefore, where possible the updates were enhanced to affect the view, otherwise structurally related updates were added. The updates can be seen in Appendix B.

Each update name consists of a letter, a number, and ends with a letter(s). The initial

[1]added by this work

letter defines the subset of XPathMark queries it belongs to. The subsets are labelled A-F. The queries used have been taken from subsets A - unary tree pattern queries, B - core or navigational XPath queries, and E - XPath 1.0 queries. Queries denoted X are those that have been added. These were selected to give a set of queries that were representative of those supported by the system. The added ones were required where a type of query for a specific view was not defined within XPathMark. The number defines the query position within the subset. Finally, the end letter(s) describe the query type. Five query types have been used: (L) **L**inear path expressions; (A) path expression with an **A**nd predicate; (O) path expression with an **O**r predicate; (AO) path expression with an **A**nd **O**r predicate; and (LB) **L**inear path expression with a **B**oolean filter.

For the experiments, the following times were measured:

- *Find Target Nodes* - time taken to find the target node(s) of an insertion/deletion.
- *Compute Delta Tables* - time taken to build the Δ^+/Δ^- tables using the target nodes and XML fragment.
- *Get Update Expression* - time taken to build the algebraic expression corresponding to the view; generate the update expression; unfold (distribute the joins over the unions) and prune it.

  Note: Inserted data-driven pruning is performed when computing the delta tables. Tuples which don't match the view definition are discarded.
- *Build Lattice* - time taken to build the snowcap and leaves lattice, and populate it, based on the algebraic expression for the view.
- *Execute Update* - time taken to evaluate the algebraic update expression and modify the database accordingly. The time to perform modifications is also included, if they are required.
- *Update Lattice* - time taken to update the lattice. This is performed, for insertions, by adding the new tuples to the leaves and propagating them up the lattice, using joins to calculate the new tuples for insertions. Conversely, for deletions, removing the deleted tuples.

- *Update Source* - time taken to insert/delete XML, defined by the update, from the source document.

When these stages are mentioned in the results tables, the key shown in Table 5.1 is used.

Symbol	**Stage**
TN	Find Target Nodes
DT	Compute Delta Tables
UE	Get Update Expression
BL	Build Lattice
EU	Execute Update
UL	Update Lattice
US	Update Source

TABLE 5.1: Key for Results Tables

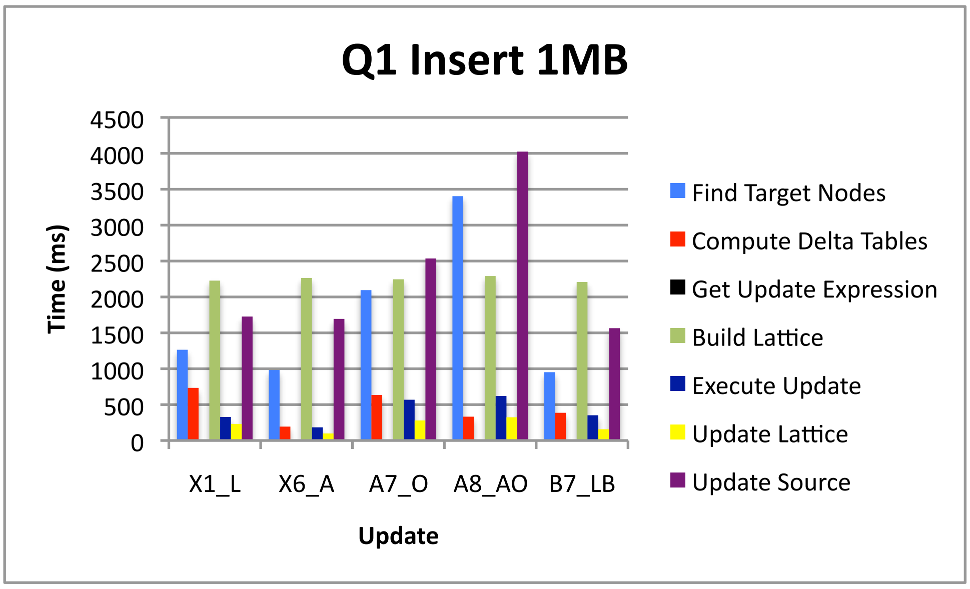
Comparable experiments were performed for insertions and deletions. Results presented in this chapter for updates on 1MB documents show the component times, the total times, the scalability, the impact of varying the path length and the impact of varying the annotations.

5.2.1 Insertions

The performance results for insertions on the views $Q1$, $Q2$ and $Q13$ for document size 1MB are shown in Tables 5.2, 5.3, 5.4 and Figures 5.1, 5.2 and 5.3 respectively. Each result set is presented as a detail table, a graphical representation of the detail, and a summary graph. These views were selected because they contained the most tuples and the definitions were the most varied. This was due to $Q1$ being similar to $Q17$ and $Q2$ being similar to $Q3$ and $Q4$. The difference being alternate or extra filters that reduced the size of the view. The results for the other views can be seen in Appendix C.1.1.

Query	TN	DT	UE	BL	EU	UL	US
X1_L	1264	733	8	2227	328	231	1727
X6_A	982	194	6	2264	184	99	1694
A7_O	2095	634	8	2246	568	280	2536
A8_AO	3404	332	9	2291	620	324	4024
B7_LB	952	386	6	2208	352	157	1565

Table 5.2: Q1 Insert Time for 1MB Document (ms)



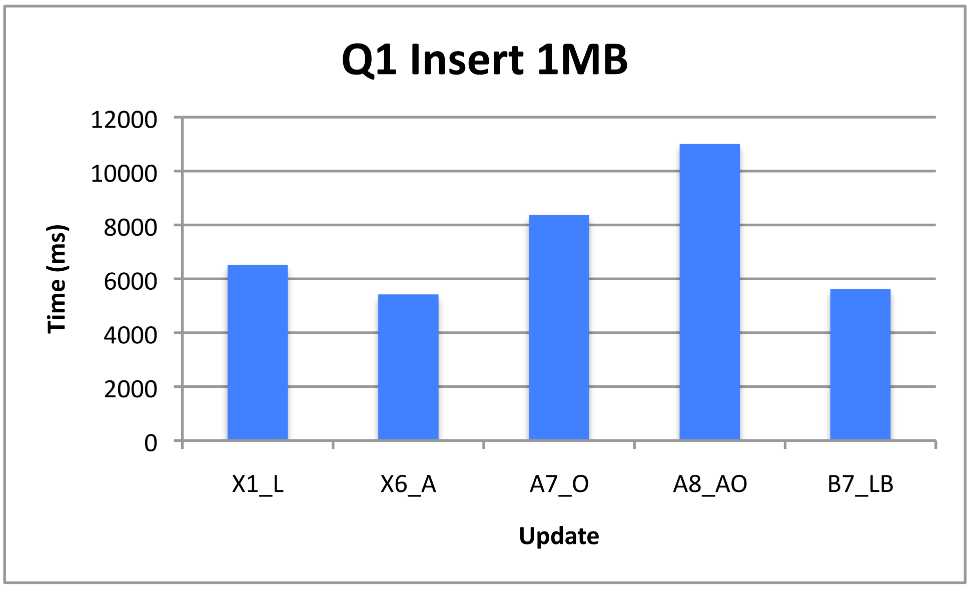
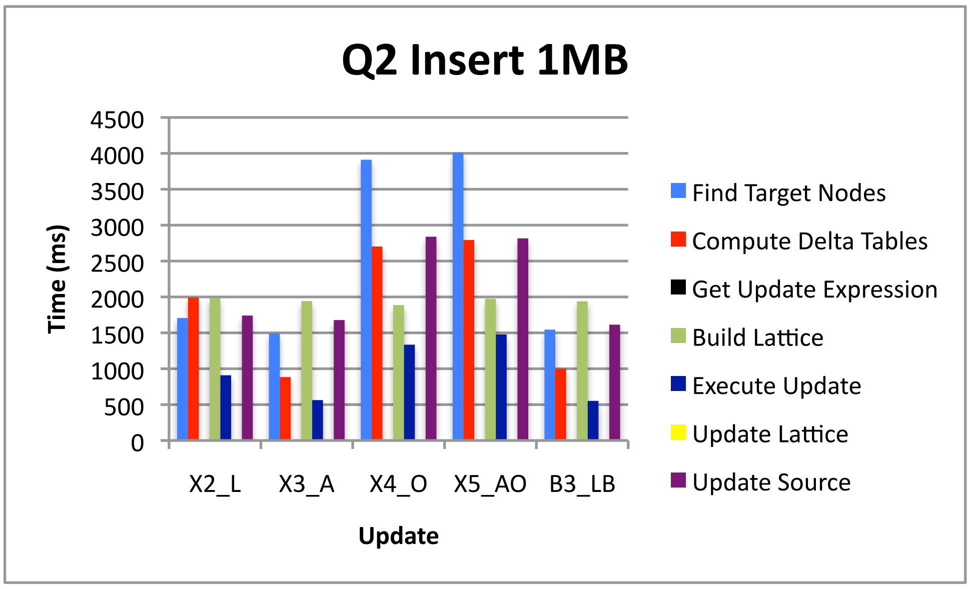


Figure 5.1: Q1 Insert Times Detail (upper) and Summary (lower)

Query	TN	DT	UE	BL	EU	UL	US
X2_L	1706	1993	9	1985	909	0	1741
X3_A	1492	884	8	1943	563	0	1678
X4_O	3911	2703	10	1887	1335	0	2839
X5_AO	4009	2793	10	1974	1478	1	2816
B3_LB	1545	1003	10	1940	552	0	1614

TABLE 5.3: Q2 Insert Time for 1MB Document (ms)



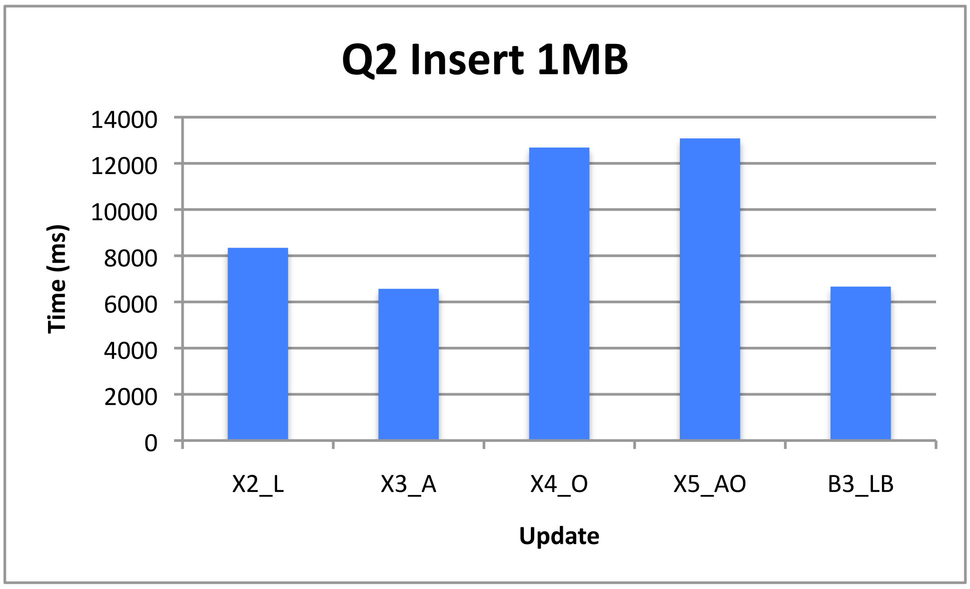
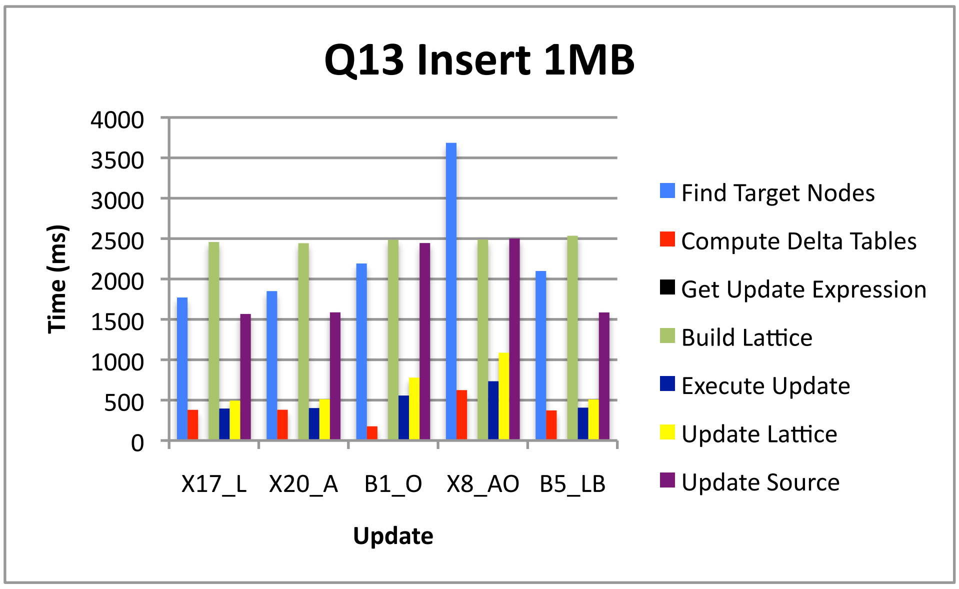


FIGURE 5.2: Q2 Insert Times Detail (upper) and Summary (lower)

Query	TN	DT	UE	BL	EU	UL	US
X17_L	1771	380	10	2458	396	494	1567
X20_A	1850	381	10	2443	402	512	1587
B1_O	2192	176	8	2485	557	779	2445
X8_AO	3686	624	12	2488	734	1087	2501
B5_LB	2099	373	10	2535	407	509	1586

Table 5.4: Q13 Insert Time for 1MB Document (ms)



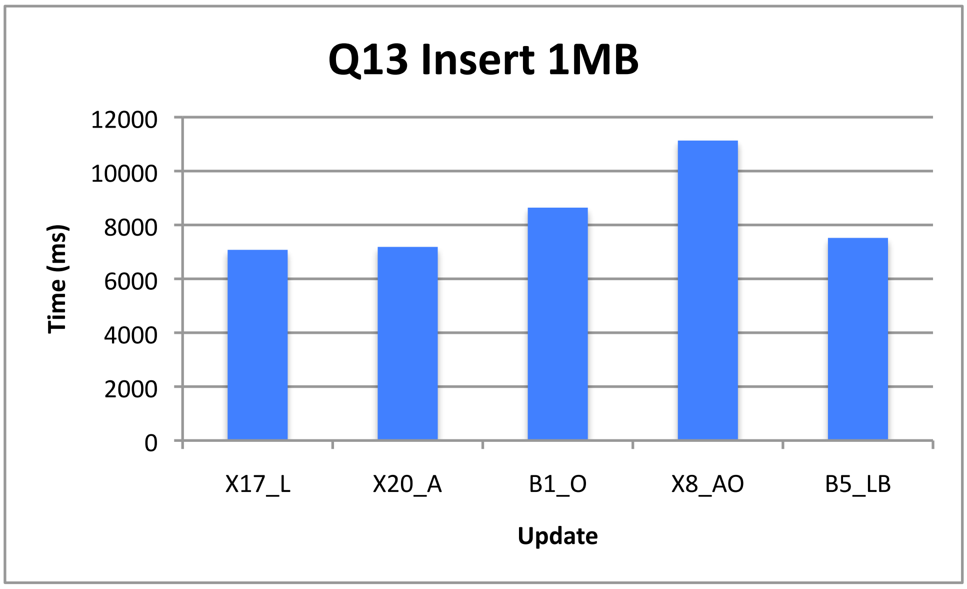


Figure 5.3: Q13 Insert Times Detail (upper) and Summary (lower)

Total Times

The total times for all the views and queries used for insertions testing, for the 1MB document, are shown in Table 5.5 and Figure 5.4.

Update	Q1	Q2	Q3	Q4	Q6	Q13	Q17
X1_L	6518						6326
X6_A	5423						5223
A7_O	8367						8255
A8_AO	11004						10929
B7_LB	5626						5636
X2_L		8343	6523	6994			
X3_A		6568	5815	6585			
X4_O		12685	9902	9994			
X5_AO		13081	10058	10171			
B3_LB		6664	5787	6682			
E6_L					5151		
B1_A					6170		
X7_O					9892		
X8_AO					9992	11132	
B5_LB					5167	7519	
X17_L						7076	
E5_A						7185	
B1_O						8642	

Table 5.5: Total Running Time (ms) for all Views and Queries

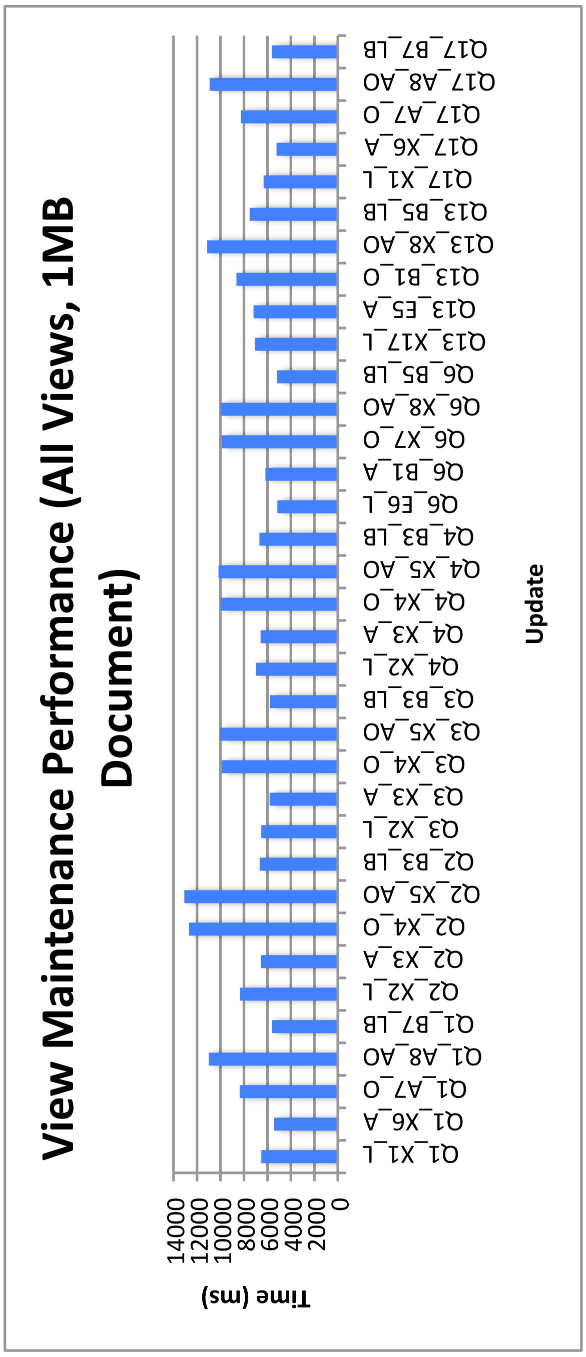


Figure 5.4: Total Running Time for all Views and Queries

Scalability

A scalability test was carried out to check the performance of the algorithms on larger source documents. This was to confirm that the method scaled up and was not only beneficial for smaller document sizes. The document sizes ranged from 100KB to 10MB. The results are shown in Table 5.6 and Figure 5.5.

Varying Path

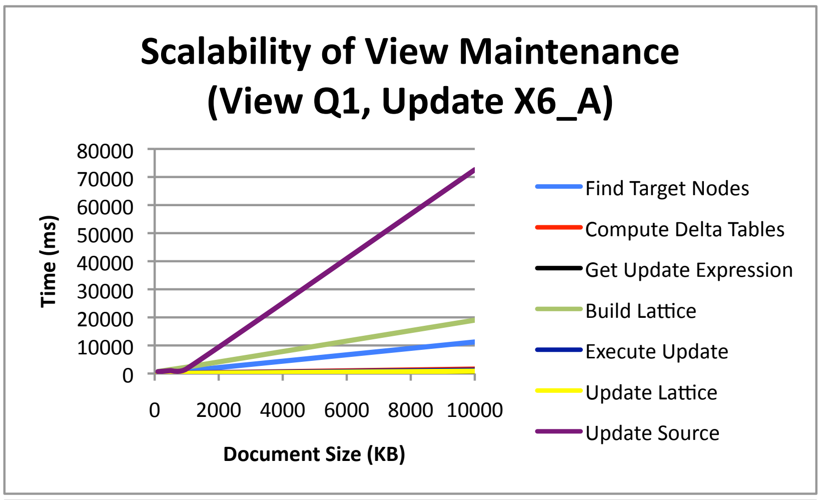
Experiments were performed to show the effect of varying the path length - the number of steps - of the XPath expression identifying the target node(s) of an update. For this experiment, view $Q1$ was used. The paths used were: (1) /site; (2) /site/people; and (3) /site/people/person. The results can be seen in Table 5.7 and Figure 5.6.

Varying Annotations

Experiments were performed to determine how different combinations of stored attributes on the view nodes affect the performance. For this experiment, update $X1\_L$ and view $Q1$ along with variations were used. For each node, the ID is always stored. The different combinations used were: (1) only IDs stored; (2) val and cont only stored for the leaves; (3) val and cont only stored for the root; and (4) val and cont stored for every node. The results can be seen in Table 5.8 and Figure 5.7.

	Document Size (KB)			
Stage	**100**	**500**	**1000**	**10000**
Find Target Nodes	192	604	982	11244
Compute Delta Tables	60	104	194	1592
Get Update Expression	6	9	6	9
Build Lattice	705	1425	2264	19007
Execute Update	140	188	184	1222
Update Lattice	83	121	99	853
Update Source	659	1034	1694	72600

Table 5.6: Scalability Q1 Insert Update X6_A (ms)



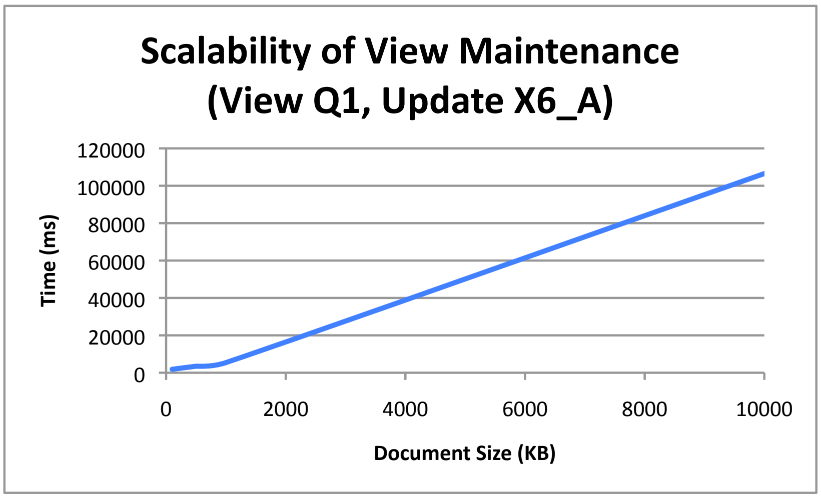


Figure 5.5: Scalability Q1 Insert Update X6_A

Query	TN	DT	UE	BL	EU	UL	US
/site	818	28	6	2328	224	666	1582
/site/people	845	29	6	2308	186	489	1544
/site/people/person	1264	733	8	2227	328	231	1727

TABLE 5.7: Varying Path for XMark View Q1 and Update X1_L for 1MB Document

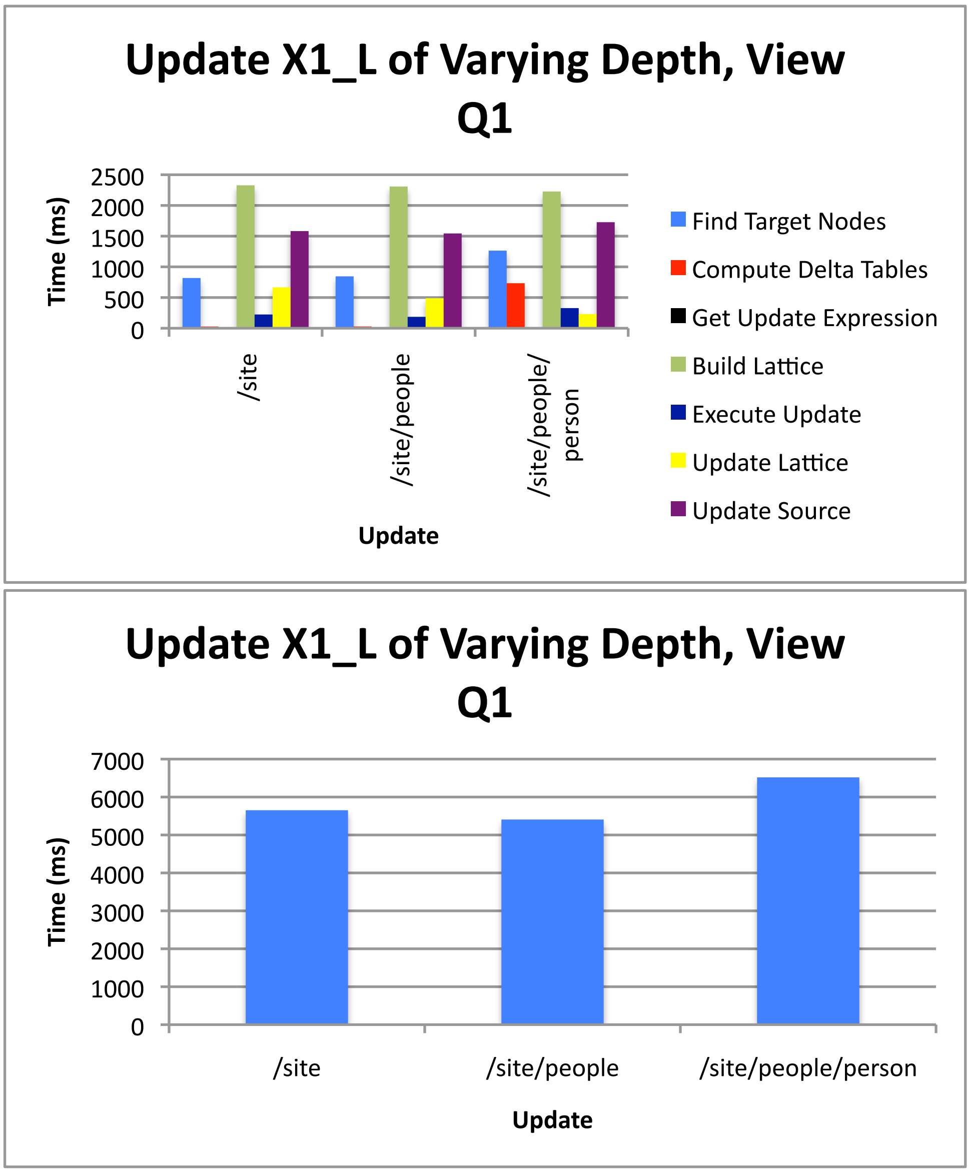


FIGURE 5.6: Varying Path for XMark View Q1 and Update X1_L for 1MB Document

Query	TN	DT	UE	BL	EU	UL	US
IDs	226	124	8	665	182	97	787
Leaves	230	124	6	679	179	103	775
Root	223	101	8	1247	1820	366	70
All	233	101	6	1313	2188	274	80

Table 5.8: Varying Annotations for XMark View Q1 and Update X1_L for 100KB Document

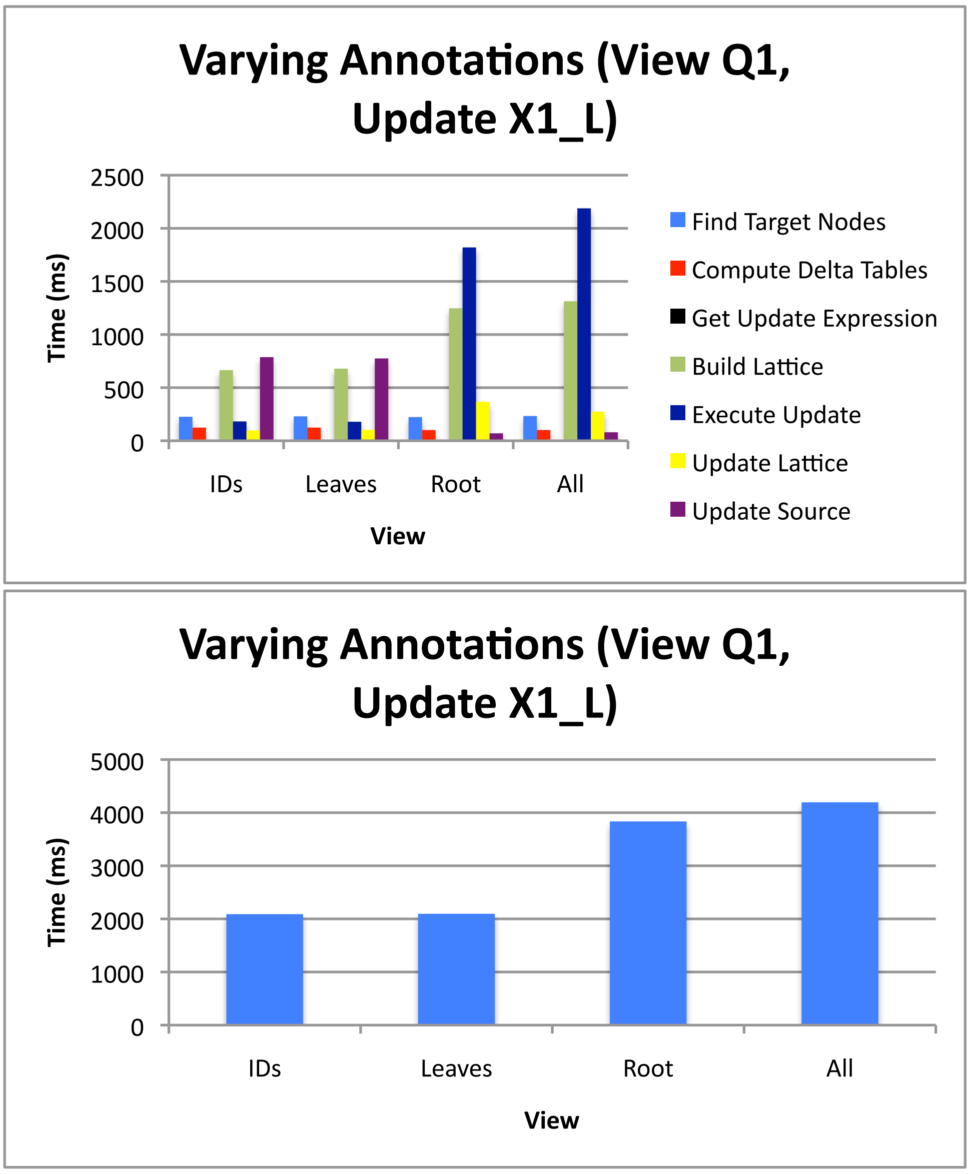


Figure 5.7: Varying Annotations for XMark View Q1 and Update X1_L for 100KB Document

5.2.2 Deletions

The performance results for deletions on the views $Q1$, $Q2$ and $Q13$ (used for the same reasons as the insertion tests) for document size 1MB are shown in Tables 5.9, 5.10, 5.11 and Figures 5.8, 5.9 and 5.10 respectively. The graphs show the results for the individual times as well as the total times. The results for the other views can be seen in Appendix C.1.2.

Total Times

The total times for all the views and queries used for testing are shown in Table 5.12 and Figure 5.11.

Scalability

A scalability test was carried out to check the performance of the algorithms on larger source documents. The document sizes ranged from 100KB to 10MB. The results are shown in Table 5.13 and Figure 5.12.

Varying Path

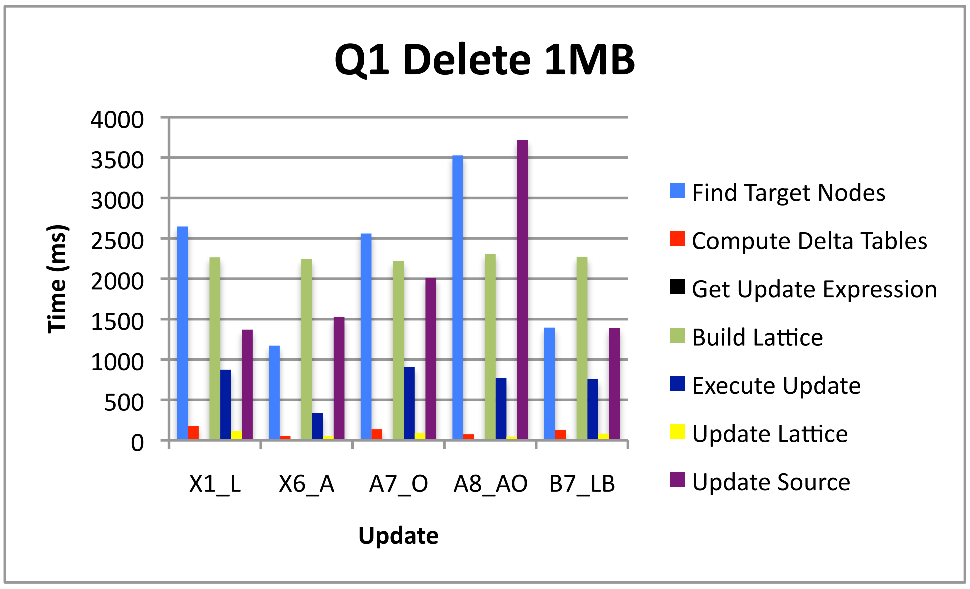
Experiments were performed to show the effect of varying the path length - the number of steps - of the XPath expression identifying the target node(s) of an update. For this experiment, view $Q1$ was used. The paths used were: (1) /site; (2) /site/people; (3) /site/people/person; (4) /site/people/person/name. The results can be seen in Table 5.14 and Figure 5.13.

Varying Annotations

Experiments were performed to determine how different combinations of stored attributes on the view nodes affect the performance. For this experiment, update $X1\_L$ and view $Q1$ along with variations were used. For each node, the ID is always stored. The different combinations used were: (1) only IDs stored; (2) val and cont only stored for the leaves; (3) val and cont only stored for the root; and (4) val and cont stored for every node. The results can be seen in Table 5.15 and Figure 5.14.

Query	TN	DT	UE	BL	EU	UL	US
X1_L	2647	178	14	2265	874	114	1370
X6_A	1172	54	9	2243	337	50	1525
A7_O	2560	136	14	2217	905	92	2012
A8_AO	3527	74	14	2307	771	46	3719
B7_LB	1395	130	12	2271	756	82	1389

Table 5.9: Q1 Delete Time for 1MB Document (ms)



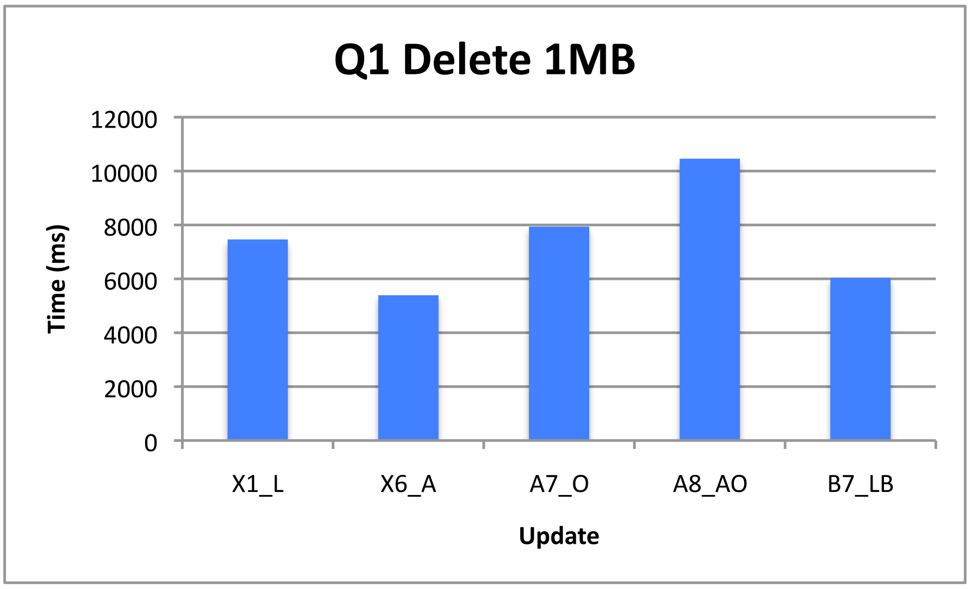
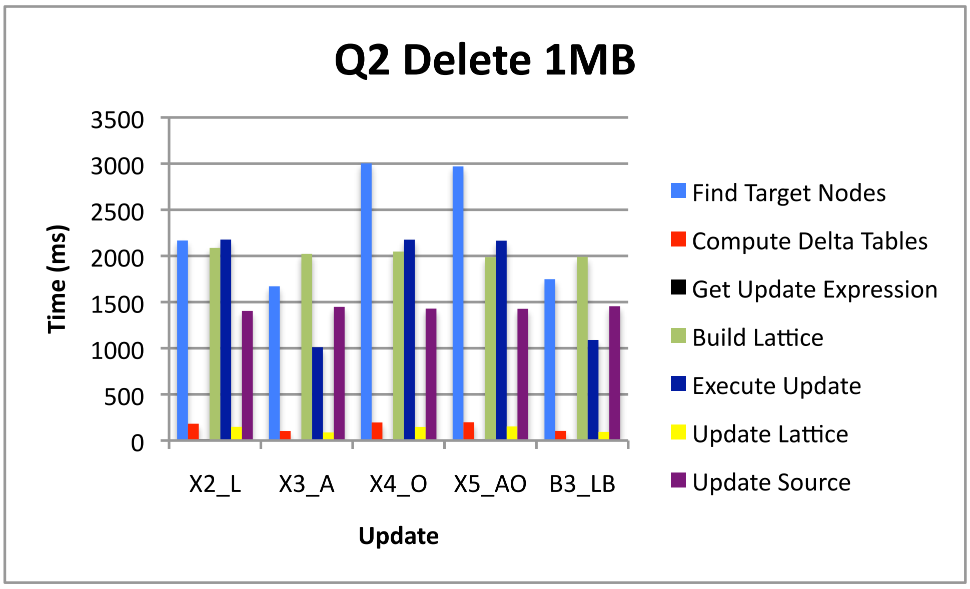


Figure 5.8: Q1 Delete Times Detail (upper) and Summary (lower)

Query	TN	DT	UE	BL	EU	UL	US
X2_L	2167	182	14	2087	2177	148	1404
X3_A	1671	103	11	2022	1011	88	1448
X4_O	3003	196	14	2047	2176	147	1429
X5_AO	2970	198	13	1987	2165	153	1427
B3_LB	1748	104	11	1990	1089	93	1455

TABLE 5.10: Q2 Delete Time for 1MB Document (ms)



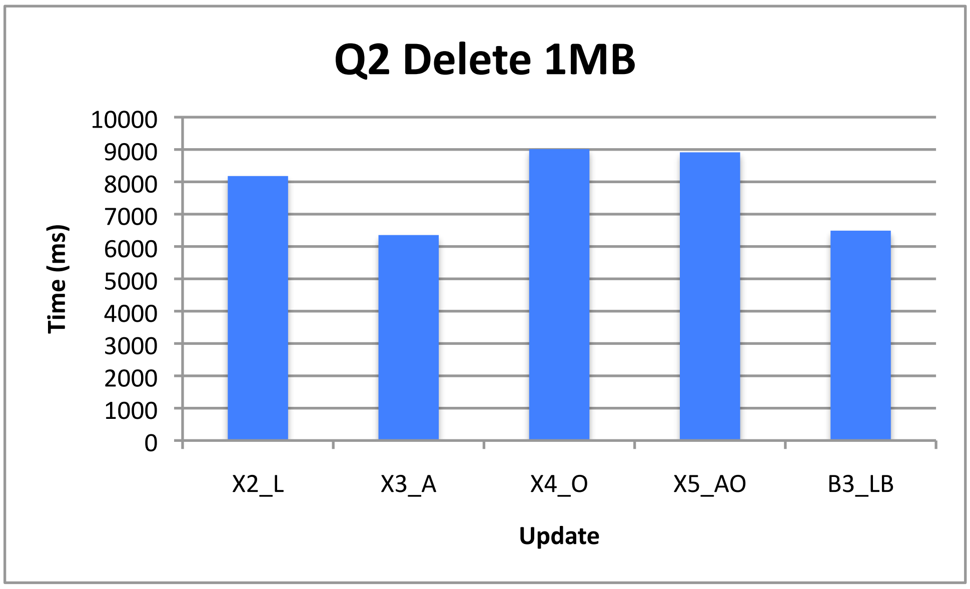
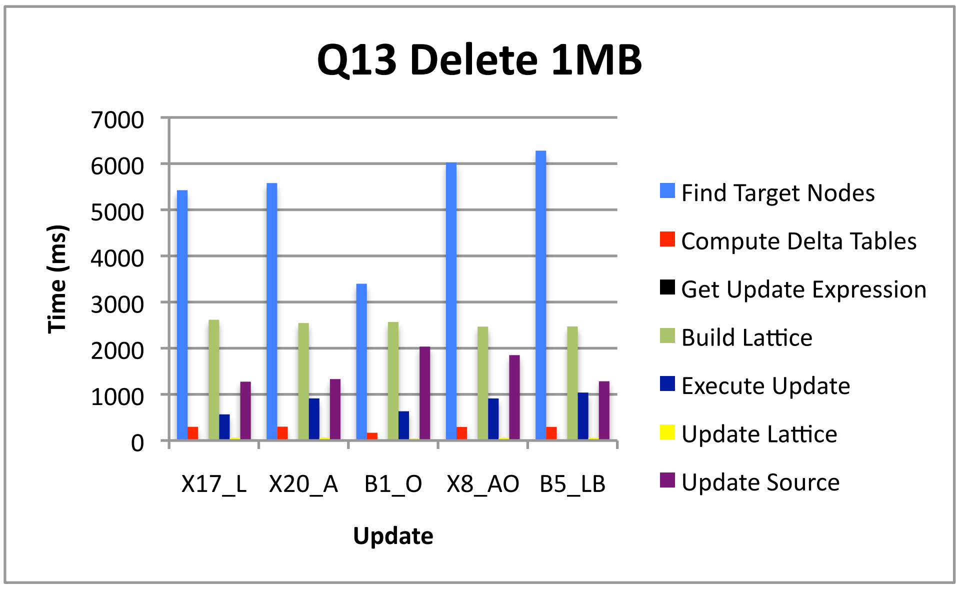


FIGURE 5.9: Q2 Delete Times Detail (upper) and Summary (lower)

Query	TN	DT	UE	BL	EU	UL	US
X17_L	5424	297	14	2617	567	64	1275
X20_A	5578	300	13	2545	913	67	1331
B1_O	3397	168	14	2568	634	45	2035
X8_AO	6025	294	15	2468	912	63	1850
B5_LB	6280	295	14	2472	1039	62	1284

Table 5.11: Q13 Delete Time for 1MB Document (ms)



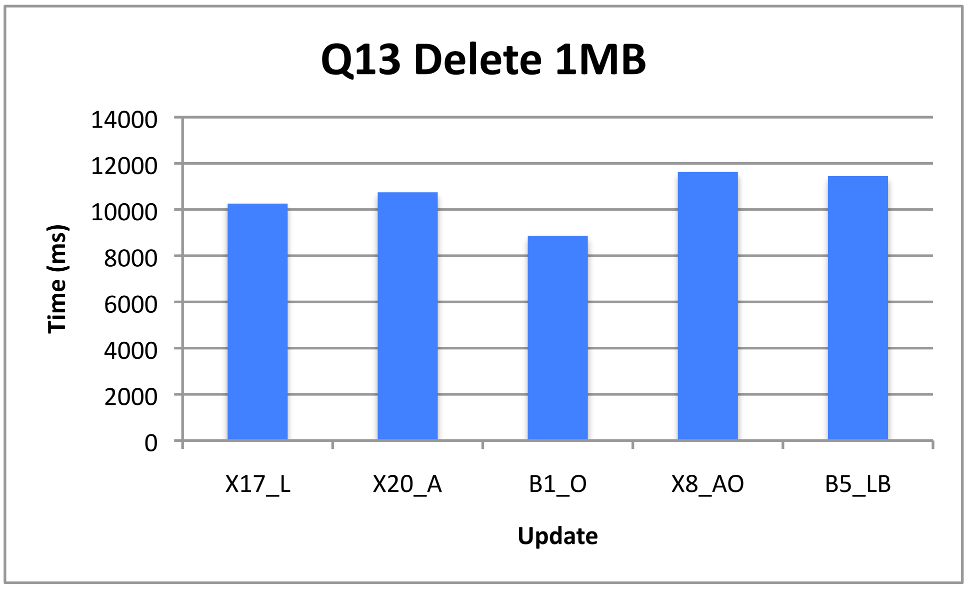


Figure 5.10: Q13 Delete Times Detail (upper) and Summary (lower)

Update	Q1	Q2	Q3	Q4	Q6	Q13	Q17
X1_L	4475						7045
X6_A	3450						5291
A7_O	4851						7731
A8_AO	5994						10820
B7_LB	3833						5748
X2_L		8179	6778	10552			
X3_A		6354	5805	8379			
X4_O		9012	7498	12108			
X5_AO		8913	7541	13680			
B3_LB		6490	6041	8842			
E6_L					9425		
B1_A					8477		
X7_O					9979		
X8_AO					9993	11627	
B5_LB					9418	11446	
X17_L						10258	
E5_A						10747	
B1_O						8861	

Table 5.12: Total Running Time (ms) for all Views and Queries

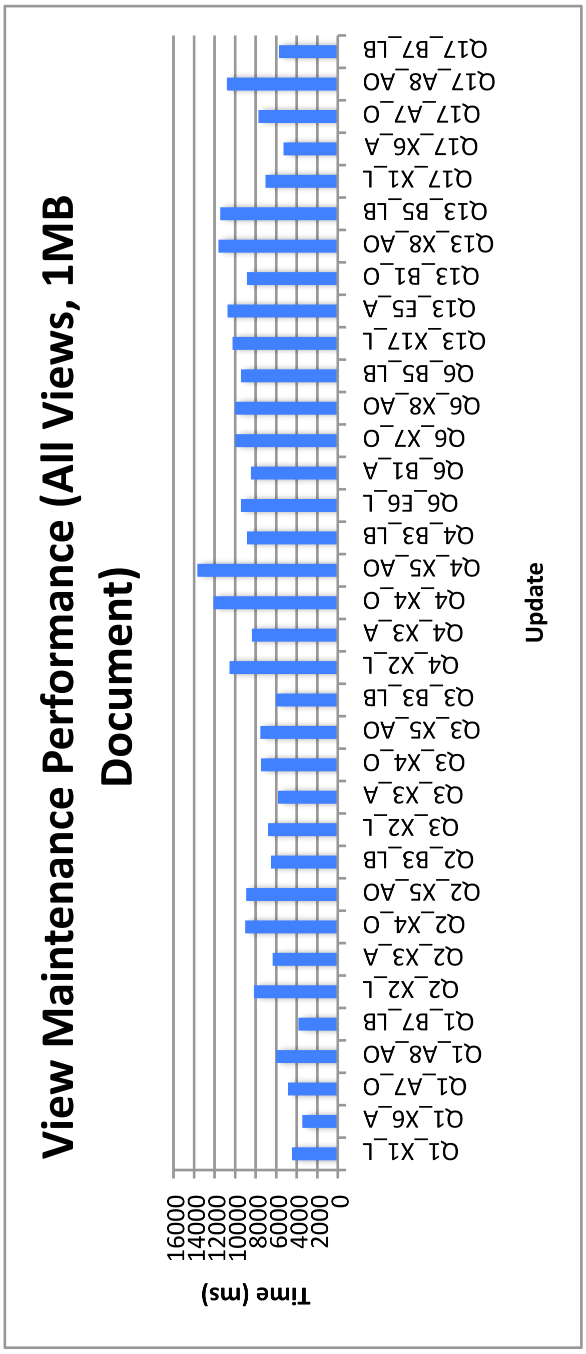
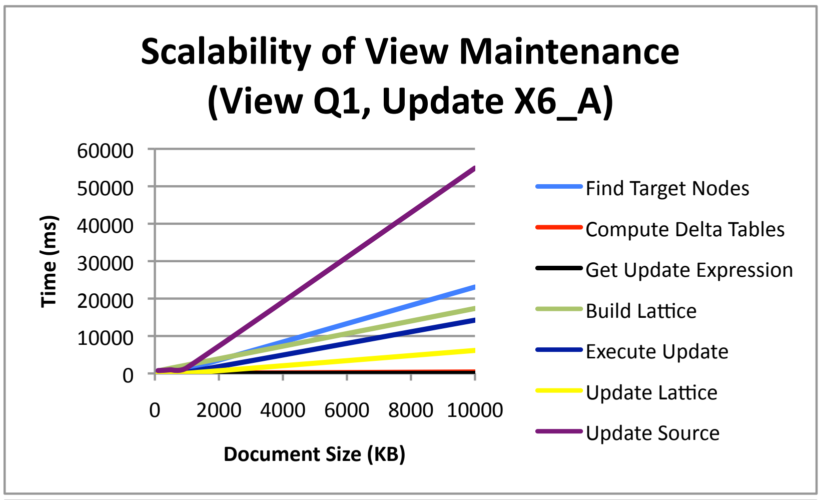


Figure 5.11: Total Running Time for all Views and Queries

	Document Size (KB)			
Stage	**100**	**500**	**1000**	**10000**
Find Target Nodes	260	754	1172	23086
Compute Delta Tables	11	31	54	398
Get Update Expression	7	10	9	17
Build Lattice	676	1407	2243	17360
Execute Update	209	257	337	14236
Update Lattice	6	18	50	6148
Update Source	767	973	1525	54872

TABLE 5.13: Scalability Q1 Delete Update X6_A (ms)



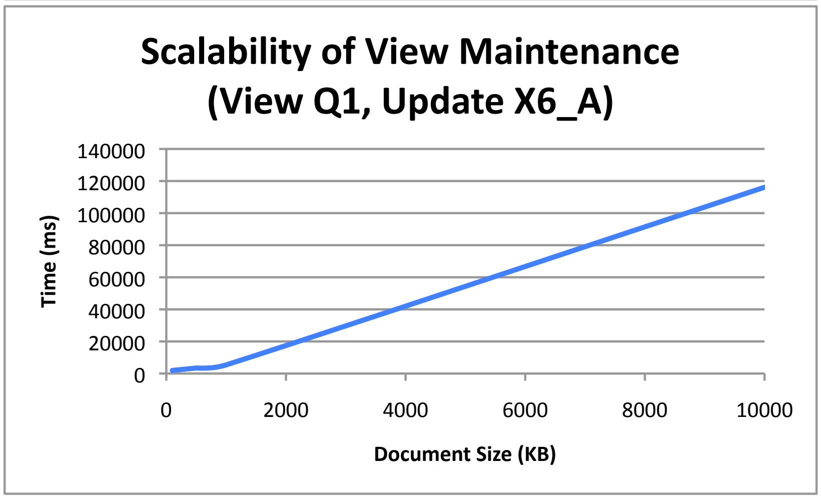


FIGURE 5.12: Scalability Q1 Delete Update X6_A

Query	TN	DT	UE	BL	EU	UL	US
/site	38590	774	10	2349	2337	221	674
/site/people	2173	177	11	2288	2011	125	1334
/site/people/person	2647	178	14	2265	874	114	1370
/site/people/person/name	921	24	8	2329	672	42	1528

Table 5.14: Varying Path for XMark View Q1 and Update X1_L for 1MB Document

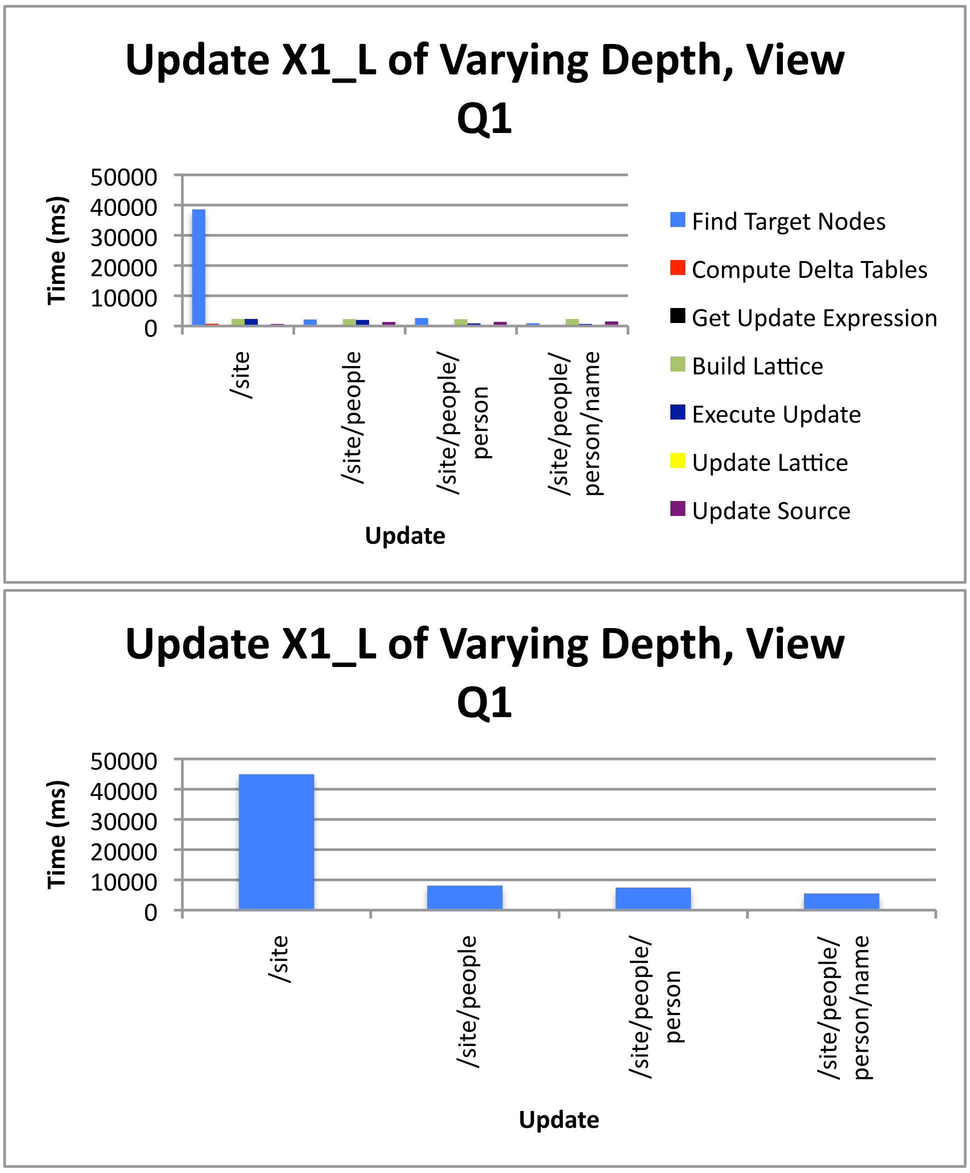
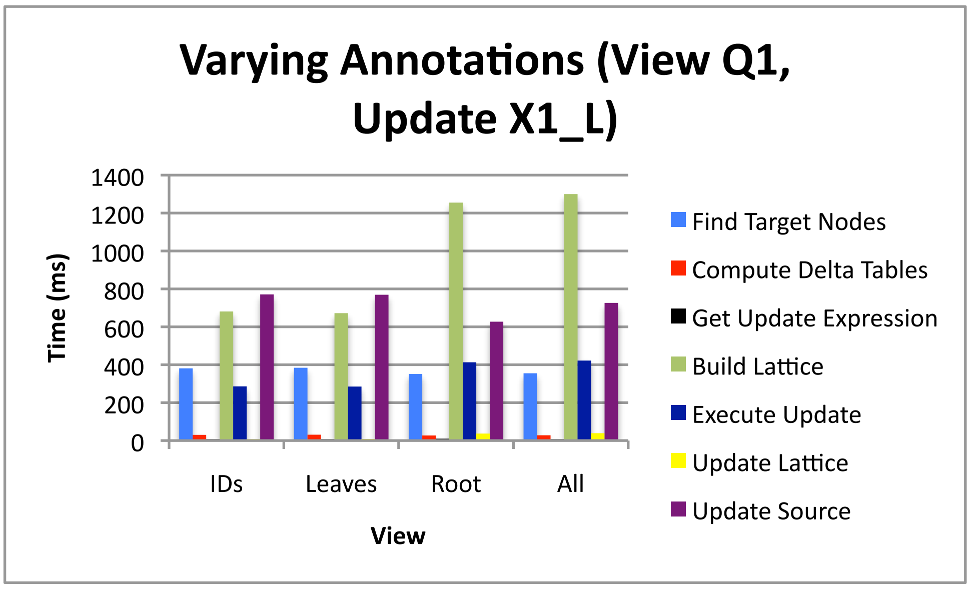


Figure 5.13: Varying Path for XMark View Q1 and Update X1_L for 1MB Document

View	TN	DT	UE	BL	EU	UL	US
IDs	381	30	7	681	286	7	771
Leaves	384	31	7	672	285	8	769
Root	351	27	10	1255	413	37	627
All	355	28	7	1300	422	39	726

TABLE 5.15: Varying Annotations for XMark View Q1 and Update X1_L for 100KB document



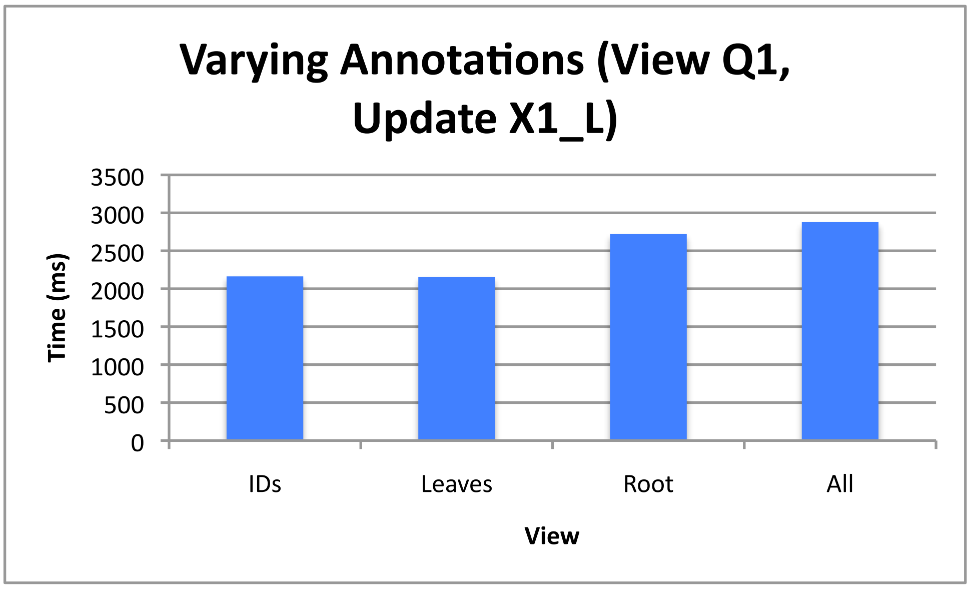


FIGURE 5.14: Varying Annotations for XMark View Q1 and Update X1_L for 100KB document

5.3 View Update

For view update, the testing approach had to be modified. The reason for this being that queries are expressed and evaluated based on the data contained within the views. Whereas with view maintenance, the update propagates from the document to the view. Therefore, if filters were used on branches, they would be irrelevant, as from the view definition it is known that all tuples stored contain all view branches. For example, for the view /site/people/person[@id][name] and the query /site/people/person, the results are the same as the view definition. The alternative, to express filters on values (vals), generally only returned one tuple. This was due to the majority of vals generated by [158] being IDs, and therefore, unique. A single affected tuple was not enough to produce meaningful results. Therefore, the approach to the varying path experiments for view maintenance was taken for testing insertions and deletions, respectively, using the same XML source documents and views as described in Section 5.2. The times measured remained the same.

Comparable experiments to view maintenance are performed. The only exception is that there is no varying path experiment. This is because it would be the same as the component times experiment due to the same data set being used.

5.3.1 Insertions

The performance results for insertions on the views $Q1$, $Q2$ and $Q13$ for document size 1MB are shown in Tables 5.16, 5.17, 5.18 and Figures 5.15, 5.16 and 5.17 respectively. The graphs show the results for the individual times as well as the total times. The results for the other views can be seen in Appendix C.2.1.

Query	TN	DT	UE	BL	EU	UL	US
X1_L	944	9	15	880	349	422	1707
X1_L_2	975	9	12	824	328	277	1659
X1_L_3	1420	665	13	849	472	229	1703

TABLE 5.16: Q1 View Insert Time for 1MB Document (ms)

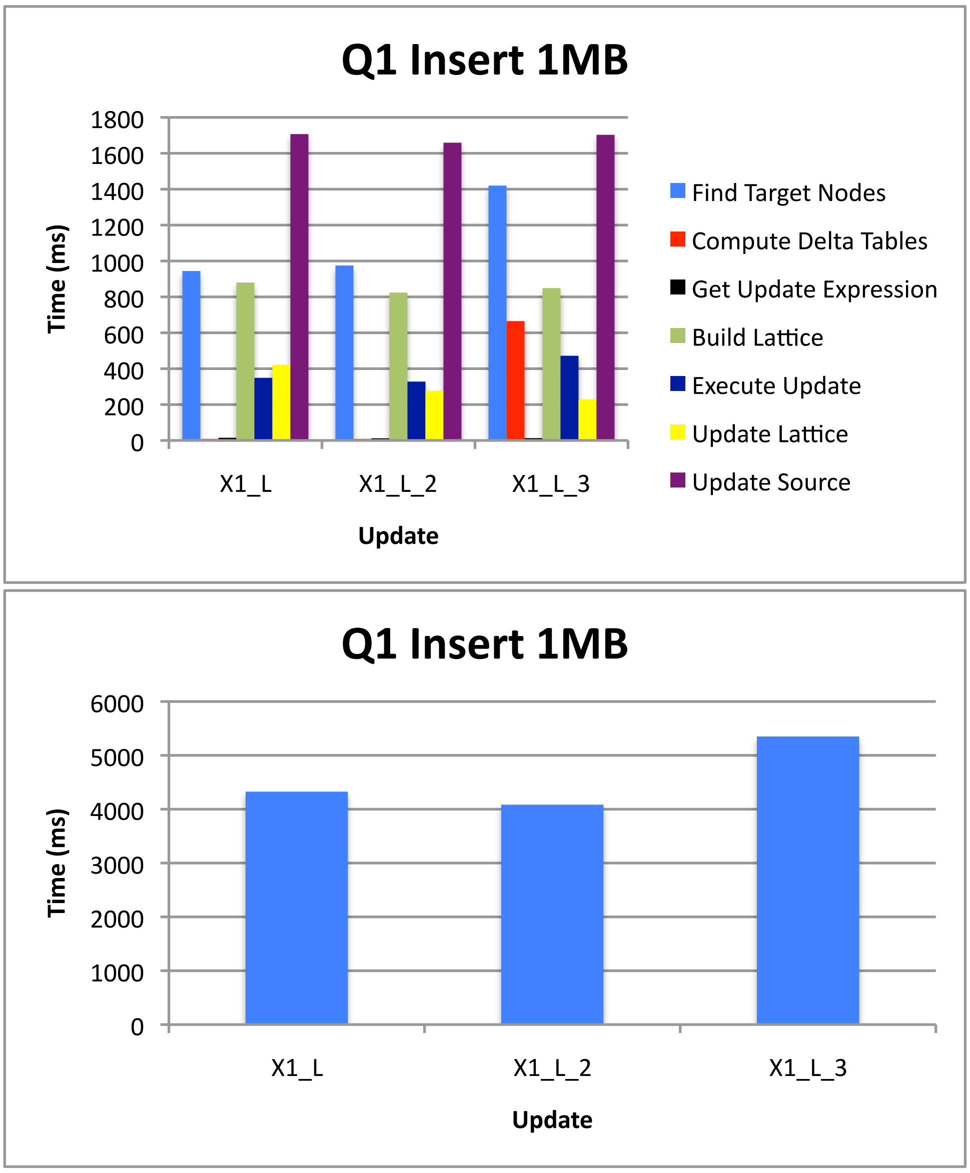
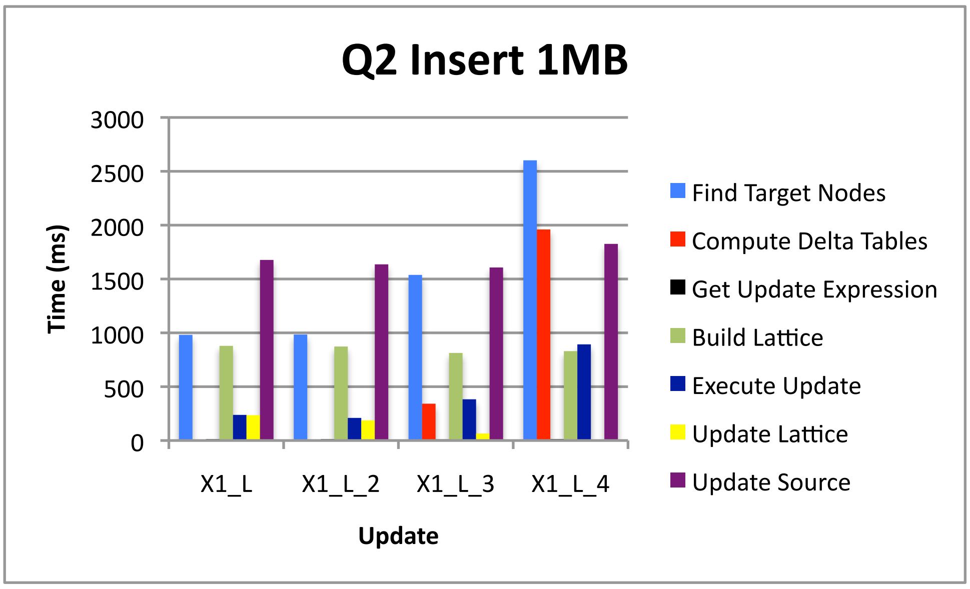


FIGURE 5.15: Q1 View Insert Times Detail (upper) and Summary (lower)

Query	TN	DT	UE	BL	EU	UL	US
X1_L	981	9	13	879	238	235	1677
X1_L_2	984	9	13	873	210	187	1636
X1_L_3	1538	342	11	813	383	65	1607
X1_L_4	2602	1960	14	830	893	0	1826

Table 5.17: Q2 View Insert Time for 1MB Document (ms)



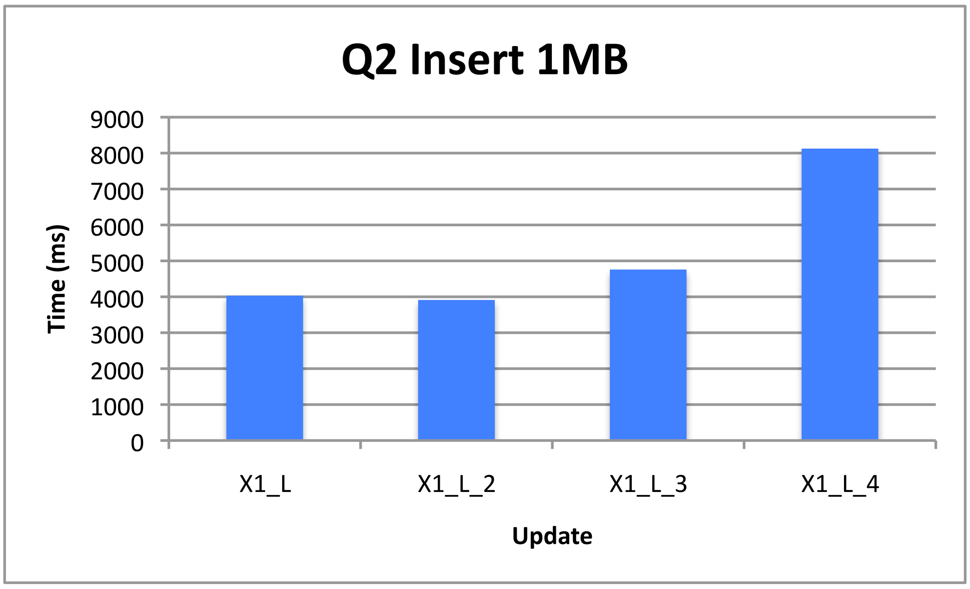
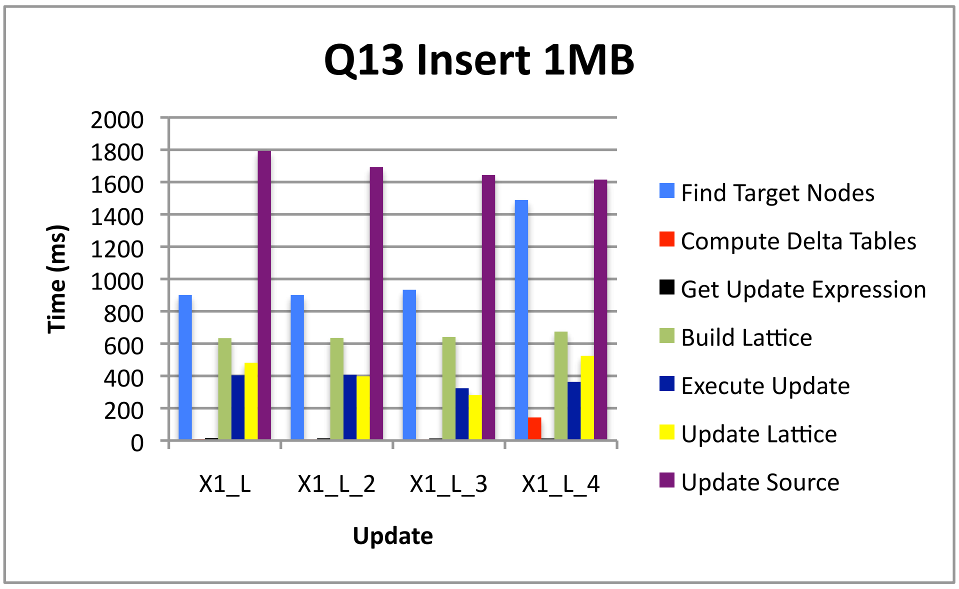


Figure 5.16: Q2 View Insert Times Detail (upper) and Summary (lower)

Query	TN	DT	UE	BL	EU	UL	US
X1_L	901	9	15	634	404	481	1793
X1_L_2	901	7	14	635	407	398	1693
X1_L_3	933	7	13	641	324	282	1644
X1_L_4	1489	143	13	674	363	524	1615

TABLE 5.18: Q13 View Insert Time for 1MB Document (ms)



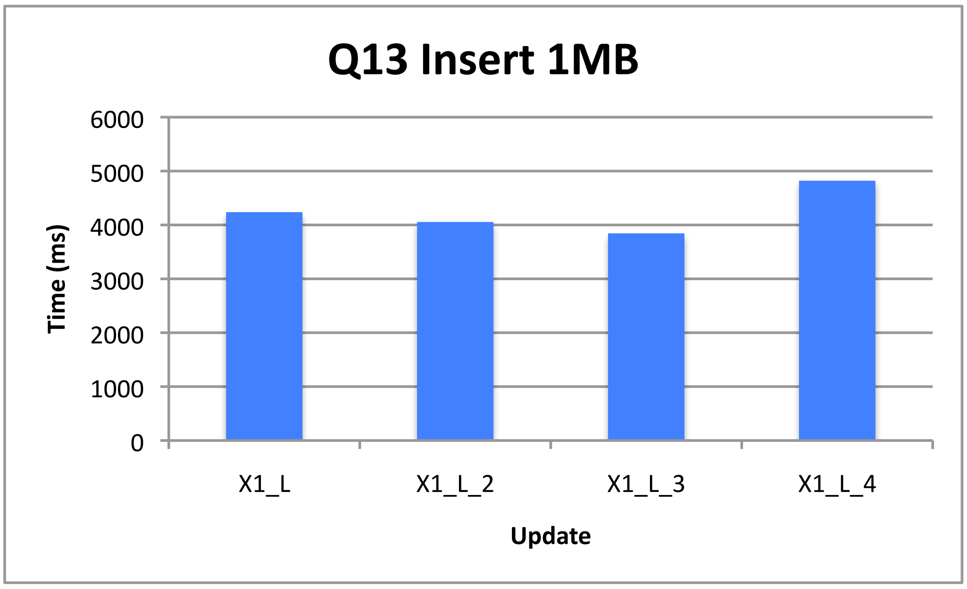


FIGURE 5.17: Q13 View Insert Times Detail (upper) and Summary (lower)

Total Times

The total times for all the views and queries used for testing are shown in Table 5.19 and Figure 5.18.

Scalability

A scalability test was performed to check the performance of the algorithms on larger numbers of view tuples. The number of view tuples ranged from 25 to 2550. The results are shown in Table 5.20 and Figure 5.19.

Varying Annotations

Experiments were performed to determine how different combinations of stored attributes on the view nodes affect the performance. For this experiment, update $X1\_L$ and view $Q1$ along with variations were used. For each node the ID is always stored. The different combinations used were: (1) only IDs stored; (2) val and cont only stored for the leaves; (3) val and cont only stored for the root; and (4) val and cont stored for every node. The results can be seen in Table 5.21 and Figure 5.20.

Update	Q1	Q2	Q3	Q4	Q6	Q13	Q17
X1_L	4326	4032	3535	3902	2884	4237	4011
X1_L_2	4084	3912	3513	3780	2749	4055	3791
X1_L_3	5351	4759	3754	3652	4295	3844	4341
X1_L_4		8125	3550			4821	

TABLE 5.19: Total Running Time (ms) for all Views and Queries

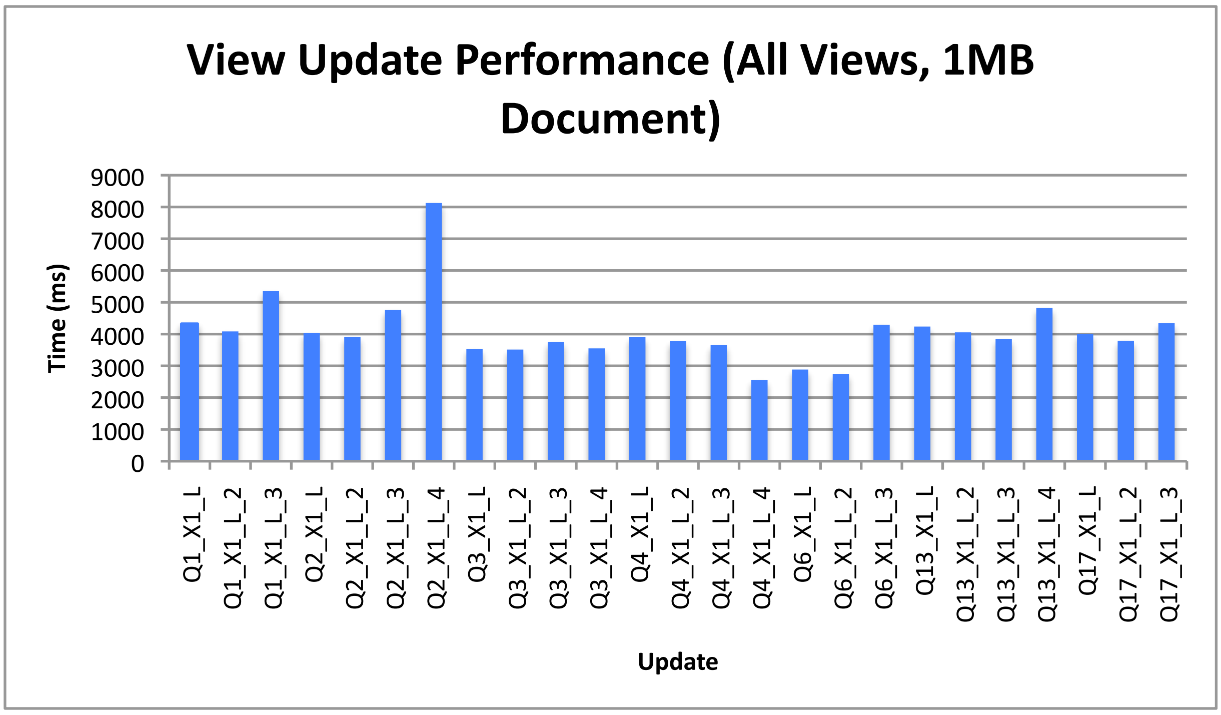
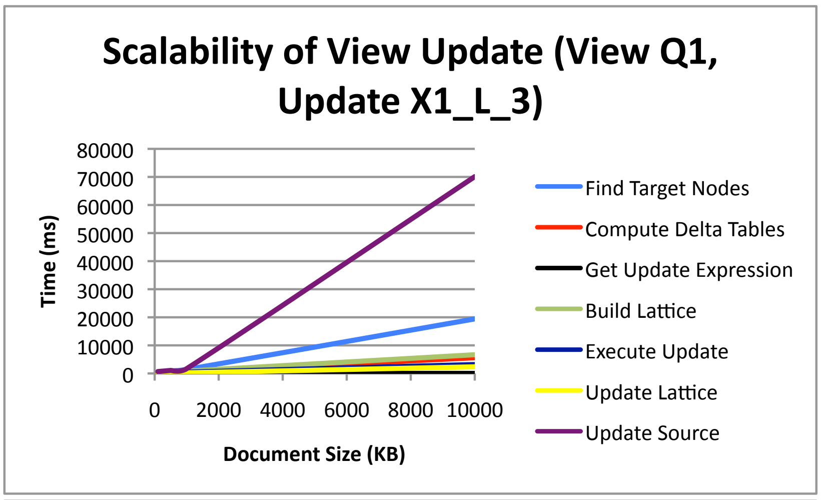


FIGURE 5.18: Total Running Time for all Views and Queries

	Document Size (KB)			
Stage	**100**	**500**	**1000**	**10000**
Find Target Nodes	299	901	1420	19428
Compute Delta Tables	100	346	665	5588
Get Update Expression	20	14	13	15
Build Lattice	494	630	849	6670
Execute Update	196	351	472	3172
Update Lattice	107	274	229	2366
Update Source	685	1078	1703	70111

Table 5.20: Scalability Q1 View Insert Update X1_L_3 (ms)



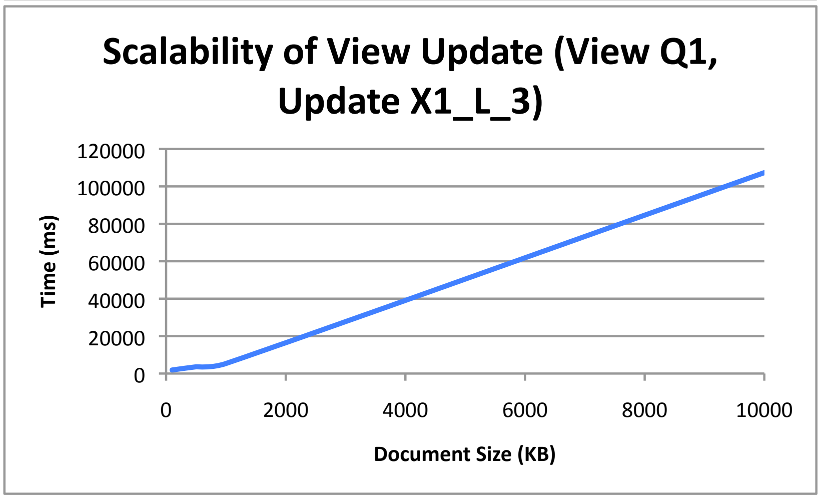


Figure 5.19: Scalability Q1 View Insert Update X1_L_3

View	TN	DT	UE	BL	EU	UL	US
IDs	308	102	16	498	194	109	676
Leaves	302	103	13	494	193	110	676
Root	391	81	24	1141	1920	363	159
All	389	82	12	1151	2157	284	151

TABLE 5.21: Varying Annotations for XMark View Q1 and Update X1_L_3 for 100KB Document

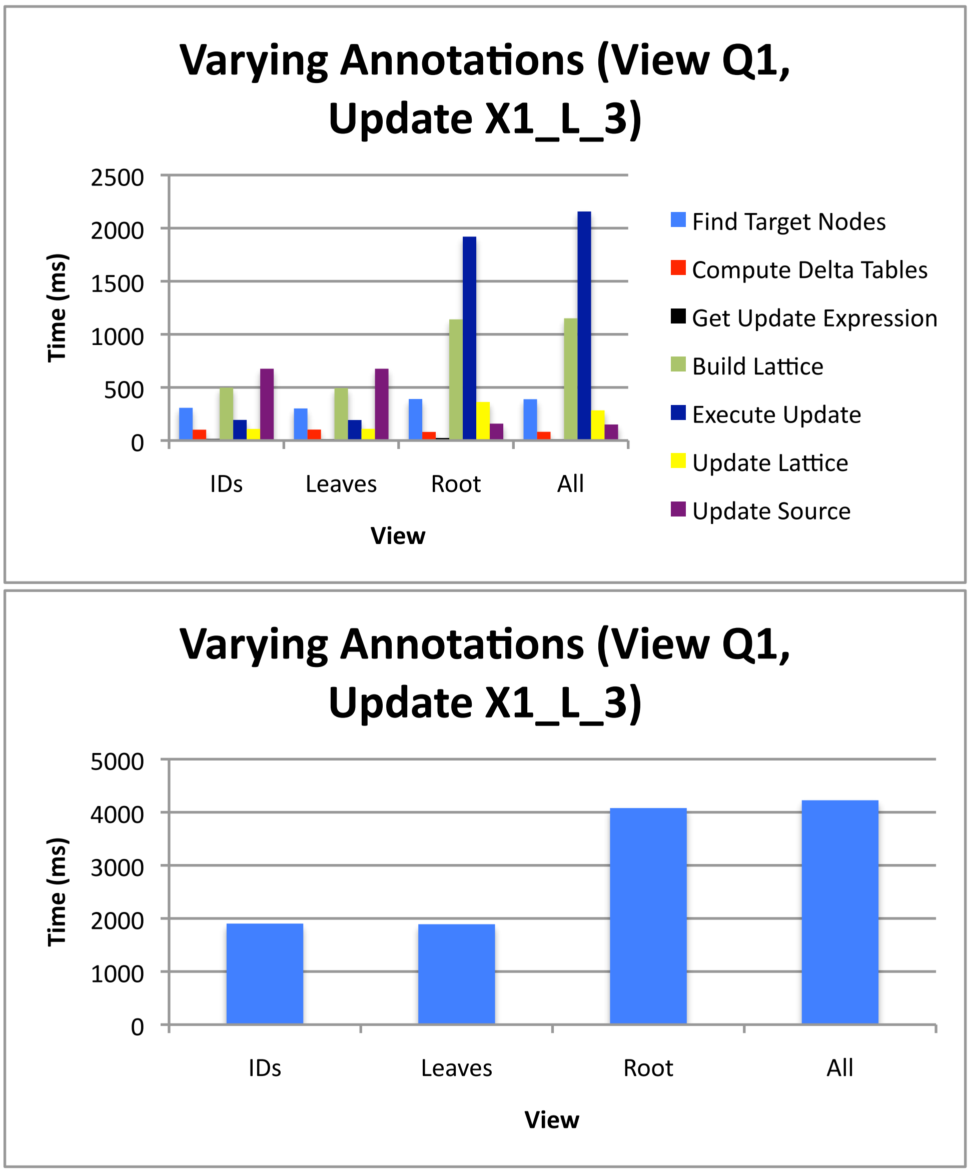


FIGURE 5.20: Varying Annotations for XMark View Q1 and Update X1_L_3 for 100KB Document

5.3.2 Deletions

The performance results for deletions on the views $Q1$, $Q2$ and $Q13$ for document size 1MB are shown in Tables 5.22, 5.23, 5.24 and Figures 5.21, 5.22 and 5.23 respectively. The graphs show the results for the individual times as well as the total times. The other results can be seen in Appendix C.2.2.

Total Times

The total times for all the views and queries used for testing are shown in Table 5.25 and Figure 5.24.

Scalability

A scalability test was performed to check the performance of the algorithms on larger numbers of view tuples. The number of view tuples ranged from 25 to 2550. The results are shown in Table 5.26 and Figure 5.25.

Varying Annotations

Experiments were performed to determine how different combinations of stored attributes on the view nodes affect the performance. For this experiment, update $X1\_L$ and view $Q1$ along with variations were used. For each node the ID is always stored. The different combinations used were: (1) only IDs stored; (2) val and cont only stored for the leaves; (3) val and cont only stored for the root; and (4) val and cont stored for every node. The results can be seen in Table 5.27 and Figure 5.26.

Query	TN	DT	UE	BL	EU	UL	US
DeleteX1_L	150	31	14	831	1671	6831	691
DeleteX1_L_2	248	56	24	1235	2445	5499	1904
DeleteX1_L_3	140	24	15	836	1008	94	1360

TABLE 5.22: Q1 View Delete Time for 1MB Document (ms)

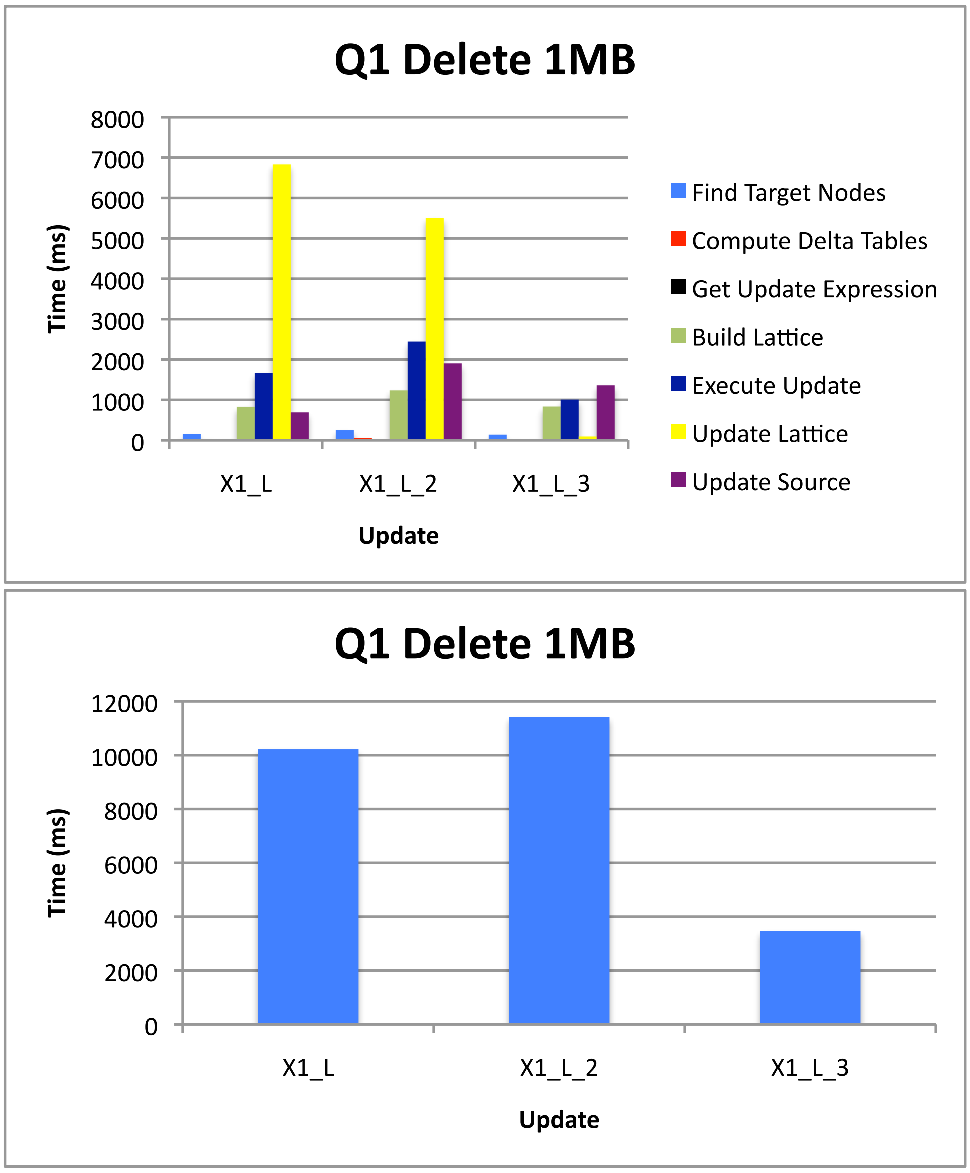
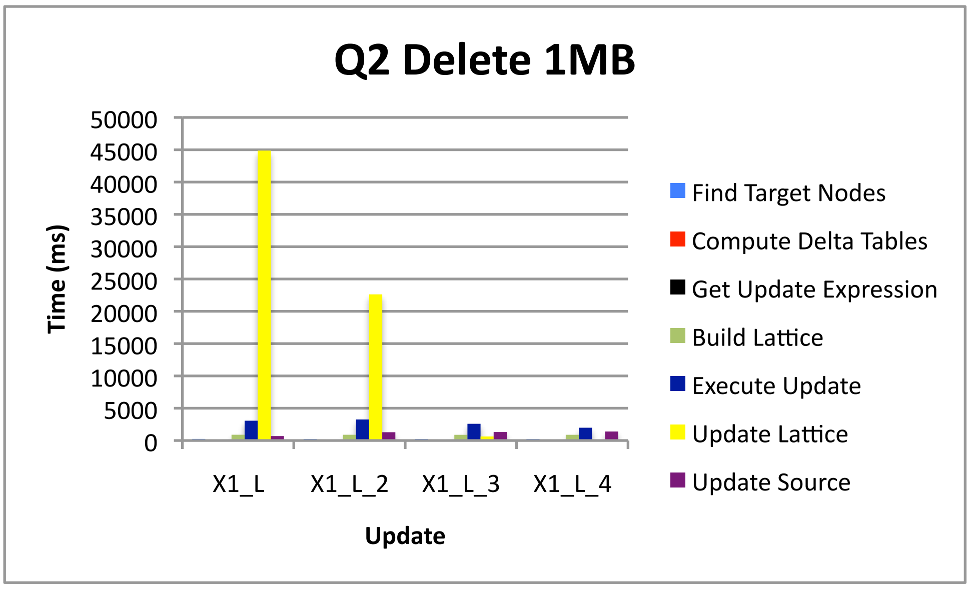


FIGURE 5.21: Q1 View Delete Times Detail (upper) and Summary (lower)

Query	TN	DT	UE	BL	EU	UL	US
DeleteX1_L	246	74	14	893	3065	44873	688
DeleteX1_L_2	241	72	15	890	3259	22621	1286
DeleteX1_L_3	239	61	15	881	2587	629	1313
DeleteX1_L_4	228	43	16	885	1979	149	1397

Table 5.23: Q2 View Delete Time for 1MB Document (ms)



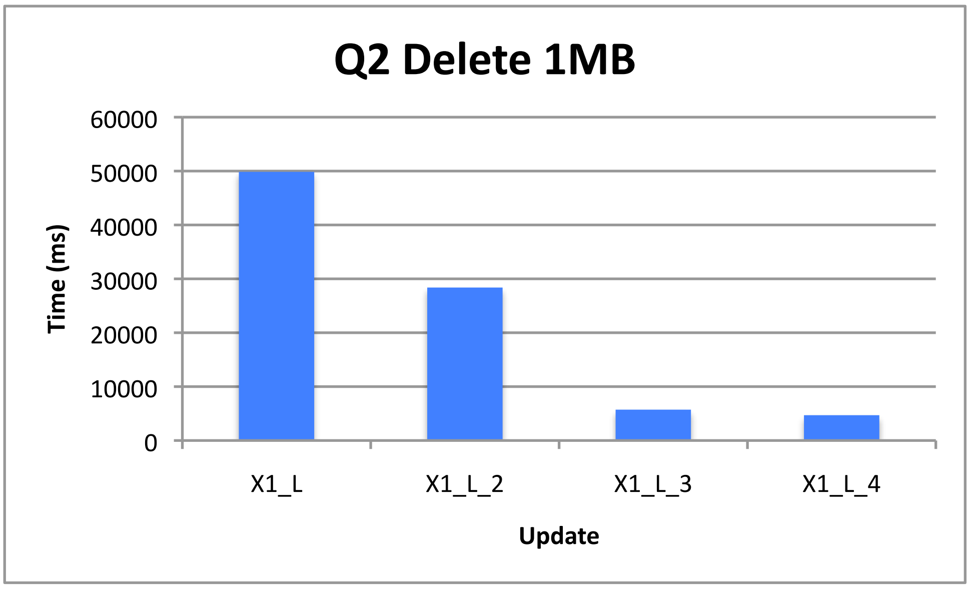
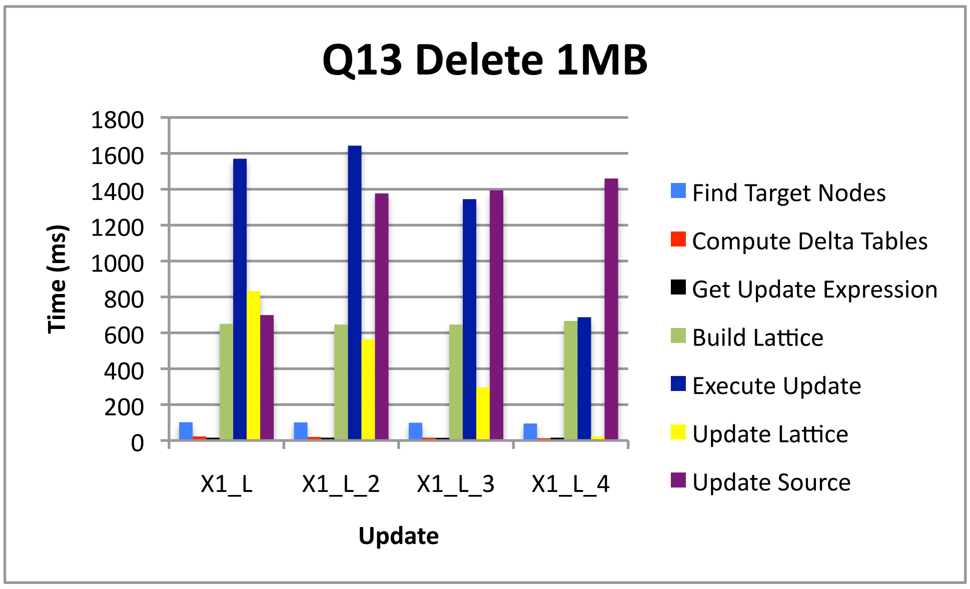


Figure 5.22: Q2 View Delete Times Detail (upper) and Summary (lower)

Query	TN	DT	UE	BL	EU	UL	US
DeleteX1_L	102	23	16	650	1570	832	699
DeleteX1_L_2	101	20	16	646	1643	563	1377
DeleteX1_L_3	99	16	15	646	1345	296	1394
DeleteX1_L_4	95	13	16	666	687	24	1460

TABLE 5.24: Q13 View Delete Time for 1MB Document (ms)



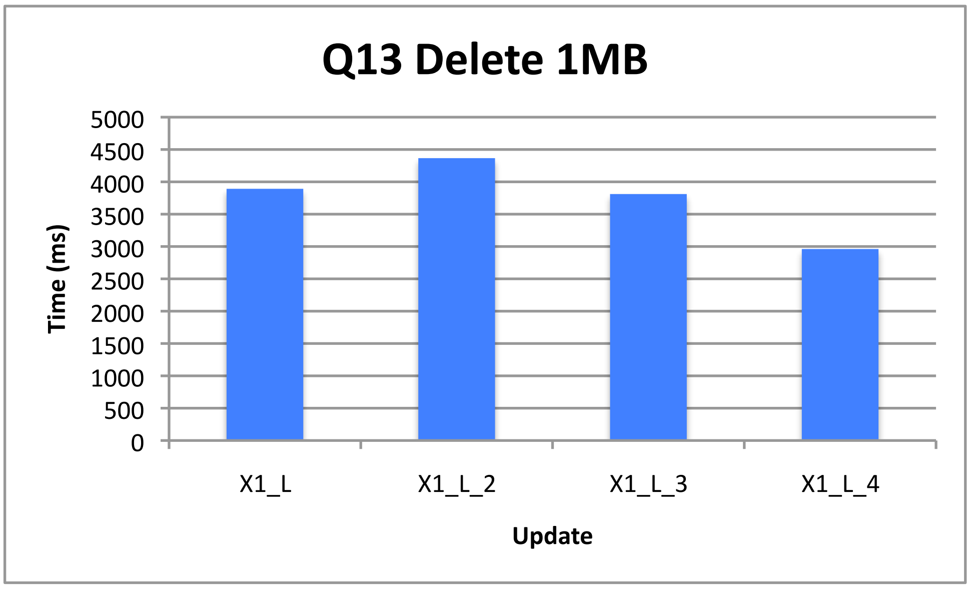


FIGURE 5.23: Q13 View Delete Times Detail (upper) and Summary (lower)

Update	Q1	Q2	Q3	Q4	Q6	Q13	Q17
DeleteX1_L	10219	49853	2072	1987	1838	3892	3245
DeleteX1_L_2	11411	28384	2764	2762	2629	4366	3783
DeleteX1_L_3	3477	5725	2533	2702	2368	3811	2904
DeleteX1_L_4		4697	2413			2961	

Table 5.25: Total Running Time (ms) for all Views and Queries

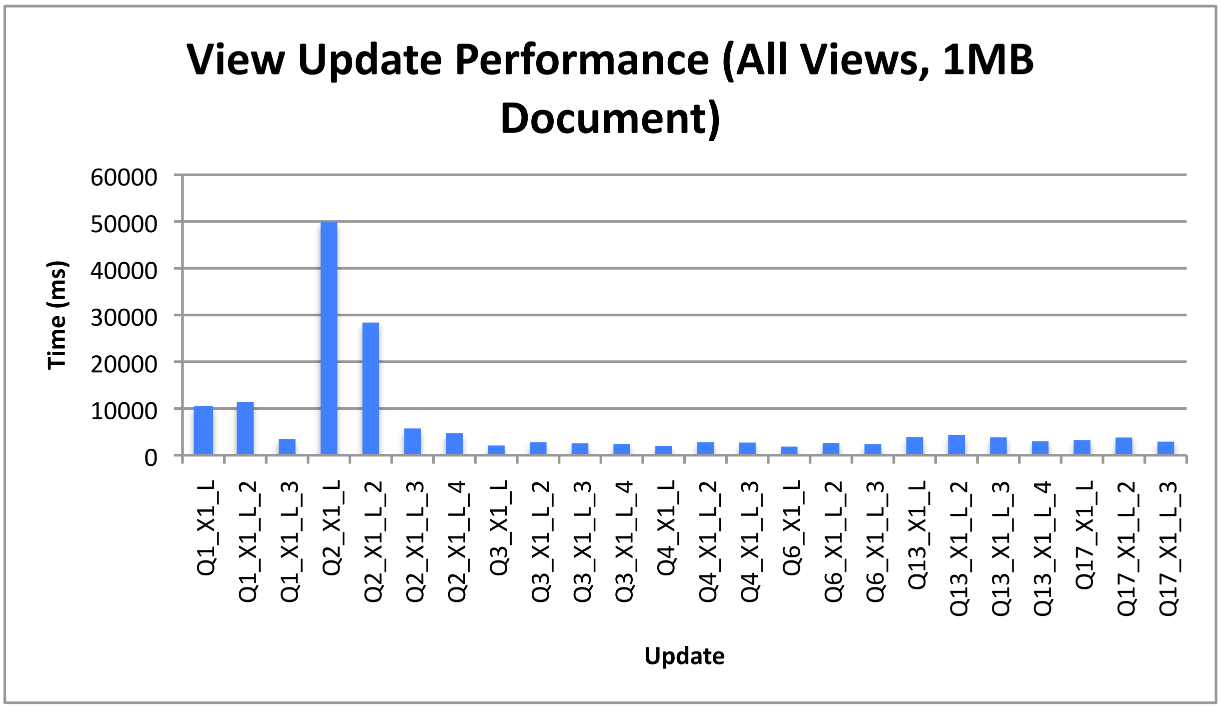
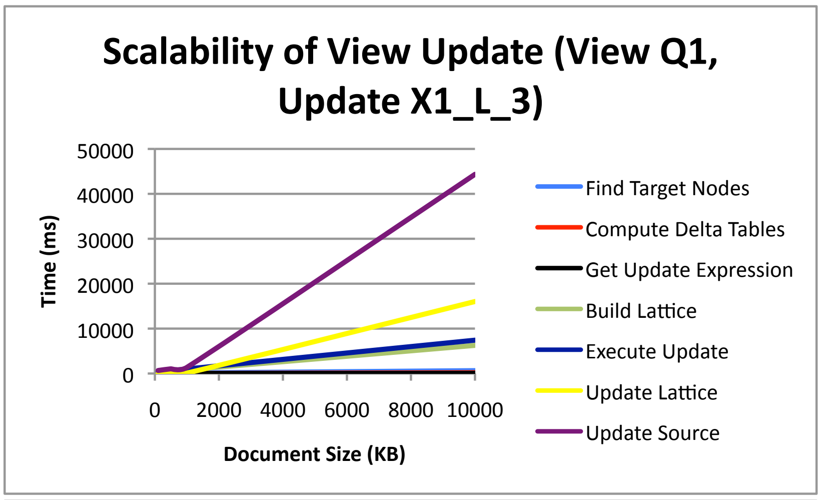


Figure 5.24: Total Running Time for all Views and Queries

	Document Size (KB)			
Stage	**100**	**500**	**1000**	**10000**
Find Target Nodes	53	101	140	639
Compute Delta Tables	4	17	24	186
Get Update Expression	13	17	15	45
Build Lattice	565	711	836	6264
Execute Update	293	620	1008	7423
Update Lattice	4	43	94	16031
Update Source	672	1042	1360	44349

TABLE 5.26: Scalability Q1 View Delete Update X1_L_3 (ms)



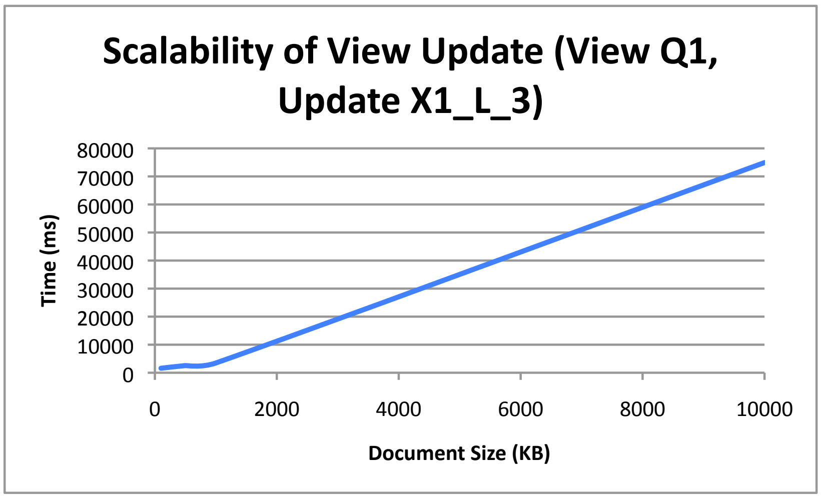


FIGURE 5.25: Scalability Q1 View Delete Update X1_L_3

View	TN	DT	UE	BL	EU	UL	US
IDs	53	5	13	551	289	4	667
Leaves	54	4	14	552	301	4	662
Root	205	4	15	1038	305	36	721
All	205	4	13	1058	434	38	642

Table 5.27: Varying Annotations for XMark View Q1 and Update X1_L_3 for 100KB document

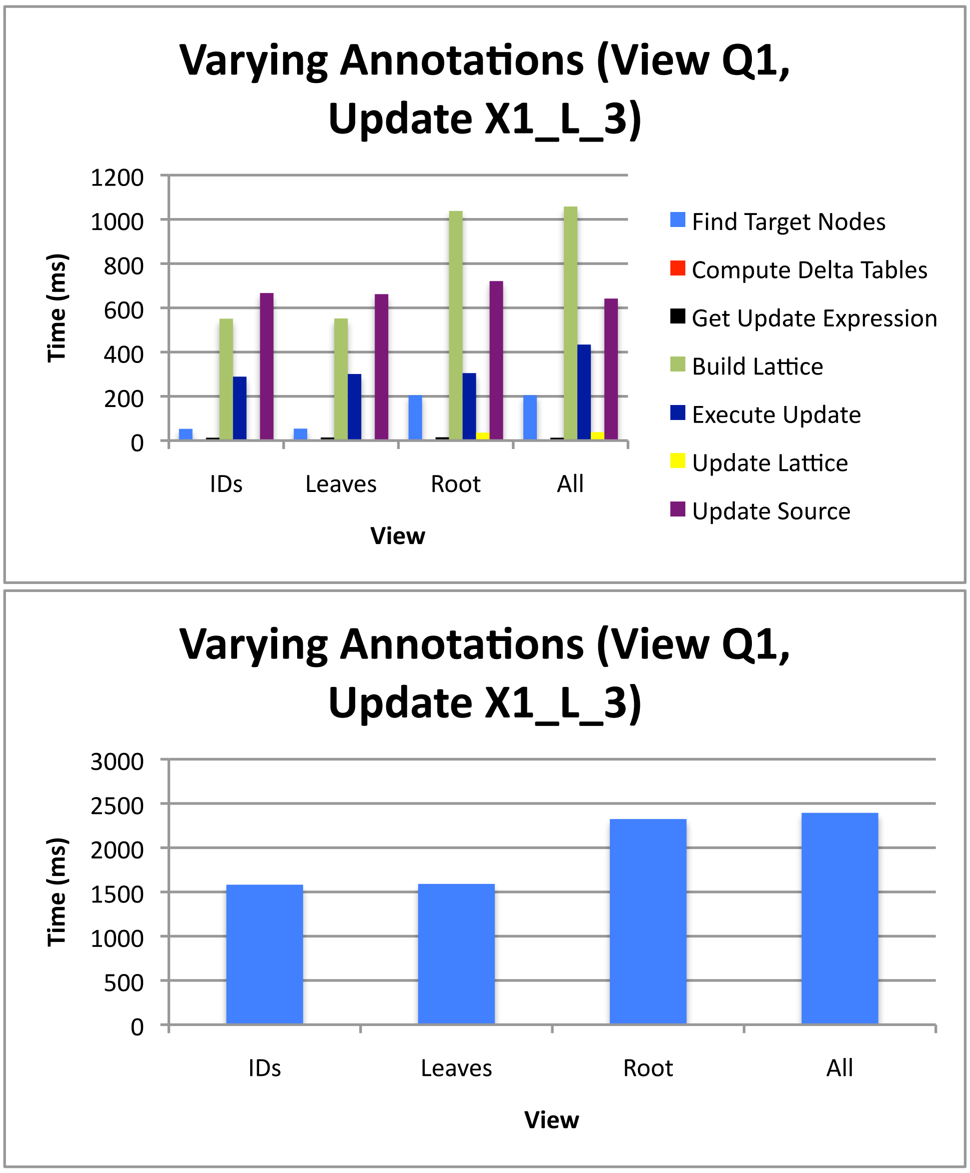


Figure 5.26: Varying Annotations for XMark View Q1 and Update X1_L_3 for 100KB document

5.4 View Maintenance vs View Update

The performance of the view maintenance vs the view update method was explored. These tests were on view Q1 using query X1_L. The results can be seen in Table 5.28 and Figure 5.27.

	Document Size (KB)			
Stage	**100**	**500**	**1000**	**10000**
VMIns	2112	4120	6518	120700
VMDel	2126	4475	7462	273427
VUIns	1901	3594	5351	107350
VUDel	1604	2551	3477	74937

TABLE 5.28: View Maintenance vs View Update X1_L_3 (ms)

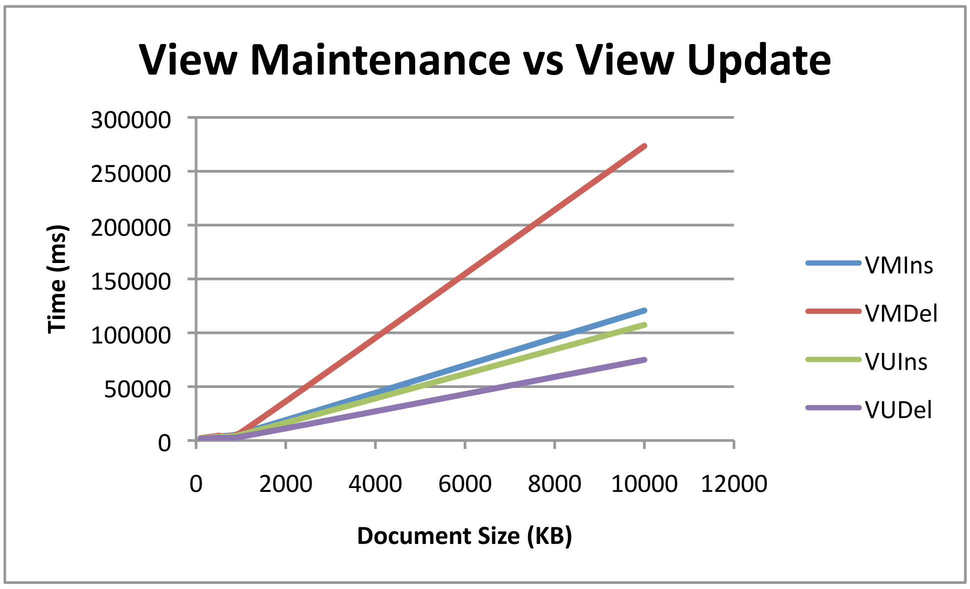


FIGURE 5.27: View Maintenance vs View Update X1_L_3

5.5 Optimisations

The utilisation of optimisation rules for dynamic reasoning on sequences of updates was tested with regards to how they affected performance. Testing was performed for the optimisation rules O1, O3 and I5. These were performed using view Q1 and update X1_L alongside another query which would allow pruning. This query was changed between tests to increase the amount of pruning performed. The results for these tests, compared with the same tests using no optimisation can be seen in Tables 5.29, 5.31 and 5.33 and Figures 5.28, 5.30 and 5.32. The tests that allow 100% pruning are also performed for document sizes 500KB - 10MB for view maintenance and view update and compared with the same tests without optimisation. These results can be seen in Tables 5.30, 5.32, 5.34, 5.35, 5.36 and 5.37 and Figures 5.29, 5.31, 5.33, 5.34, 5.35 and 5.36.

	Pruning Percentage				
	20%	40%	60%	80%	100%
Optimise	2423	2436	2558	2504	2297
No Optimise	2749	2795	3004	3139	2905

Table 5.29: Performance for Reduction Rule O1 (ms)

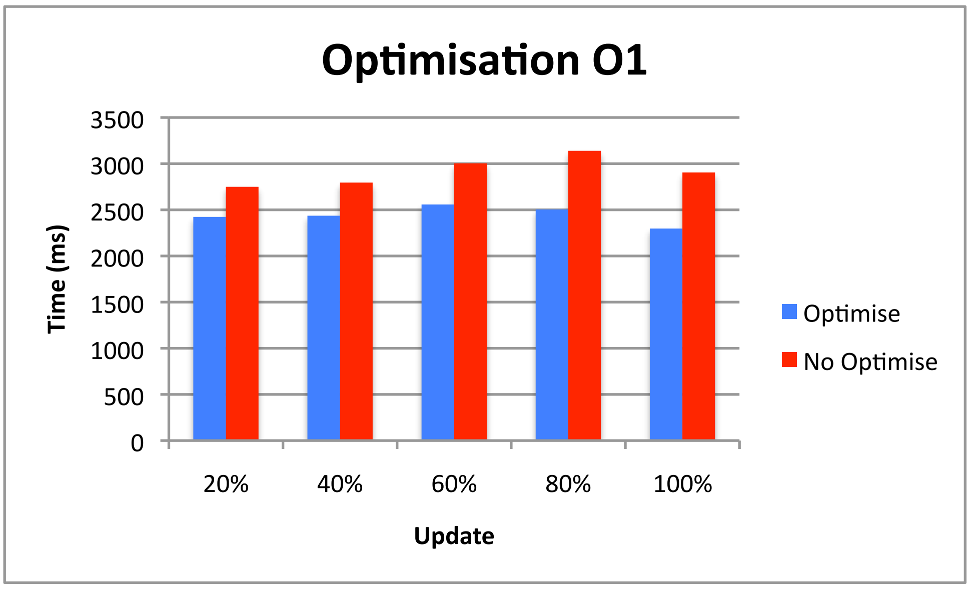


Figure 5.28: Performance for Reduction Rule O1

	Document Size (KB)			
	100	**500**	**1000**	**10000**
Optimise	2297	5210	9358	359007
No Optimise	2905	7104	13370	628898

TABLE 5.30: Scalability for Reduction Rule O1 (ms)

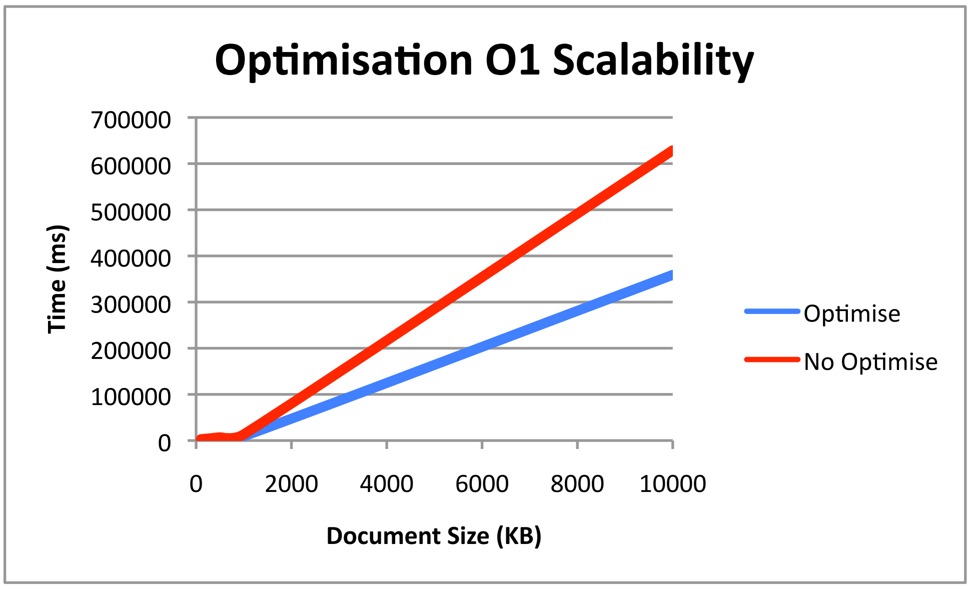


FIGURE 5.29: Scalability for Reduction Rule O1

	Pruning Percentage				
	20%	40%	60%	80%	100%
Optimise	2620	2654	2958	2992	2687
No Optimise	2598	2673	2717	2757	2857

Table 5.31: Performance for Reduction Rule O3 (ms)

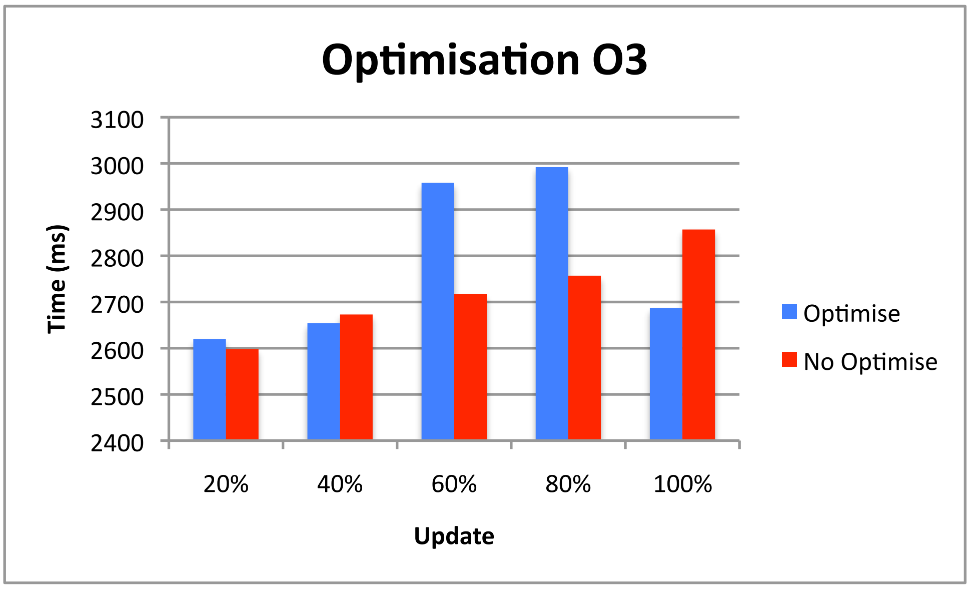


Figure 5.30: Performance for Reduction Rule O3

	Document Size (KB)			
	100	**500**	**1000**	**10000**
Optimise	2687	4973	8551	317257
No Optimise	2857	5368	9113	346458

TABLE 5.32: Scalability for Reduction Rule O3 (ms)

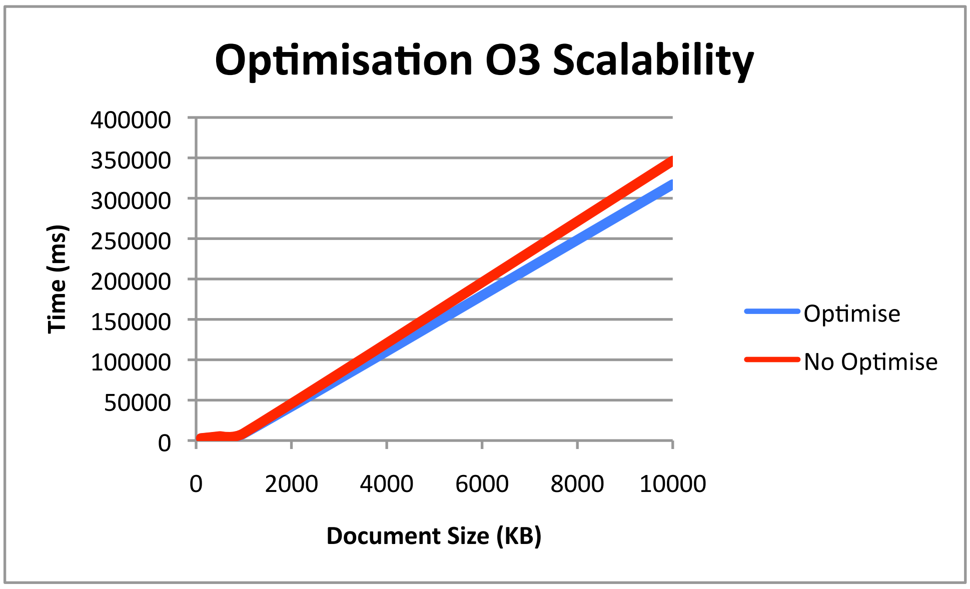


FIGURE 5.31: Scalability for Reduction Rule O3

	Pruning Percentage				
	20%	40%	60%	80%	100%
Optimise	2453	2518	2593	2557	2370
No Optimise	2493	2511	2561	2713	2620

Table 5.33: Performance for Reduction Rule I5 (ms)

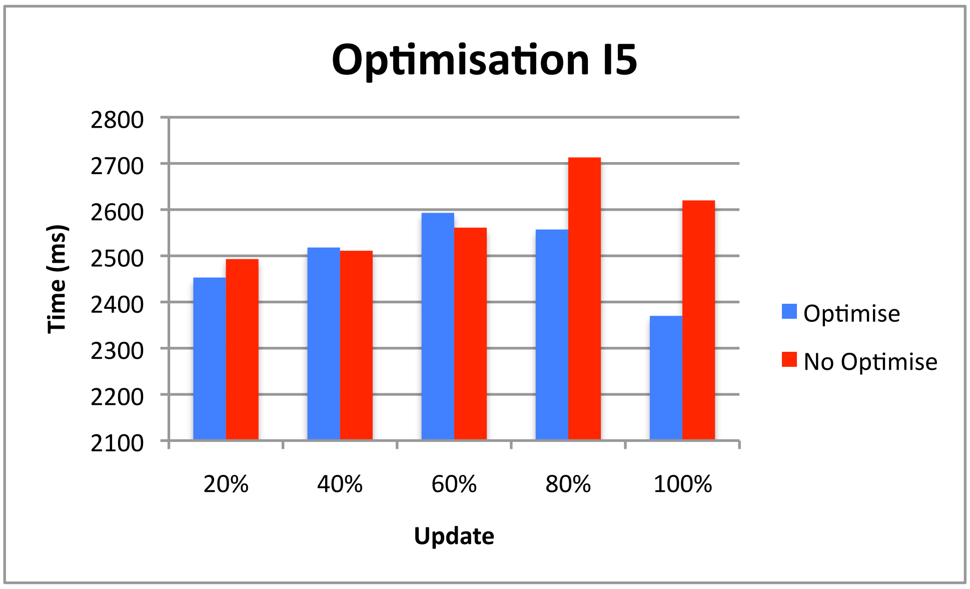


Figure 5.32: Performance for Reduction Rule I5

	Document Size (KB)			
	100	**500**	**1000**	**10000**
Optimise	2370	5370	9023	160284
No Optimise	2620	5812	9924	232481

Table 5.34: Scalability for Reduction Rule I5 (ms)

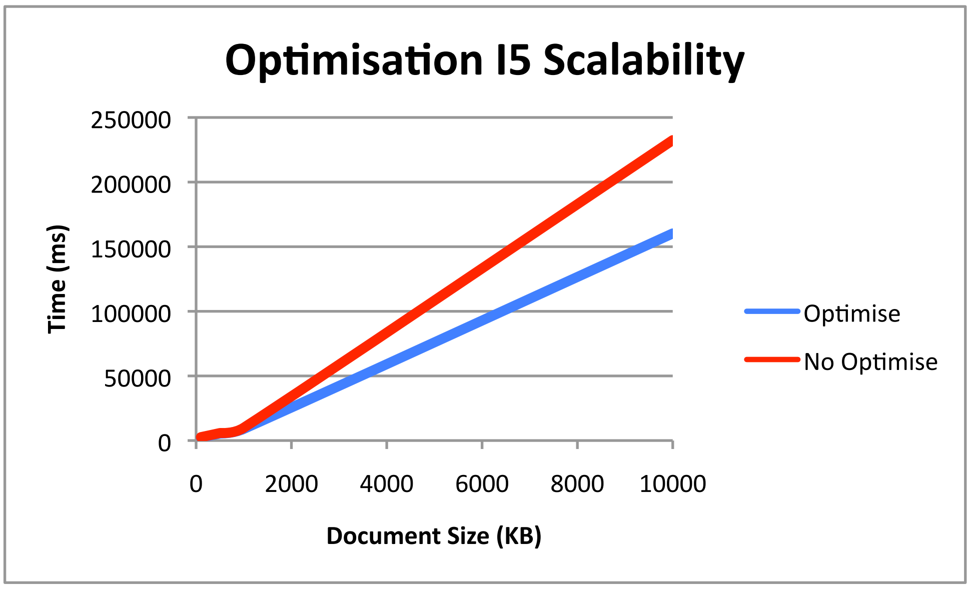


Figure 5.33: Scalability for Reduction Rule I5

	Document Size (KB)			
	100	**500**	**1000**	**10000**
Optimise	1885	3387	5086	114412
No Optimise	2519	5066	8866	326646

Table 5.35: Scalability for Reduction Rule O1 (ms)

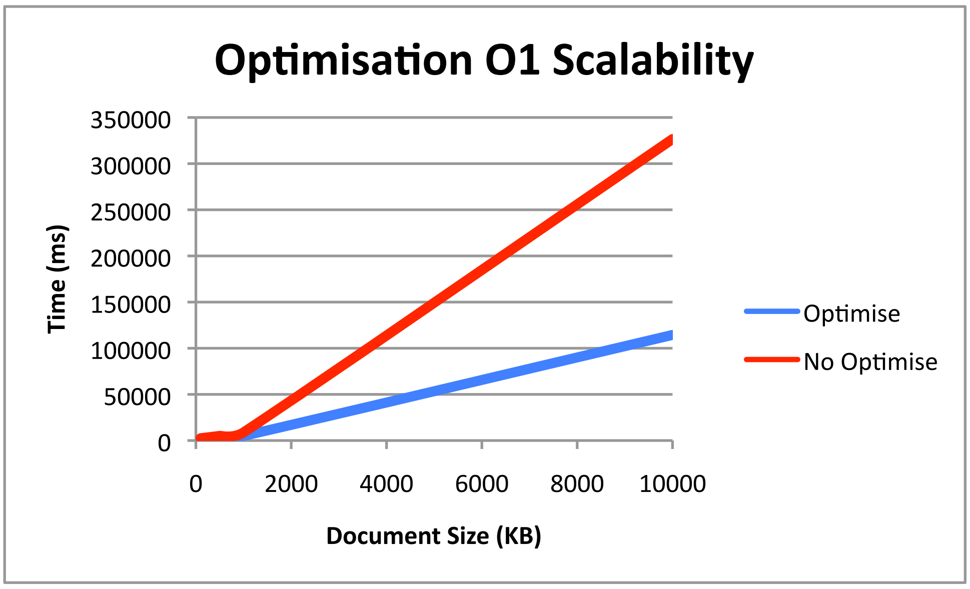


Figure 5.34: Scalability for Reduction Rule O1

	Document Size (KB)			
	100	**500**	**1000**	**10000**
Optimise	1855	2723	3794	119720
No Optimise	2137	2674	3700	113217

TABLE 5.36: Scalability for Reduction Rule O3 (ms)

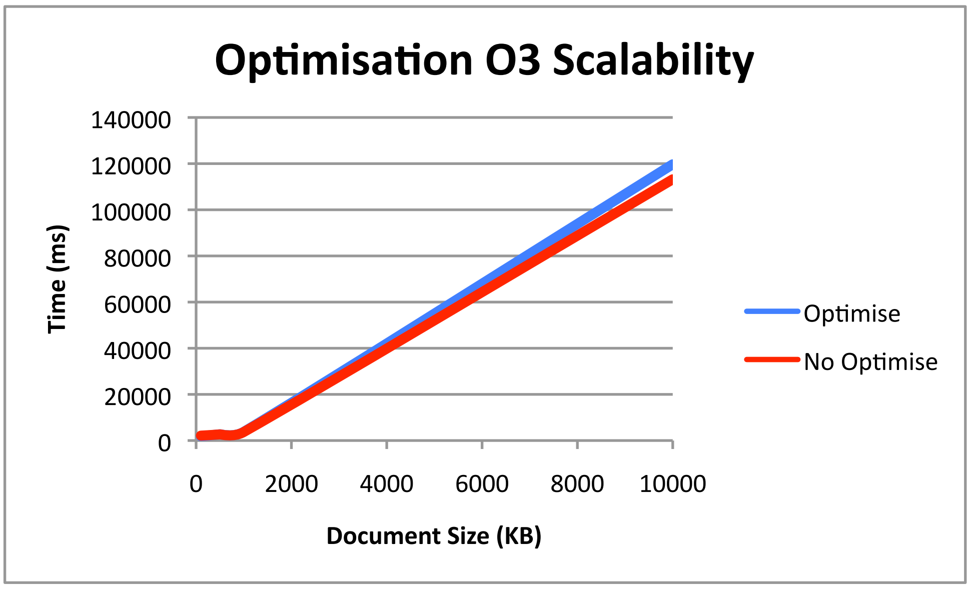


FIGURE 5.35: Scalability for Reduction Rule O3

	Document Size (KB)			
	100	**500**	**1000**	**10000**
Optimise	2403	4859	7880	145100
No Optimise	2602	5358	9133	215304

Table 5.37: Scalability for Reduction Rule I5 (ms)

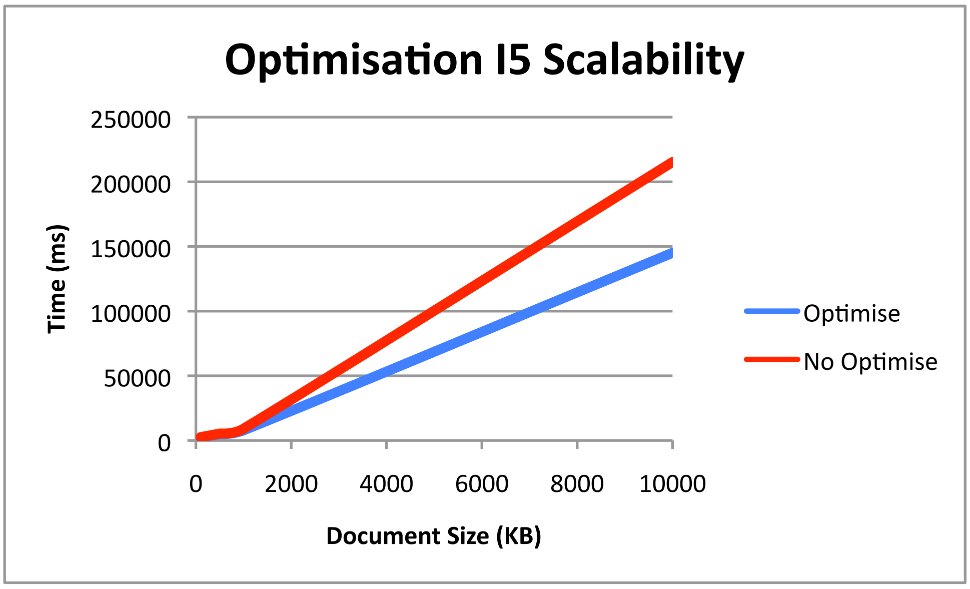


Figure 5.36: Scalability for Reduction Rule I5

5.6 Additional Tests

Additional tests were performed to evaluate design decisions. These tested the performance of lattice structures with varying amounts of information.

5.6.1 Lattice Snowcaps vs Leaves

In this experiment, the trade-offs between two simple alternatives of storing the lattice nodes was studied. The first alternative, which is termed *snowcaps (S)*, stores a minimal set of snowcaps. More precisely, given a view lattice, the minimum number of snowcaps required to maintain the view along with the leaf nodes is maintained. The minimum number is defined as one snowcap from each level of the lattice. When several options exist at a level, the first one is always selected. It should be noted that the root node snowcap in the lattice is empty and is never maintained, as it is the view definition. In the second alternative, *leaves (L)*, only the lattice leaves are stored and the required snowcaps are computed on the fly. There are several alternatives in between - these represent two extreme cases. In this experiment, two different times were measured: (R) the time to evaluate the terms in the view expression by using the snowcaps or leaves lattice and (U) the time to update the snowcaps or leaves lattice. Table 5.38 and Figure 5.37 show the view maintenance results for snowcaps vs leaves for an insertion, whereas Table 5.39 and Figure 5.38 show the results for a deletion. Table 5.40 and Figure 5.39 show the view update results for snowcaps vs leaves for an insertion, whereas Table 5.41 and Figure 5.40 show the results for a deletion.

Document Size	BL (S)	EU (S)	UL (S)	BL (L)	EU (L)	UL (L)
100KB	816	241	449	337	472	0
500KB	1726	301	488	890	678	0
1MB	3051	270	684	1650	856	0
10MB	22343	48141	41412	17159	2843	0

Document Size	BL (S)	EU (S)	UL (S)	BL (L)	EU (L)	UL (L)
100KB	304	136	0	212	244	0
500KB	555	194	0	502	331	0
1MB	944	244	0	876	446	0
10MB	8525	1097	0	8398	1302	0

Table 5.38: Snowcaps vs Leaves Lattice for Insertions for View Q_4, Update X2_L (upper) and View Q_6, Update E6_L (lower) (ms)

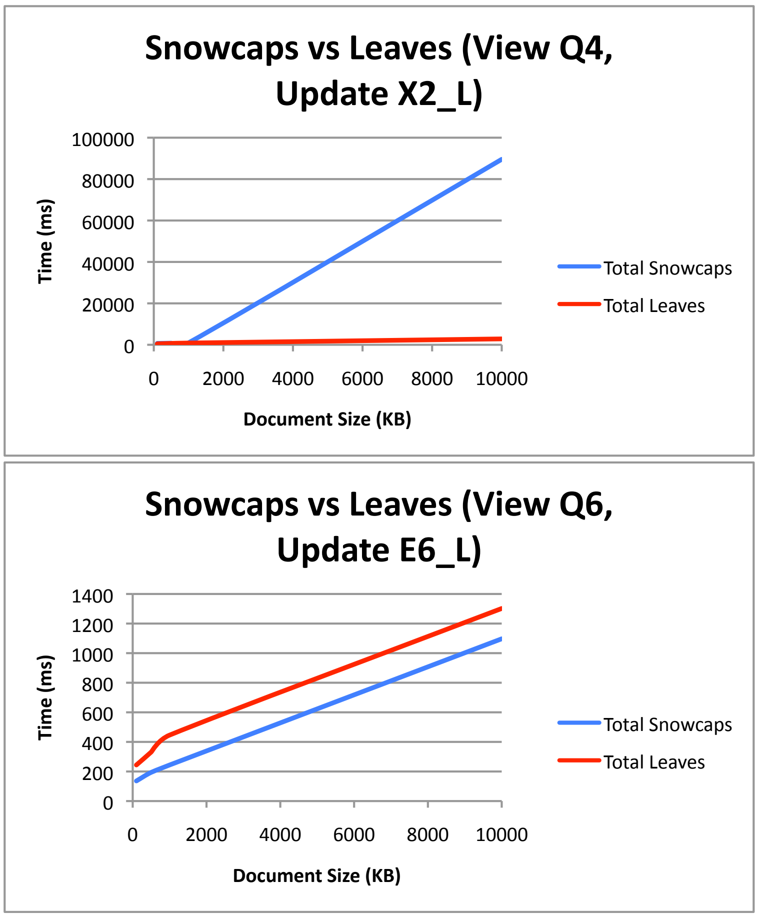
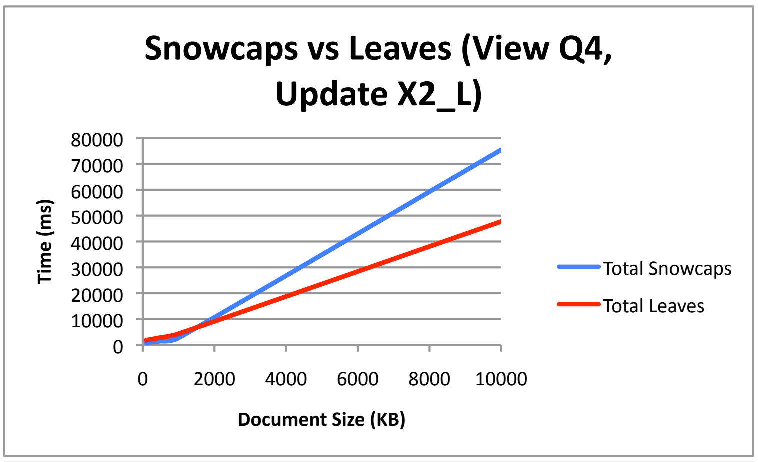


Figure 5.37: Snowcaps vs Leaves Lattice for Insertions

Document Size	BL (S)	EU (S)	UL (S)	BL (L)	EU (L)	UL (L)
100KB	775	880	14	325	1893	7
500KB	1703	1439	95	863	2865	36
1MB	2841	2377	385	1653	4272	101
10MB	22206	20548	54874	17009	27069	20657
Document Size	**BL (S)**	**EU (S)**	**UL (S)**	**BL (L)**	**EU (L)**	**UL (L)**
100KB	286	150	3	202	263	4
500KB	534	404	8	438	572	9
1MB	896	696	17	823	919	18
10MB	8457	5336	1055	8798	5782	1060

TABLE 5.39: Snowcaps vs Leaves Lattice for Deletions for View Q_4, Update X2_L (upper) and View Q_6, Update E6_L (lower) (ms)



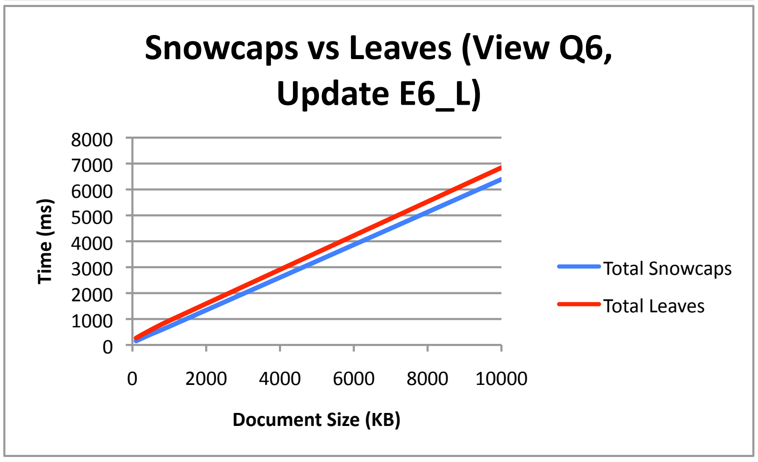


FIGURE 5.38: Snowcaps vs Leaves Lattice for Deletions

Document Size	BL (S)	EU (S)	UL (S)	BL (L)	EU (L)	UL (L)
100KB	479	234	202	23	534	0
500KB	420	216	191	24	456	0
1MB	395	418	189	25	427	1
10MB	405	420	194	26	421	0

Document Size	BL (S)	EU (S)	UL (S)	BL (L)	EU (L)	UL (L)
100KB	157	216	0	18	374	0
500KB	156	182	0	31	368	0
1MB	227	231	0	102	486	0
10MB	1156	650	0	1019	1111	0

Table 5.40: Snowcaps vs Leaves Lattice for Insertions for View Q_4, Update X1_L_3 (upper) and View Q_6, Update X1_L_3 (lower) (ms)

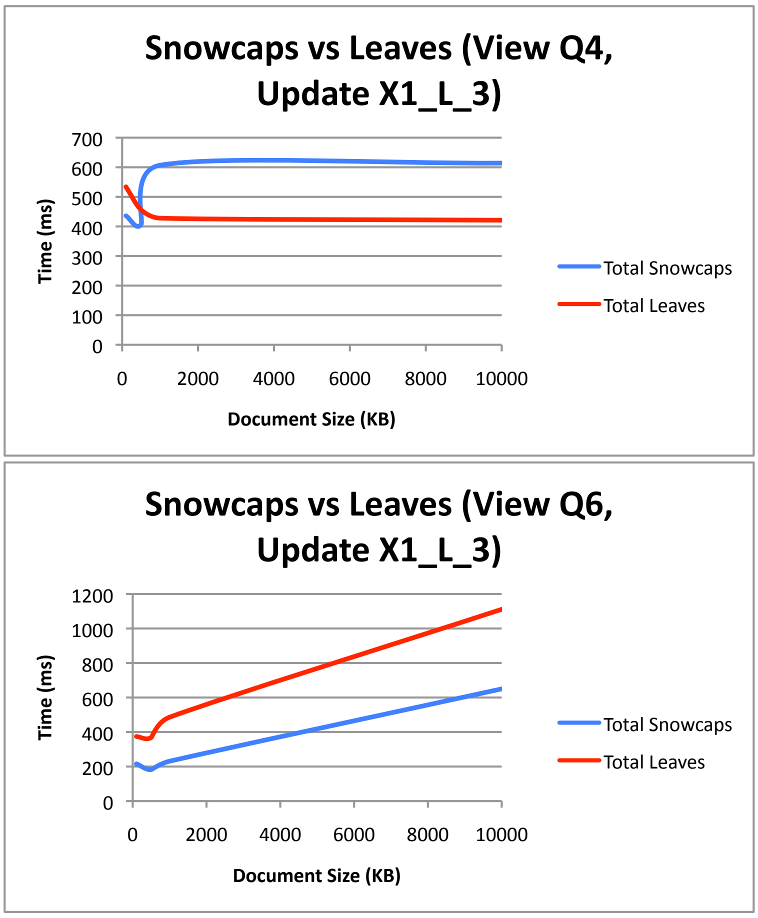
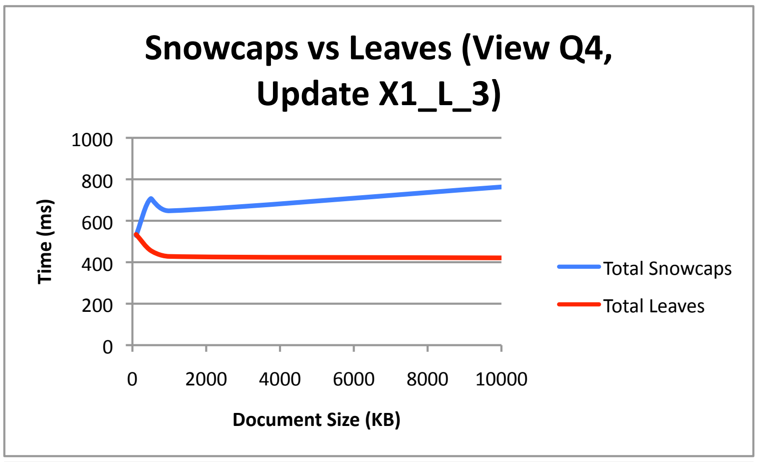


Figure 5.39: Snowcaps vs Leaves Lattice for Insertions

Document Size	BL (S)	EU (S)	UL (S)	BL (L)	EU (L)	UL (L)
100KB	545	529	0	23	534	0
500KB	471	706	1	24	456	0
1MB	413	648	0	25	427	1
10MB	545	762	1	26	421	0

Document Size	BL (S)	EU (S)	UL (S)	BL (L)	EU (L)	UL (L)
100KB	163	170	2	18	374	0
500KB	213	400	17	31	368	0
1MB	175	711	27	102	486	0
10MB	1080	5305	1362	1019	1111	0

TABLE 5.41: Snowcaps vs Leaves Lattice for Deletions for View Q_4, Update X1_L_3 (upper) and View Q_6, Update X1_L_3 (lower) (ms)



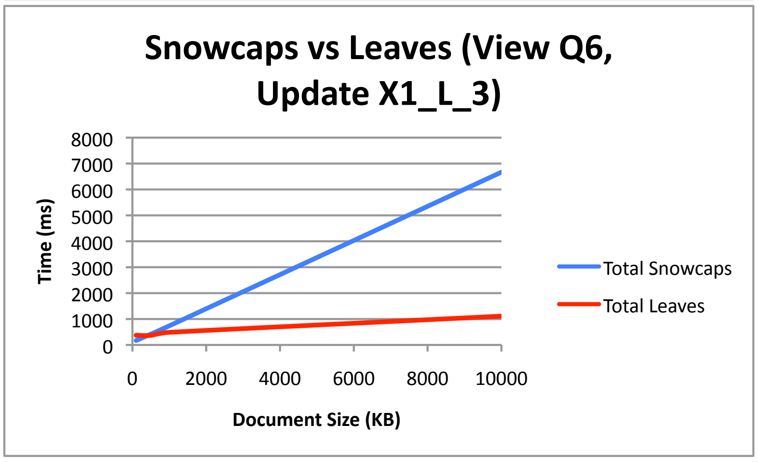


FIGURE 5.40: Snowcaps vs Leaves Lattice for Deletions

In this chapter the results for each set of experiments is presented in a consistent pattern. The detail of insertion and deletion times for view maintenance and view update are presented along with the overall time for these operations. In each case scalability is investigated as well as the impact of enhancing the view with different annotations. Comparable results for view maintenance and view update are presented as well as an investigation of the impact of dynamic reasoning and the impact of varying strategies for employing snowcaps. The next chapter discusses the meaning of these results.

6 Discussion

In Chapter 4, consistent algebraic approaches to view maintenance and view update were presented. In Chapter 5, a set of experiments that evaluated these approaches were presented, along with a selection of their results. In this chapter the significance of the results for both view maintenance and view update are discussed. In addition, the relationship between the results and related work in Chapter 2 is assessed, and the recommendations for future work are presented.

6.1 View Maintenance

This section discusses the behaviour observed from the view maintenance experiments. Some of these observations were similar for both update types, insertions and deletions, e.g., the time to build the lattice, as the lattice is the same for both insertions and deletions. However, others varied depending on the update types, e.g., the time to update the view was greater for deletions due to a greater number of delta tuples[1].

The view maintenance method was split into several stages. These stages are discussed in Section 5.2. The insertion results are discussed in Section 6.1.1 and the deletion results in Section 6.1.5.

6.1.1 Insertions

The performance of each stage was affected by different factors. These ranged from the update query types to the view structure. For insertions the time to *Find Target Nodes* was dependant on the update query type. Generally the Linear, Linear with Boolean Filter and AND queries were most efficient. The OR and AND/OR queries were more computationally expensive (see Figures 5.1, 5.2, and 5.3). This was due to queries being expressed in XAMs which are unable to express OR expressions. Therefore, to handle OR and AND/OR queries, separate XAMs were required specifying each side of the OR.

The time to *Compute Delta Tables* is not affected by the query type. The number of target nodes determines the performance time. For example, in Q_1 the fastest query to compute the delta tables is the AND, which takes 194ms, and the slowest was the Linear query taking 733ms (see Table 5.2 and Figure 5.1). The difference in times was due to the AND query having 56 target tuples, whereas the Linear query had 255.

Get Update Expression is affected by the view definition and the pruning methods. The query type is not a contributing factor. As the number of delta tables increase, the number of terms in the update expression increases. Therefore, the time to run the pruning methods increases with the number of delta tables. The number of delta tuples has an effect on the ID-driven pruning (see Proposition 4.1.16). More delta tuples results in greater evaluation time for this pruning method.

To build the lattice only the view definition and the document is required. Therefore, the *Build Lattice* time is similar for each query type (see, for example, *Build Lattice* (BL) in Table 5.2 and Figure 5.1).

[1]the tuples stored within the Δ tables.

There are several factors which contribute to the *Execute Update* time. These are:

- the total number of target nodes: the higher the number of target nodes, the greater the size of the delta tables (size is number of tuples).
- the number of val and cont attributes on the ancestors of the target nodes in the view. This is because PIMT execution is required which is expensive.
- the number of terms in the update expression surviving pruning. This is because term computation is expensive and was the reason for introducing pruning methods.

The query type doesn't affect the *Execute Update* time (see Figures 5.1, 5.2, and 5.3). However, the OR and AND/OR queries always take the longest to run. This is related to the time to *Find Target Nodes* and the problem of XAMs not being able to represent OR expressions. Therefore, these queries require multiple runs of the algorithm (one for each branch of the OR) to execute the update. This also affects the *Update Lattice* time resulting in OR and AND/OR queries having the longest update time. Aside from this the *Update Lattice* time is determined by the view definition, the number of snowcaps, the number of delta tuples and the proximity of the insertion node(s) to the view root. This can be seen by the slight differences between query types for each view but large differences between certain views. For example, for query $X1\_L$ on view $Q1$, document size 1MB, the time to update the lattice is 231ms (see Table 5.2 and Figure 5.1) but the time for query $X2\_L$ on view $Q2$, document size 1MB is less than 1ms (see Table 5.3 and Figure 5.2). $X1\_L$ adds 255 new tuples, whereas $X2\_L$ adds 708 new tuples. In both scenarios deltas are leaves. However, $Q1$ is structured $/site/people/person[id]/name$ and Q_2 /site/open_auctions/open_auction/bidder/increase. The lattices for $Q1$ and $Q2$ can be seen in Figures 6.1 and 6.2 respectively. From Figure 6.2, it can be seen that $Q2$ only requires adding the new tuples to the *increase* leaf, whereas, Q_1 requires adding the tuples to the *name* leaf as well as performing the join to update the snowcap $/site/people/person/name$, which requires performing a join of 255 new tuples with 1275 existing tuples.

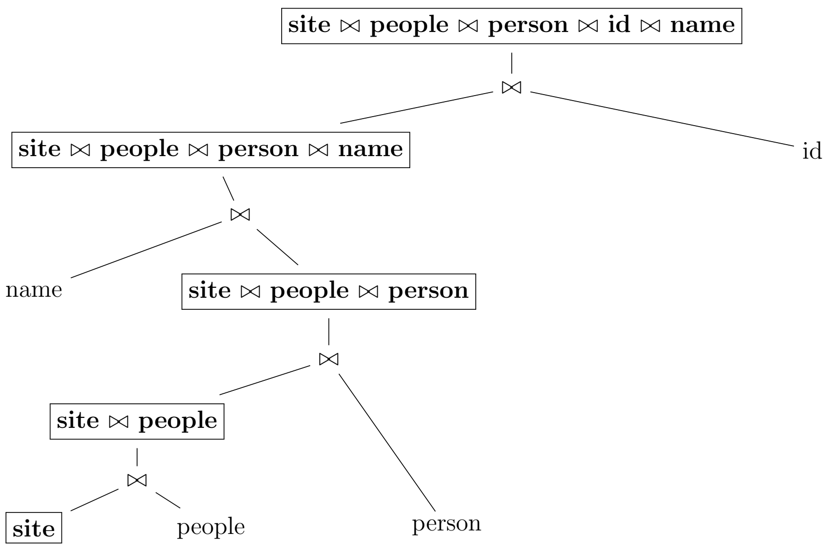

site ⋈ people ⋈ person ⋈ id ⋈ name
⋈
site ⋈ people ⋈ person ⋈ name
id
⋈
name
site ⋈ people ⋈ person
⋈
site ⋈ people
⋈
site
people
person

FIGURE 6.1: Lattice for Q1

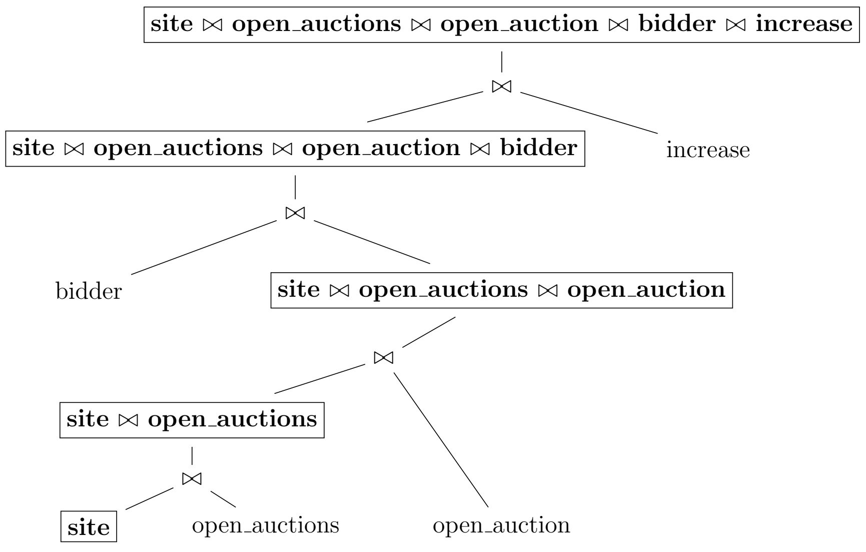

site ⋈ open_auctions ⋈ open_auction ⋈ bidder ⋈ increase
⋈
site ⋈ open_auctions ⋈ open_auction ⋈ bidder
increase
⋈
bidder
site ⋈ open_auctions ⋈ open_auction
⋈
site ⋈ open_auctions
⋈
site
open_auctions
open_auction

FIGURE 6.2: Lattice for Q2

The more delta tuples there are, the more computationally expensive it is to perform the joins to update the lattice. Additionally, the closer the insertion node(s) are to the root of the view, the more snowcaps in the lattice will require updating. This is a consequence of more lattice nodes being affected.

Finally, the time to *Update Source* is another stage affected by the inability to express the OR operator, resulting in OR and AND/OR queries again having the greatest performance time. It was expected that the number of nodes to be added to the document would also be a factor. However, it appears this time is negligible compared to reading the file into memory and writing back to the file.

6.1.2 Scalability

View maintenance for insertions scales linearly for all the document sizes up to 10MB. This is clear from the results shown in Figure 5.5. All the results increased with the document size[2]. The increase for *Execute Update* and *Update Lattice* was not as visible until reaching larger document sizes. The exception to this pattern was *Get Update Expression.* This was due to *Get Update Expression* not relying on the document size but the view definition and the delta tables. The figure shows that all results (affected by the document size) grow gracefully with the size of the document.

6.1.3 Varying Path

View maintenance time increases as the update query path lengthens, or more accurately, as the number of target nodes increases. *Find Target Nodes*, *Compute Delta Tables* and *Update Source* times all increased with the update query path length. This was a result of there generally being more target nodes as the leaves were approached. *Get Update Expression* and *Build Lattice* were uniform as the update query path varied. This was due to both these operations relying on the view definition and the document size. *Execute Update* increased with increasing path length, as generally this increases the number of target nodes. This is not always the case, i.e., /site/people has the same number of target nodes as /site. The insertion /site/people is actually faster (see Table 5.7 and Figure 5.6). The factors contributing to this performance are the number of delta tuples, existing tuples and number of terms in the update expression. The number of delta tuples typically increases with update path length. However, delta tables decrease improving the amount of the expression that can

[2]In this chapter when saying "increased with the document size" it means increased with the number of relevant tuples for the view. However, the assumption is made that the number of relevant tuples will increase with the document size

be replaced by the precalculated lattice nodes. The results show that the main contributing factor to this time is the number of delta tuples. Finally *Update Lattice* decreases with the increase of path length. This is due to the fact that the closer the target node(s) are to the root, the more lattice nodes are affected, increasing the number of joins required.

6.1.4 Varying Annotations

The greater the number of val and conts and the closer they are to the root, the greater the performance time due to the increased cost of PIMT. These results can be seen in Figure 5.7. *Find Target Nodes* is uniform as a result of it only being affected by the query type and the number of target nodes. These factors remain the same for each view as the val and conts do not affect them. *Get Update Expression* also remains similar as it is only affected by the view definition and document size. *Build Lattice*, *Execute Update* and *Update Lattice* increase with the number of val and conts and also the proximity of val and conts to the root. Val and conts closer to the root take longer to process due to them using more memory. A single val and cont at the root takes longer than multiple val and conts near the leaves (see Table 5.8 and Figure 5.7). *Update Source* is uniform when val and cont is not stored for the root as it will only be affected by the query type and the number of target nodes. However, it is more efficient in the cases where val and cont is stored for the root. This is due to the cont for the root being the same as the document, therefore, when PIMT queries it for target nodes it is the same as querying the document for target nodes when updating the source. Saxon caches previously evaluated queries, so in these cases *Execute Update* takes the performance hit for the query, resulting in faster *Update Source* times. *Compute Delta Tables* is affected when the val and conts are on the target or descendant nodes. When the val and conts are ancestors of the target node in the lattice they do not affect the time as they are not included in the delta tuples.

6.1.5 Deletions

The factors affecting each stage for the deletion case were the same as for insertions. The only exception was the time to *Update Lattice*. The general behaviour was the same, the greater the number of delta tuples, the longer the time to update. However, in this case the OR and AND/OR queries did not always take the longest to run. For example, compare Table 5.9 and Figure 5.8 with Table 5.10 and Figure 5.9. This is because the lattice is searched for the relevant tuples and they are removed. Whereas, for insertions, joins are performed for each new set of tuples. Since there are no joins, whether the deletions are performed all at once or separately doesn't affect the time.

6.1.6 Scalability

The scalability (see Table 5.13 and Figure 5.12) shows that view maintenance for deletions scales linearly for documents up to 10MB. The behaviour was similar to insertions in most cases. The only difference was that as document size grew, the time to *Find Target Nodes* exceeded the time to *Build Lattice*. However, despite the differences the results still grow gracefully with the document size.

6.1.7 Varying Path

View maintenance time decreases as the update path lengthens. This is because the longer the update paths, the fewer nodes there are to be deleted. *Find Target Nodes* and *Compute Delta Tables* decrease as the update path increases. This can be seen in Table 5.14 and Figure 5.13 and is due to there being fewer tuples to delete as the target nodes approach the leaves. Similarly *Execute Update* and *Update Lattice* decrease for the same reason. Additionally, the fragment of each term in the update expression that is available from the lattice increases and the number of snowcaps affected decreases due to there being less delta tables and delta tuples. This results in an overall improvement in performance. *Get Update Expression* is uniform due to only being affected by the view and the document size. Finally, *Update Source* increases. The more target nodes, the longer the deletion takes. Additionally, fewer target nodes typically mean it is closer to the root resulting in a larger fragment of the XML document being removed. This results in less write time back to the file.

6.1.8 Varying Annotations

The greater the number of val and conts and the closer they are to the root, the greater the performance times due to the increased cost of PDMT. These results can be seen in Figure 5.7. *Find Target Nodes* and *Update Source* are uniform as a result of them only being affected by the query type and the number of target nodes for *Find Target Nodes* and the number of target nodes and the inserted XML fragment for *Update Source*. All these factors remain the same for each view as the val and conts don't affect them. This is also true for *Compute Delta Tables* as val and conts are not stored in delta tables for deletions. *Get Update Expression* also remains similar as it is only affected by the view definition and document size/number of view tuples. *Build Lattice* and *Execute Update* increase with the number of val and conts and also the proximity of val and conts to the root. Val and conts closer to the root take longer to process due to them using more memory. A single val and cont at the root takes longer than multiple val and conts near the leaves.

6.1.9 Comparison of Insertions and Deletions

For all the experiments performed, the time to *Find Target Nodes* was the most dominant. This can be seen from the insertion and deletion experiments shown in Sections 5.2.1 and 5.2.2 respectively. It can also be observed that there are two other times, *Build Lattice* and *Update Source*, which are equivalent or greater than the time to *Find Target Nodes*. *Build Lattice* is a one off cost which is only required the first time view maintenance is performed on the view. Its practical contribution to degrading query performance is therefore limited. *Update Source* is not actually part of the view maintenance process. It is a side-effect required to keep the database consistent. This required cost is unavoidable regardless of whether the view is maintained incrementally or fully recomputed following each change that affects it.

Some of the times were comparable for insertions and deletions due to the same or similar methods being utilised. However, the biggest difference came as a result of the generally increased number of delta tables for deletions. Since the delta tables for deletions included the target node(s) and all their descendants, there were generally more delta tuples for these cases. As a result of this the time to *Execute Update* was generally faster for insertions. However, this improved the *Update Source* time. This was due to the process for updating the source documents. The source document is brought into an in-memory representation. This is then updated and output back to a file. In the deletion case the write to file time will be quicker due to there being less to write. Additionally, the update time may be faster as the removal of one node (it's descendants are deleted as a side-effect of this) is faster than the addition of multiple nodes.

The time to *Compute Delta Tables* is generally less in the case of deletions. For example, compare Table 5.3 and Figure 5.2 with Table 5.10 and Figure 5.9. This was because the deletions only had to store the URI and ID for each delta tuple. The extra time required to calculate the val and conts for the insertion delta tuples, where required, resulted in longer times to compute the delta tables despite the number of delta tuples being generally smaller.

Update Lattice time is not consistently better for insertions or deletions, it depends on the scenario. This is because of the different approaches taken to update the lattice in the insertion and deletion case. Insertions require adding the new tuples to the relevant leaves and propagating the new tuples up towards the root of the lattice, joining the new tuples with the existing tuples. The amount of work required is dependent on the number of existing tuples in each required join and the number of new (delta) tuples. Conversely, for the deletion case the tuples are removed from the relevant leaves and propagated up towards the root of the lattice, removing any tuples which contains a tuple that has been deleted. This requires searching through all the tuples at each affected lattice node for each tuple to be removed. Therefore, depending on the scenario the relevant method for either insertions or deletions may perform better. This will be further explored in Section 6.5.1.

Scalability

View maintenance scales linearly for insertions and deletions on document sizes up to 10MB. There are slight variations in behaviour but both scale gracefully.

Varying Path

View maintenance time decreases for deletions as the update path lengthens, whereas insertion times increase. The overall performance for insertions, by comparison with deletions at the same path length, is significantly better. The main difference was the time to *Find Target Nodes*. The closer the target node(s) were to the root, for deletions, the more nodes there were to be deleted. However, the number of target nodes increased with the update path length making *Find Target Nodes* more expensive, for insertions, when near the leaves as a result of the tree structure of XML. *Compute Delta Tables* and *Execute Update* both increased with path length for insertions and decreased with path length for deletions. This was due to the number of descendants/Δ^- decreasing with path length but the number of target nodes/Δ^+ increasing. *Update Lattice* decreased for both with path length due to the decreasing fragment of the lattice affected as the query approached the leaves. *Update Source* increases for deletions as the update path lengthens. This is due to the behaviour described in Section 6.1.7. However, for insertions it remains uniform (see Section 6.1.1).

Varying Annotations

For both insertions and deletions, the greater the number of val and conts and the closer they are to the root, the greater the performance times due to the increased cost of PIMT/PDMT. This has more effect on insertions since val and cont have a bigger effect, especially for *Execute Update* and *Update Lattice*. All the differences are already visible in the comparison of insertions and deletions. The main difference between these results is the increased time for *Compute Delta Tables*. The val and conts are not stored for deletions and consequently do not affect the performance of this operation but their increasing number and proximity to the root has an impact on insertions. *Update Source* differs due to the same reasons as insert and delete comparisons and caching in Saxon.

6.2 View Update

This section discusses the behaviour observed from the view update experiments. Some of these observations were similar for both update types: insertions and deletions. However, others varied depending on the update. The view update method was split into several stages. These stages are discussed in Section 5.2. The insertion results are discussed in Section 6.2.1 and the deletion results in Section 6.2.4.

6.2.1 Insertions

The performance of each stage was affected by different factors. These revolved around the view definition; the number of tuples in the view; the number of delta tables; and the proximity of the target node(s) to the view's root or leaves.

Find Target Nodes increased with the path length. This was due to the tree structure of XML which results in the number of target nodes generally increasing with proximity to the leaves. The number of tuples in the view also contributes to the performance.

As an XML document is a tree, the number of nodes increases with each level. Therefore, the *Compute Delta Tables* time increases with the proximity of the target node(s) to the view leaves due to the larger number of delta tuples. This can be seen in Tables 5.16, 5.17, and 5.18 and Figures 5.15, 5.16, and 5.17.

The query doesn't affect the time to *Get Update Expression.* This is affected by the view definition and the pruning methods. Despite queries near the root having more delta tables but fewer tuples, the time difference is negligible.

The *Build Lattice* time only relies on the view definition. The performance is also affected by the number of view tuples and not the number of applicable tuples returned from the document. *Execute Update* behaves similarly to view maintenance as it is only affected by the number of delta tuples and the number of expressions left after pruning. It is not dependent on the query type. The target node(s) position in relation to the root is also of importance with the number of target nodes decreasing as the root is approached. However, the effect is negligible for the insertion tests as they were inserting small XML fragments. This behaviour is more noticeable in the deletion tests, as the delta tables were larger, this is discussed further in Section 6.2.4.

Update Lattice time increases with the target node(s) proximity to the view root. The closer to the root, the more snowcaps are affected, resulting in more joins being performed. The number of delta tuples is also a contributing factor as it was for view maintenance. Similarly to view maintenance *Update Source* is uniform regardless of the number of delta tables and their size, as it is dominated by input/output of files rather than number of target nodes/size of the XML fragment.

6.2.2 Scalability

View update scales linearly for insertions on document sizes up to 10MB. This is clear from the results shown in Figure 5.19. All results grow gracefully with view size except *Get Update Expression* which is uniform as it is not affected by the document size only the view.

6.2.3 Varying Annotations

Similarly to view maintenance, the greater the number of val and conts and the closer they are to the root, the greater the performance times due to the increased cost of PIMT - the same algorithms as for view maintenance are used for modifications as they only affect the view, not the documents. These results can be seen in Figure 5.20. These results follow the same trends as view maintenance insertions (see Figure 5.7) as the val and conts affect the views and not the documents.

6.2.4 Deletions

The deletion results are only discussed briefly as they generally perform in a similar way to insertions (see Tables 5.22, 5.23, and 5.24 and Figures 5.21, 5.22 and 5.23). All the queries for view update are linear, resulting in similar *Find Target Nodes* times. Performance time is also affected by the number of tuples in the view and the target node position relative to the root in the view. The target nodes are taken directly from the tuples. Therefore, performance is better when closer to the root because the whole tuple doesn't need to be checked, only up to the target node. *Compute Delta Tables* decreases with path length as the number of delta tables decreases. This results in a slight difference in the results for *Execute Update* due to the semantics of delta tables for deletions. The performance time decreases as the target node(s) approach the view leaves. This is due to there being fewer delta tables and more of the update expression is precalculated by the lattice, resulting in less work having to be performed to evaluate the update expression.

6.2.5 Scalability

View update scales linearly for deletions on document sizes up to 10MB. This is clear from the results shown in Figure 5.25. *Update Source* was the most dominant time but not the most important. The behaviour was similar to insertions in most cases. As document size grew, the time to *Update Lattice* exceeded the time to *Build Lattice*. This was due to *Update Lattice* having to search all the tuples in each affected lattice node after each update. This eventually became more expensive than the time to perform the joins to *Build Lattice*. These results suggest that for larger document sizes the lattice is inefficient for view update deletions and should therefore not be used.

6.2.6 Varying Annotations

Varying val and conts only affects the operations involving the view. Therefore, it follows the same trends as view maintenance deletions.

6.2.7 Comparison of Insertions and Deletions

As in view maintenance less significance is placed on the *Build Lattice* and *Update Source* times for the same reasons. The most dominant time for view update insertions is *Find Target Nodes*. *Compute Delta Tables* can also dominate in cases of large numbers of target nodes, i.e., Q2. Conversely, deletions are almost always dominated by *Execute Update* and *Update Lattice*. When there are large number of delta tuples *Update Lattice* dominates, e.g., Table 5.23 and Figure 5.22.

Some of the times were comparable for insertions and deletions due to the same or similar methods being utilised. The *Find Target Nodes* time was greater for insertions. This is due to requiring source queries to ensure the new IDs in the delta tuples are correct and do not reuse the ID of another element outside the view but present in the source document.

Compute Delta Tables took significantly longer for insertions but this is only noticeable when the documents get larger. This was due to the delta tuples having to be calculated for insertions. However, for deletions they could just be taken straight from the view.

Similarly to view maintenance the *Execute Update* time was greater for deletions. This was again the result of deletions generally having more tuples to remove than the insertions having to insert, with the number increasing closer to the root. For example, for view Q1 and update X1_L for the 1MB document there is only one new tuple. However, for update DeleteX1_L there are 708 tuples to be removed. Other similarities to view maintenance exist with *Update Lattice* which is faster for insertions but may not always be, for the same reasons as view maintenance - see Section 6.1.9.

Get Update Expression is similar and uniform for both. Despite the deletions generally having more delta tuples the effect appears to be negligible in the case of view updates. This is due to the difference between numbers of tuples being less significant than in view maintenance, as generally speaking the number of deletion delta tuples in view maintenance is much larger.

Varying Annotations

In the same way as with view maintenance, for insertions and deletions, the greater the number of val and conts and the closer they are to the root, the greater the performance times due to the increased cost of PIMT/PDMT.

6.3 View Maintenance vs View Update

View maintenance and view update are two inverse problems, however, both can be used to achieve the same result. This is limited to the extent that view update can only handle the view maintenance updates where all the nodes in the query specifying the target node(s) are contained within the view. For example, for view $Q1 = /site/people/person[id][name]$ and update $A6_A = /site/people/person[phone][homepage]$, processing via view update is not feasible. This is due to there being hidden information, i.e., it is not possible to tell whether the nodes in the view have a *phone* and *homepage* child. Additionally, there may be more *person* nodes in the source document but the view only knows about those with *id* and *name* children. Generally, view update time is less than view maintenance time for similar operations, e.g, see Table 5.28 and Figure 5.27. This is more evident for deletions. This behaviour is a result of smaller lattices and the faster access time of the view as opposed to the document for deletion target nodes. Therefore, where applicable, the view update method can be used to improve the performance time of the view maintenance method.

There are various reasons for the view update method being more efficient. *Find Target Nodes* involves querying the view rather than the document for deletions. Due to the view generally being significantly smaller and having a faster access time, the performance is improved. *Compute Delta Tables* is the same for insertions due to the same method being used. This is affected by the number of target nodes, which is the same in these experiments. However, deletions store all the target nodes and their descendants. This can result in a much larger set of nodes for view maintenance as there are generally fewer nodes in a view, therefore, fewer descendants. *Build Lattice* is improved by storing only the tuples which are relevant to the view. Therefore, the lattice is smaller and takes less time to build. The view maintenance lattice contains irrelevant tuples as the nodes are evaluated on the document when it is built. For example, the leaf with pattern //name returns all name nodes in the document. However, not all of these belong to person nodes. These are discarded by the joins but the extra tuples result in extra work. *Execute Update* has better performance times for view update. This is related again to the irrelevant tuples. However, a lot of the results are close due to the lattice replacement, which is usually a join expression which has removed the irrelevant tuples. The number of delta tuples can play a big role in this. The number of tuples to be deleted is smaller for view update because there can be many hidden from the view, i.e., descendants of a node in the document may not be present in the view. This isn't always a factor though as they still have to join with the view tuples so the view maintenance would flush out the others - when replaced with a join from the lattice, which is the common occurrence, replacement by just a leaf is rare which could potentially contain irrelevant tuples in the lattice leaf, unlike view update. *Update Lattice* is generally quicker for view update due to similar reasons for the build time. There are usually more nodes than those which belong to the view in the view maintenance lattice, and consequently there are more tuples to join. Performance still relies on the number of tuples in the lattice and the number of delta tuples. Finally, *Update Source* for the majority of the results is similar. The same factors apply. However, view update can take longer if view tuples come from multiple documents requiring update source to perform updates on multiple documents.

Overall view update performs better than view maintenance in the same scenario. This is more evident for deletions but still holds true for the case of insertions.

6.4 Optimisations

In general, using optimisations for view maintenance improves the performance. This improvement is more significant as the percentage of the update pruned increases and as the document sizes increase. For some of the smaller documents with lower pruning percentage, the use of pruning is not always an improvement: for example, see Table 5.31 before 100% - with the exception of 40% - and Table 5.33 for 40% and 60%. However, for pruning rule O1 it is always an improvement. Therefore, pruning rule O1 is always beneficial, whereas rules O3 and I5 are only beneficial when the percentage pruned is large.

Optimisations for view update are beneficial for all rules with the exception of rule O3. O3 only appears to be useful for smaller document/view sizes, see for example Table 5.36. However, the other rules benefit view update with this benefit increasing with size, similarly to view maintenance. Therefore, pruning rule O1 and I5 are always beneficial, but O3 is only useful for smaller document/view sizes.

6.5 Additional Tests

6.5.1 Impact of Snowcaps versus Leaves

View Maintenance:

In the snowcaps vs leaves experiments, the total times to evaluate the terms and update the lattice for snowcap lattices and leaf lattices are presented for views Q4 and Q6 respectively. Results are shown for view maintenance insertions and deletions and view update insertions and deletions. These can be seen in Tables 5.38, 5.39, 5.40 and 5.41 and Figures 5.37, 5.38, 5.39 and 5.40, respectively. These results show that for view maintenance insertions the snowcaps lattice is better than the leaves lattice for views of significant size but the leaves lattice is better for smaller views. This can be seen by the difference in performance between views Q4 (3 tuples) and Q6 (217 tuples).

For view maintenance deletions the snowcaps lattice is beneficial for both large and small views. However, a more significant difference is shown for the smaller view for smaller documents, there appears to be a crossover for the smaller view where the snowcaps lattice becomes inefficient somewhere between a document size of 1 and 10MB.

View update shows the same behaviour for insertions. An exception is that for smaller view sizes the snowcaps lattice is still better until the document size increases. In the case of Q4, when the document size is 1MB and over. Deletions are only more efficient using snowcaps for small document sizes. In these cases, the larger the view the better the efficiency. These results suggest that efficiency can be improved by altering the lattice structure based on the document/view size.

6.5.2 Comparison with previous work

The most closely related work to this thesis is in [6]. This is also an algebraic view maintenance method which can handle statement-level updates - expressed in a Galax algebraic expression. However, views are represented as XQuery in contrast to this thesis which uses XAMs. Both methods can support non-recursive views. Source queries in the Galax-based system are rewritten to gather auxiliary information in a manner similar to the gathering of auxiliary information to assign IDs for new tuples that is described in this thesis. The quantity of this auxiliary data collected can be controlled by means of an external parameter. However, if it is set too low then full recomputation may be required. The only comparable control in the work of this thesis is the volume of data stored in the lattice - snowcaps vs leaves. However, the work described here will always result in incremental view maintenance and not have to resort to full recomputation. The main difference is that the method presented in this thesis is a general solution, whereas [6] is tied to the Galax algebra.

The boolean XPath incremental view maintenance problem is solved in [17]. This is a subset of the view maintenance problem, solved in this thesis, which involves determining if an XPath expression is still satisfied following an update. The views are represented as XPath queries, in contrast to XAMs. However, they are not materialised. Additionally, they do not support multiple returning nodes. An auxiliary data structure is used for handling incremental updates. This data structure consists of a set of records for each node in the document. The data contained in these records varies depending on the XPath fragment being handled. Each record contains all the children of the node that satisfy the query. This is in contrast to the lattice used in this thesis. Despite mostly focussing on the boolean problem, view maintenance is considered for XPath. This method handles node-level updates for insertions and relabelling, statement-level updates for deletions. The system described in this thesis can support statement-level updates for all these situations although relabelling must be modelled as a deletion followed by an insertion. A theoretical approach is presented by these authors and evaluated by means of complexity results, in contrast to being implemented and presented as performance results in this thesis.

The work presented in [120] is the closest non-algebraic method to this thesis. The main difference is that it can only handle node-level updates, as opposed to statement-level.

The current work on view updates focuses on publishing relational data to XML and maintaining these views. The Round-trip XML View Update Problem (RXU), a subcase of XQuery view update, is handled in [141]. This involves loading the XML into a relational database and then publishing XML views. The view update problem, therefore, involves updating the view and translating the updates to relational updates, instead of updates on the XML document. A similar problem - without the XML loading - is handled in [109] where the XML view must adhere to a schema. A framework, UFilter [146] again handles the view update translatability problem for arbitrary XML views using schema and data checking. Virtual views are handled in [147] by translating updates into XML queries without side-effects, if possible. It is extended to produce STAR [148] which can handle views with duplication. Finally, the idea of using existing relational view update methods was considered in [144]. This involves reducing the XML view update problem to the relational view update problem.

The work in this thesis does not handle recursion. However, [7] supports recursive XML views populated from relational data. Recursive XPath queries can be handled by [149], in addition to recursive views.

6.6 Limitations

This section will discuss the limitations of the view maintenance and view update methods.

6.6.1 General Limitations

Some general limitations apply to both the view maintenance and view update methods. These methods can only handle positive XQuery and non-recursive views and updates. A further limitation is a consequence of using XAMs. XAMs lack support for OR operators, joins, or less than/greater than operators. The only comparison operation supported is equality.

6.6.2 View Update Limitations

The only limitation specific to the view update problem is that branch and value predicates can only be checked if they are contained in the view. Other applicable limitations are covered by the general limitations.

6.6.3 Experimental Limitations

This section discusses limitations with the experiments. There was only one document in the database for each test. This does not have much of an effect on the view maintenance method as tuples from the wrong document would simply be discarded during the joins. However, this could significantly modify the view update times if multiple documents were affected by an update. Another limitation is that *Get Update Expression* didn't have a huge variation in the number of delta tables contained in the update expression. This meant it can not be said for sure that the number results in much of an effect. Further testing would be required with larger numbers of view nodes to verify this claim. A final limitation is that IDs suffer from the overflow problem, defined in [44][3].

6.7 Future Work

Based on the limitations, the following areas are proposed as future work. Expand the number of update types handled. This is to allow the methods to support a larger part of XQuery Update. Currently insert as last child and delete are handled. Other types to include would be: insert the trees in P as first children of node v ($ins^{\swarrow}(v, P)$) and insert the trees in P as children of node v at child position i ($ins^{\downarrow}(v, P, i)$). However, this is just an implementation issue and would fit easily into the framework. Further work will be on improving the overall efficiency of the method by, e.g, making use of additional information to improve pruning. This could be achieved by making use of schemas when available. In situations where schemas are available term pruning can be improved, based on Appendix A. Finally, some focus will be on providing support for negative XQuery and recursive views and updates. This will allow a larger fragment of XQuery to be supported by the methods.

This work has presented a similar incremental algebraic approach to view maintenance and view update using generic operators. It has also shown that converting view maintenance into view update, where possible, gives better performance times, as discussed in Section 6.3. Finally, determining the question of whether dynamic reasoning improved the performance time for view maintenance and view update was explored. It was found that for O1 that this was always the case. However, for rule O3 and I5 it was only beneficial when the percentage pruned was high. For all the rules, the bigger the document, the larger the performance benefit (when percentage pruned is high).

The main contribution of this thesis is methods which take a similar approach to solve the view maintenance and view update problem. These methods are original as they can handle statement-level updates which use generic operators and views returning data from

[3]When the ID size exceeds the number of bits assigned for storage.

multiple nodes.

7 Conclusion

This thesis has focussed on developing algebraic incremental methods to develop a single consistent solution for the problems of view maintenance and view update (RQ1). Similar incremental algebraic methods were developed for solving the problems of view maintenance and view update, using generic operators and supporting statement-level updates and multiple returned nodes. Both these methods followed the same process to support updates with differences occurring in the implementation of certain stages.

Views are typically smaller in size than documents. Based on this, a question this thesis aimed to answer was if view maintenance could be expressed as view update would it improve the performance time (RQ2). The smaller data set used in view update might suggest better performance but the necessary differences in implementation of certain stages may be slower in certain cases resulting in similar, if not worse times for view update rather than view maintenance. It was found that, in general, view update was faster than view maintenance for identical updates that guaranteed the same results. This was due to the smaller lattices and the faster access time to the view. The improved performance was more prominent for deletions. This is because a deleted node also deletes all of it's descendants. In most cases there are more descendants within a source document than are contained in a view, i.e, more delta tables. Therefore, where applicable, the view update method can be used to improve the performance time of the view maintenance method.

The developed methods initially only performed pruning on the algebraic expression. However, this is extended to include dynamic reasoning on the target nodes, based on [156]. This extension was included as another question this thesis aimed to answer was if dynamic reasoning improved the performance time for view maintenance and view update (RQ3). For view maintenance the most effective pruning rule was O1[1]. The other pruning rules, O3[2] and O5[3], were only beneficial when the percentage pruned was large. The results were similar for view update. Rules O1 and I5 were always beneficial with increasing effectiveness with document/view size. The results for O3 differed in that it was only beneficial for smaller document/view sizes. Therefore, most dynamic reasoning reduction rules are effective for view maintenance and view update, with this benefit increasing with the size of the data set/percentage pruned.

The main contribution of this thesis is the development of similar incremental algebraic methods which provide a consistent solution to the view maintenance and view update problems. The originality of these methods is their ability to handle statement-level updates using generic operators and views returning data from multiple nodes. It has also shown that, where possible, expressing view maintenance as view update improves the performance time, with the difference being more significant for deletions. Finally, it has shown that dynamic reasoning rules can be used to improve the performance time for the view maintenance and view update methods.

[1]Deletion following an insertion or deletion on the same node.

[2]An insertion or deletion on a node, followed by the deletion of an ancestor of this node.

[3]Combine two insertions on the same node into one.

References

[1] A. Arion. *Xml access modules: Towards physical data independence in xml databases.* In *In XIME-P* (2005).

[2] M. Bojanczyk and P. Parys. *XPath Evaluation in Linear Time.* J. ACM **58**(4), 17:1 (2011). URL `http://doi.acm.org/10.1145/1989727.1989731`.

[3] L. Xu, T. W. Ling, H. Wu, and Z. Bao. *DDE: from Dewey to a fully dynamic XML labeling scheme.* In *SIGMOD* (2009).

[4] A. Balmin, Y. Papakonstantinou, and V. Vianu. *Incremental Validation of XML Documents.* ACM Trans. Database Syst **29**(4), 710 (2004). URL `http://doi.acm.org/10.1145/1042046.1042050`.

[5] V. Josifovski, M. Fontoura, and A. Barta. *Querying XML Streams.* VLDB J **14**(2), 197 (2005). URL `http://dx.doi.org/10.1007/s00778-004-0123-7`.

[6] J. N. Foster, R. Konuru, J. Simeon, and L. Villard. *An algebraic approach to view maintenance for XQuery.* In *PLAN-X workshop* (2008).

[7] B. Choi, G. Cong, W. Fan, and S. Viglas. *Updating Recursive XML Views of Relations.* J. Comput. Sci. Technol. **23**(4) (2008).

[8] *The XML Query Language* (2009). http://www.w3.org/XML/Query.

[9] *XML Path Language* (1999). http://www.w3.org/TR/xpath/.

[10] A. Balmin, F. Ozcan, K. Beyer, R. Cochrane, and H. Pirahesh. *A framework for using materialized XPath views in XML query processing.* In *VLDB* (2004).

[11] M. El-Sayed, E. A. Rundensteiner, and M. Mani. *Incremental maintenance of materialized XQuery views.* In *ICDE* (2006).

[12] B. Mandhani and D. Suciu. *Query caching and view selection for XML databases.* In *VLDB* (2005).

[13] N. Onose, A. Deutsch, Y. Papakonstantinou, and E. Curtmola. *Rewriting nested XML queries using nested views.* In *SIGMOD* (2006).

[14] W. Xu and M. Ozsoyoglu. *Rewriting XPath queries using materialized views.* In *VLDB* (2005).

[15] *The XQuery Update Facility 1.0* (2009). http://www.w3.org/TR/2009/CR-xquery-update-10-20090609/.

[16] A. Gupta and I. S. Mumick. *Maintenance of Materialized Views: Problems, Techniques and Applications.* IEEE Quarterly Bulletin on Data Engineering; Special Issue on Materialized Views and Data Warehousing **18**(2), 3 (1995).

[17] H. Björklund, W. Gelade, M. Marquardt, and W. Martens. *Incremental XPath Evaluation.* In *ICDT* (2009).

[18] A. Sawires, J. Tatemura, O. Po, D. Agrawal, and K. S. Candan. *Incremental maintenance of path-expression views.* In *SIGMOD* (2005).

[19] A. Sawires, J. Tatemura, O. Po, D. Agrawal, A. E. Abbadi, and K. S. Candan. *Maintaining XPath views in loosely coupled systems.* In *VLDB* (2006).

[20] I. Manolescu, K. Karanasos, V. Vassalos, and S. Zoupanos. *Efficient XQuery rewriting using multiple views.* In *ICDE* (2011).

[21] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. *Extensible markup language (xml) 1.0 (fifth edition).* World Wide Web Consortium, Recommendation REC-xml-20081126 (2008).

[22] C. F. Goldfarb and P. Prescod. *Charles F. Goldfarb's XML Handbook* (Prentice-Hall, Upper Saddle River, New Jersey, 2003), 5th ed.

[23] *Microsoft SQL Server.* http://www.microsoft.com/sqlserver/en/us/default.aspx.

[24] *EMC Documentum xDB.* http://uk.emc.com/products/detail/software2/documentum-xdb.htm.

[25] *Mark Logic Server.* http://www.marklogic.com/.

[26] M. Benedikt and C. Koch. *XPath Leashed.* ACM Comput. Surv **41**(1) (2008). URL http://doi.acm.org/10.1145/1456650.1456653.

[27] F. Neven. *Automata Theory for XML Researchers.* ACM SIGMOD Record **31**(3), 39 (2002).

[28] G. Gottlob, C. Koch, and R. Pichler. *Efficient Algorithms for Processing XPath Queries.* ACM Trans. Database Syst **30**(2), 444 (2005). URL http://doi.acm.org/10.1145/1071610.1071614.

[29] M. Bojanczyk and P. Parys. *XPath Evaluation in Linear Time.* In M. Lenzerini and D. Lembo, eds., *Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2008, June 9-11, 2008, Vancouver, BC, Canada*, pp. 241–250 (ACM, 2008). URL http://dl.acm.org/citation.cfm?id=1376916.

[30] P. Parys. *XPath Evaluation in Linear Time with Polynomial combined complexity.* In J. Paredaens and J. Su, eds., *Proceedings of the twenty-eighth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, PODS'09, Providence, Rhode Island, June 29–July 1, 2009*, pp. 55–64 (ACM Press, pub-ACM:adr, 2009).

[31] M. Götz, C. Koch, and W. Martens. *Efficient Algorithms for Descendant-only Tree Pattern queries.* Inf. Syst **34**(7), 602 (2009). URL http://dx.doi.org/10.1016/j.is.2009.03.010.

[32] C. R, J. Simon, and M. Fernndez. *A Complete and Efficient Algebraic Compiler for Xquery.* In *In ICDE* (2006).

[33] P. A. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. *MonetDB/XQuery: a Fast XQuery Processor powered by a Relational Engine.* In S. Chaudhuri, V. Hristidis, and N. Polyzotis, eds., *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pp. 479–490 (ACM, 2006). URL `http://doi.acm.org/10.1145/1142473.1142527`.

[34] P. E. O'Neil, E. J. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. *ORDPATHs: Insert-Friendly XML Node Labels.* In *SIGMOD* (2004).

[35] X. Wu, M. L. Lee, and W. Hsu. *A Prime Number Labeling Scheme for Dynamic Ordered XML Trees.* Data Engineering, International Conference on **0**, 66 (2004).

[36] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. *Storing and Querying Ordered XML using a Relational Database System.* In M. Franklin, B. Moon, and A. Ailamaki, eds., *Proceedings of the ACM SIGMOD International Conference on Management of Data, June 3–6, 2002, Madison, WI, USA*, pp. 204–215 (ACM Press, pub-ACM:adr, 2002).

[37] Abiteboul, Alstrup, Kaplan, Milo, and Rauhe. *Compact Labeling Scheme for Ancestor Queries.* SICOMP: SIAM Journal on Computing **35** (2006).

[38] E. Cohen, H. Kaplan, and T. Milo. *Labeling dynamic XML trees.* In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS-02)*, pp. 271–281 (ACM Press, New York, 2002).

[39] Y. Xu and Y. Papakonstantinou. *Efficient Keyword Search for Smallest LCAs in XML Databases.* In F. Özcan, ed., *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pp. 537–538 (ACM, 2005). URL `http://doi.acm.org/10.1145/1066157.1066217`.

[40] C. Sun, C. Y. Chan, and A. K. Goenka. *Multiway SLCA-based keyword search in XML data.* In C. L. Williamson, M. E. Zurko, P. F. Patel-Schneider, and P. J. Shenoy,

eds., *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, pp. 1043–1052 (ACM, 2007). URL `http://doi.acm.org/10.1145/1242572.1242713`.

[41] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. *On Supporting Containment Queries in Relational Database Management Systems.* In T. Sellis and S. Mehrotra, eds., *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data 2001, Santa Barbara, California, United States, May 21–24, 2001*, pp. 425–436 (ACM Press, pub-ACM:adr, 2001).

[42] Q. Li and B. Moon. *Indexing and Querying XML Data for Regular Path Expressions.* In P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, eds., *Proceedings of the Twenty-seventh International Conference on Very Large Data Bases: Roma, Italy, 11–14th September, 2001*, pp. 361–370 (Morgan Kaufmann Publishers, pub-MORGAN-KAUFMANN:adr, 2001). URL `http://www.vldb.org/conf/2001/P361.pdf`.

[43] X. Wu, M. L. Lee, and W. Hsu. *A prime number labeling scheme for dynamic ordered xml trees.* In *ICDE* (2004).

[44] C. Li and T. W. Ling. *QED: a Novel Quaternary Encoding to completely avoid Re-Labeling in XML Updates.* In O. Herzog, H.-J. Schek, N. Fuhr, A. Chowdhury, and W. Teiken, eds., *Proceedings of the 2005 ACM CIKM International Conference on Information and Knowledge Management, Bremen, Germany, October 31 - November 5, 2005*, pp. 501–508 (ACM, 2005). URL `http://doi.acm.org/10.1145/1099554.1099692`.

[45] C. Li, T. W. Ling, and M. Hu. *Efficient Processing of Updates in Dynamic XML Data.* In L. Liu, A. Reuter, K.-Y. Whang, and J. Zhang, eds., *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, p. 13 (IEEE Computer Society, 2006). URL `http://doi.ieeecomputersociety.org/10.1109/ICDE.2006.58`.

[46] L. Xu, Z. Bao, and T. W. Ling. *A Dynamic Labeling Scheme Using Vectors.* In R. Wagner, N. Revell, and G. Pernul, eds., *Database and Expert Systems Applications, 18th International Conference, DEXA 2007, Regensburg, Germany, September 3-7, 2007, Proceedings*, vol. 4653 of *Lecture Notes in Computer Science*, pp. 130–140 (Springer, 2007). URL `http://dx.doi.org/10.1007/978-3-540-74469-6_14`.

[47] C. Li, T. W. Ling, and M. Hu. *Efficient Updates in Dynamic XML Data: from Binary String to Quaternary String.* VLDB J **17**(3), 573 (2008). URL `http://dx.doi.org/10.1007/s00778-006-0021-2`.

[48] M. F. O'Connor and M. Roantree. *EBSL: Supporting Deleted Node Label Reuse in XML.* In M.-L. Lee, J. X. Yu, Z. Bellahsene, and R. Unland, eds., *Database and XML Technologies - 7th International XML Database Symposium, XSym 2010, Singapore, September 17, 2010. Proceedings*, vol. 6309 of *Lecture Notes in Computer Science*, pp. 73–87 (Springer, 2010). URL `http://dx.doi.org/10.1007/978-3-642-15684-7`.

[49] C. Li, T. W. Ling, and M. Hu. *Reuse or Never Reuse the Deleted Labels in XML Query Processing Based on Labeling Schemes.* In M.-L. Lee, K.-L. Tan, and V. Wuwongse, eds., *Database Systems for Advanced Applications, 11th International Conference, DASFAA 2006, Singapore, April 12-15, 2006, Proceedings*, vol. 3882 of *Lecture Notes in Computer Science*, pp. 659–673 (Springer, 2006). URL `http://dx.doi.org/10.1007/11733836_46`.

[50] H.-K. Ko and S. Lee. *A Binary String Approach for Updates in Dynamic Ordered XML Data.* IEEE Trans. Knowl. Data Eng **22**(4), 602 (2010). URL `http://dx.doi.org/10.1109/TKDE.2009.87`.

[51] M. F. O'Connor and M. Roantree. *SCOOTER: A Compact and Scalable Dynamic Labeling Scheme for XML Updates.* In S. W. Liddle, K.-D. Schewe, A. M. Tjoa, and X. Zhou, eds., *Database and Expert Systems Applications - 23rd International Conference, DEXA 2012, Vienna, Austria, September 3-6, 2012. Proceedings, Part I*, vol. 7446 of *Lecture Notes in Computer Science*, pp. 26–40 (Springer, 2012). URL `http://dx.doi.org/10.1007/978-3-642-32600-4`.

[52] C. Ghezzi and D. Mandrioli. *Augmenting Parsers to Support Incrementality.* J. of the ACM **27**(3), 564 (1980).

[53] Wagner and Graham. *Efficient and Flexible Incremental Parsing.* ACMTOPLAS: ACM Transactions on Programming Languages and Systems **20** (1998).

[54] F. Jalili and J. H. Gallier. *Building Friendly Parsers.* In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pp. 196–206 (ACM SIGACT-SIGPLAN, Albuquerque, New Mexico, 1982).

[55] J.-M. Larchevêque. *Optimal Incremental Parsing.* ACM Transactions on Programming Languages and Systems **17**(1), 1 (1995).

[56] L. Petrone. *Reusing Batch Parsers as Incremental Parsers.* Lecture Notes in Computer Science **1026**, 111 (1995).

[57] A. M. Murching, Y. V. Prasad, and Y. N. Srikant. *Incremental Recursive Descent Parsing.* Comput. Lang **15**(4), 193 (1990). URL `http://dx.doi.org/10.1016/0096-0551(90)90020-P`.

[58] W. X. Li. *A Simple and Efficient Incremental LL(1) Parsing.* In M. Bartosek, J. Staudek, and J. Wiedermann, eds., *SOFSEM '95, 22nd Seminar on Current Trends in Theory and Practice of Informatics, Milovy, Czech Republic, November 23 - December 1, 1995, Proceedings*, vol. 1012 of *Lecture Notes in Computer Science*, pp. 399–404 (Springer, 1995). URL `http://dx.doi.org/10.1007/3-540-60609-2_24`.

[59] *XMLmind XML Editor.* Available online at http://www.xmlmind.com/.

[60] *XMLSpy XML Editor.* Available online at http://www.altova.com/xml-editor/.

[61] D. Barbosa, A. O. Mendelzon, L. Libkin, L. Mignet, and M. Arenas. *Efficient Incremental Validation of XML Documents.* In Z. M. Özsoyoglu and S. B. Zdonik, eds., *Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, 30 March - 2 April 2004, Boston, MA, USA*, pp. 671–682 (IEEE Computer Society, 2004). URL `http://doi.ieeecomputersociety.org/10.1109/ICDE.2004.1320036`.

[62] *XML Schema part 1: Structures* (2001). http://www.w3.org/TR/xmlschema-1/.

[63] Y. Papakonstantinou and V. Vianu. *DTD Inference for Views of XML Data.* In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pp. 35–46 (Dallas, Texas, 2000).

[64] M. Altinel and M. J. Franklin. *Efficient Filtering of XML Documents for Selective Dissemination of Information.* In A. El Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. Whang, eds., *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10–14, 2000, Cairo, Egypt*, pp. 53–64 (Morgan Kaufmann Publishers, pub-MORGAN-KAUFMANN:adr, 2000). URL `http://www.vldb.org/dblp/db/conf/vldb/AltinelF00.html`.

[65] I. Avila-campillo, T. J. Green, A. Gupta, M. Onizuka, D. Raven, and D. Suciu. *XMLTK: An XML Toolkit for Scalable XML Stream Processing.* In *In Proceedings of PLANX* (2002).

[66] C. Y. Chan, P. Felber, M. N. Garofalakis, and R. Rastogi. *Efficient Filtering of XML Documents with XPath Expressions.* In R. Agrawal and K. R. Dittrich, eds., *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*, pp. 235–244 (IEEE Computer Society, 2002). URL `http://doi.ieeecomputersociety.org/10.1109/ICDE.2002.994713`.

[67] Y. Diao, P. M. Fischer, M. J. Franklin, and R. To. *YFilter: Efficient and Scalable Filtering of XML Documents.* In R. Agrawal and K. R. Dittrich, eds., *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*, pp. 341–342 (IEEE Computer Society, 2002). URL `http://doi.ieeecomputersociety.org/10.1109/ICDE.2002.994748`.

[68] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. *Processing XML Streams with Deterministic Automata.* In D. Calvanese, M. Lenzerini, and R. Motwani, eds., *ICDT*, vol. 2572 of *Lecture Notes in Computer Science*, pp. 173–189 (Springer, 2003). URL `http://dx.doi.org/10.1007/3-540-36285-1_12`.

[69] A. K. Gupta and D. Suciu. *Stream Processing of XPath Queries with Predicates.* In ACM, ed., *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data 2003, San Diego, California, June 09–12, 2003*, pp. 419–430 (ACM Press, pub-ACM:adr, 2003).

[70] D. Olteanu, T. Kiesling, and F. Bry. *An Evaluation of Regular Path Expressions with Qualifiers against XML Streams.* In U. Dayal, K. Ramamritham, and T. M. Vijayaraman, eds., *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pp. 702–704 (IEEE Computer Society, 2003). URL http://doi.ieeecomputersociety.org/10.1109/ICDE.2003.1260841.

[71] F. Peng and S. S. Chawathe. *XPath Queries on Streaming Data.* In ACM, ed., *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data 2003, San Diego, California, June 09–12, 2003*, pp. 431–442 (ACM Press, pub-ACM:adr, 2003).

[72] Bar-Yossef, Fontoura, and Josifovski. *On the Memory Requirements of XPath Evaluation over XML Streams.* JCSS: Journal of Computer and System Sciences **73** (2007).

[73] B. Choi, M. Mahoui, and D. Wood. *On the Optimality of Holistic Algorithms for Twig Queries.* In V. Marík, W. Retschitzegger, and O. Stepánková, eds., *Database and Expert Systems Applications, 14th International Conference, DEXA 2003, Prague, Czech Republic, September 1-5, 2003, Proceedings*, vol. 2736 of *Lecture Notes in Computer Science*, pp. 28–37 (Springer, 2003). URL http://dx.doi.org/10.1007/978-3-540-45227-0_4.

[74] Grohe, Koch, and Schweikardt. *Tight Lower Bounds for Query Processing on Streaming and External Memory Data.* TCS: Theoretical Computer Science **380** (2007).

[75] G. Dong and J. Su. *Incremental maintenance of recursive views using relational calculus/SQL.* SIGMOD Record **29**(1), 44 (2000). URL http://doi.acm.org/10.1145/344788.344808.

[76] R. Paige. *Applications of Finite Differencing to Database Integrity Control and Query/Transaction Optimization.* In *Advances in Data Base Theory*, pp. 171–209 (1982).

[77] X. Qian and G. Widerhold. *Incremental Recomputation of Active Relational Expressions.* tkde **3**(3), 337 (1991).

[78] T. Griffin, L. Libkin, and H. Trickey. *An Improved Algorithm for the Incremental Recomputation of Active Relational Expressions.* IEEE Transactions on Knowledge and Data Engineering **9**(3), 508 (1997).

[79] J.-M. Nicolas and K. Yazdanian. *An Outline of BDGEN: A Deductive DBMS.* In *Information Processing*, pp. 711–717 (1983).

[80] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. *Maintaining Views Incrementally.* SIGMOD Record (ACM Special Interest Group on Management of Data) **22**(2), 157 (1993).

[81] I. S. Mumick and O. Shmueli. *Finiteness Properties of Database Queries.* In *Australian Database Conference*, pp. 274–288 (1993).

[82] I. S. Mumick and O. Shmueli. *Universal Finiteness and Satisfiability.* In ACM, ed., *PODS '94. Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 24–26, 1994, Minneapolis, MN*, vol. 13, pp. 190–200 (ACM Press, pub-ACM:adr, 1994).

[83] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, vol. 2 (Computer Science Press, 1989).

[84] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. *Answering Queries using Views.* In *14th ACM Symposium on Principles of Database Systems*, pp. 95–104 (San Jose, CA, 1995).

[85] J. Harrison and S. Dietrich. *Maintenance of Materialized Views in Deductive Databases: An Update Propagation Approach.* In *Workshop on Deductive Databases, Washington DC* (1992).

[86] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. *Efficiently Updating Materialized Views.* In *ACM SIGMOD* (1986).

[87] O. Shmueli and A. Itai. *Maintenance of Views.* In B. Yormark, ed., *SIGMOD'84, Proceedings of Annual Meeting*, pp. 240–255 (Boston, Massachusetts, 1984).

[88] N. Roussopoulos. *An Incremental Access Method for ViewCache: Concept, Algorithms, and Cost Analysis.* ACM Trans. on Database Sys. **16**(3), 535 (1991).

[89] S. Ceri and J. Widom. *Deriving Production Rules for Incremental View Maintenance.* In *Proc. 17th Intl. Conf. on Very Large Data Bases (VLDB)*, pp. 577–589 (ACM SIGMOD, Barcelona, Spain, 1991).

[90] V. Kuechenhoff. *On the Efficient Computation of the Difference between Consecutive Database States.* In *Proc. Conf. on Deductive and Object-oriented Databases* (Munich, Germany, 1991).

[91] Oracle. *SQL Reference.* www.oracle.com/technology/documentation/.

[92] T. Urpi and A. Olive. *A Method for Change Computation in Deductive Databases.* In *Proc. Int'l. Conf. on Very Large Data Bases*, p. 225 (Vancouver, BC, Canada, 1992).

[93] G. Dong and R. Topor. *Incremental Evaluation of Datalog Queries.* In *1992 Internat. Conference on Database Theory, Berlin* (1992).

[94] G. Dong and J. Su. *Incremental and Decremental Evaluation of Transitive Closure Queries by First-Order Queries.* In *Proc. 16th Australian Computer Science Conference* (1993).

[95] A. Gupta, H. V. Jagadish, and I. S. Mumick. *Data Integration Using Self-Maintainable Views.* Lecture Notes in Computer Science **1057**, 140 (1996).

[96] J. A. Blakeley, N. Coburn, and P. Larson. *Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates.* ACM Transactions on Database Systems **14**(3), 369 (1989).

[97] C. Elkan. *Independence of Logic Database Queries and Updates.* In *ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, Nashville* (1990).

[98] A. Y. Levy and Y. Sagiv. *Queries Independent of Updates.* In *19th Conference on Very Large Databases*, pp. 171–181 (Dublin, Ireland, 1993).

[99] F. W. Tompa and J. A. Blakeley. *Maintaining Materialized Views without Accessing Base Data.* Inform.Systems **13**(4) (1988).

[100] A. Gupta and J. A. Blakeley. *Maintaining Views using Materialized Views.* Unpublished document (1995).

[101] H. V. Jagadish, I. S. Mumick, and A. Silberschatz. *View Maintenance Issues for the Chronicle Data Model.* In *Proceedings of the ACM SIGACT-SIGMOD-SIGART PODS*, pp. 113–124 (San Jose, CA, 1995).

[102] A. Gupta. *Partial Information based Integrity Constraint Checking.* Ph.D. thesis, Stanford University (1994).

[103] M. Fernandez, A. Morishima, and D. Suciu. *Efficient Evaluation of XML Middle-Ware Queries.* In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, SIGMOD '01, pp. 103–114 (ACM, New York, NY, USA, 2001). URL `http://doi.acm.org/10.1145/375663.375674`.

[104] M. J. Carey, J. Kiernan, J. Shanmugasundaram, E. J. Shekita, and S. N. Subramanian. *XPERANTO: Middleware for Publishing Object-Relational Data as XML Documents.* In A. E. Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. Whang, eds., *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pp. 646–648 (Morgan Kaufmann, 2000). URL `http://www.vldb.org/conf/2000/P646.pdf`.

[105] P. Bohannon, S. Ganguly, H. F. Korth, P. P. S. Narayan, and P. Shenoy. *Optimizing View Queries in ROLEX to Support Navigable Result Trees.* In *VLDB*, pp. 119–130 (Morgan Kaufmann, 2002). URL http://www.vldb.org/conf/2002/S04P04.pdf.

[106] A. Deutsch, M. F. Fernandez, D. Florescu, A. Y. Levy, and D. Suciu. *XML-QL: A Query Language for XML.* In *WWW The Query Language Workshop (QL)* (Cambridge, MA, 1998). http://www.w3.org/TR/1998/NOTE-xml-ql-19980819/.

[107] M. F. Fernandez, W.-C. Tan, and D. Suciu. *SilkRoute: Trading between Relations and XML.* In *Int'l World Wide Web Conf. (WWW)* (Amsterdam, Netherlands, 2000).

[108] J. Shanmugasundaram, E. J. Shekita, R. Barr, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald. *Efficiently Publishing Relational Data as XML documents.* In *VLDB* (Cairo, Egypt, 2000).

[109] P. Bohannon, B. Choi, and W. Fan. *Incremental Evaluation of Schema-Directed XML Publishing.* In ACM, ed., *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data 2004, Paris, France, June 13–18, 2004*, pp. 503–514 (ACM Press, pub-ACM:adr, 2004).

[110] M. Benedikt, C. Y. Chan, W. Fan, R. Rastogi, S. Zheng, and A. Zhou. *DTD-Directed Publishing with Attribute Translation Grammars.* In *VLDB*, pp. 838–849 (Morgan Kaufmann, 2002). URL http://www.vldb.org/conf/2002/S23P03.pdf.

[111] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, and J. B. Rothnie, Jr. *Query Processing in a System for Distributed Databases (SDD-1).* ACM Transactions on Database Systems **6**(4), 602 (1981).

[112] M. Benedikt, C.-Y. Chan, W. Fan, J. Freire, and R. Rastogi. *Capturing both Types and Constraints in Data Integration.* In ACM, ed., *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data 2003, San Diego, California, June 09–12, 2003*, pp. 277–288 (ACM Press, pub-ACM:adr, 2003).

[113] IBM. *IBM DB2 Universal Database SQL Reference.* www-306.ibm.com/software/data/db2/.

[114] H. Liefke and S. B. Davidson. *View Maintenance for Hierarchical Semistructured Data.* In Y. Kambayashi, M. K. Mohania, and A. M. Tjoa, eds., *Data Warehousing and Knowledge Discovery, Second International Conference, DaWaK 2000, London, UK, September 4-6, 2000, Proceedings*, vol. 1874 of *Lecture Notes in Computer Science*, pp. 114–125 (Springer, 2000). URL `http://dx.doi.org/10.1007/3-540-44466-1_12`.

[115] M. EL-Sayed, L. Wang, L. Ding, and E. A. Rundensteiner. *An Algebraic Approach for Incremental Maintenance of Materialized XQuery Views.* In R. Chiang and E.-P. Lim, eds., *Proceedings of the Fourth International Workshop on Web Information and Data Management (WIDM-02)*, pp. 88–91 (ACM Press, New York, 2002).

[116] K. Dimitrova, M. El-Sayed, and E. A. Rundensteiner. *Order-Sensitive View Maintenance of Materialized XQuery Views.* In I.-Y. Song, S. W. Liddle, T. W. Ling, and P. Scheuermann, eds., *Conceptual Modeling - ER 2003, 22nd International Conference on Conceptual Modeling, Chicago, IL, USA, October 13-16, 2003, Proceedings*, vol. 2813 of *Lecture Notes in Computer Science*, pp. 144–157 (Springer, 2003). URL `http://dx.doi.org/10.1007/978-3-540-39648-2_14`.

[117] C. Re, J. Siméon, and M. F. Fernández. *A Complete and Efficient Algebraic Compiler for XQuery.* In L. Liu, A. Reuter, K.-Y. Whang, and J. Zhang, eds., *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, p. 14 (IEEE Computer Society, 2006). URL `http://doi.ieeecomputersociety.org/10.1109/ICDE.2006.6`.

[118] G. Ghelli, N. Onose, K. H. Rose, and J. Siméon. *A Better Semantics for XQuery with Side-Effects.* In M. Arenas and M. I. Schwartzbach, eds., *Database Programming Languages, 11th International Symposium, DBPL 2007, Vienna, Austria, September 23-24, 2007, Revised Selected Papers*, vol. 4797 of *Lecture Notes in Computer Science*, pp. 81–96 (Springer, 2007). URL `http://dx.doi.org/10.1007/978-3-540-75987-4_6`.

[119] G. Ghelli, C. Re, and J. Siméon. *XQuery!: An XML Query Language with Side Effects.* In T. Grust, H. Höpfner, A. Illarramendi, S. Jablonski, M. Mesiti, S. Müller, P.-L. Patranjan, K.-U. Sattler, M. Spiliopoulou, and J. Wijsen, eds., *Current Trends in*

Database Technology - EDBT 2006, EDBT 2006 Workshops PhD, DataX, IIDB, IIHA, ICSNW, QLQP, PIM, PaRMA, and Reactivity on the Web, Munich, Germany, March 26-31, 2006, Revised Selected Papers, vol. 4254 of *Lecture Notes in Computer Science*, pp. 178–191 (Springer, 2006). URL http://dx.doi.org/10.1007/11896548_17.

[120] A. Sawires, J. Tatemura, O. Po, D. Agrawal, and K. S. Candan. *Incremental Maintenance of Path Expression Views.* In F. Özcan, ed., *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pp. 443–454 (ACM, 2005). URL http://doi.acm.org/10.1145/1066157.1066208.

[121] A. Sawires, J. Tatemura, O. Po, D. Agrawal, A. E. Abbadi, and K. S. Candan. *Maintaining XPath views in loosely coupled systems.* In U. Dayal, K.-Y. Whang, D. B. Lomet, G. Alonso, G. M. Lohman, M. L. Kersten, S. K. Cha, and Y.-K. Kim, eds., *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pp. 583–594 (ACM, 2006). URL http://www.vldb.org/conf/2006/p583-sawires.pdf.

[122] *World Wide Web Consortium (W3C).* http://www.w3.org/.

[123] S. Abiteboul, O. Benjelloun, and T. Milo. *The Active XML Project: an overview.* VLDB J **17**(5), 1019 (2008). URL http://dx.doi.org/10.1007/s00778-007-0049-y.

[124] S. Abiteboul and B. Marinoiu. *Distributed Monitoring of Peer to Peer Systems.* In I. Fundulaki and N. Polyzotis, eds., *9th ACM International Workshop on Web Information and Data Management (WIDM 2007), Lisbon, Portugal, November 9, 2007*, pp. 41–48 (ACM, 2007). URL http://doi.acm.org/10.1145/1316902.1316910.

[125] R. Ennals and D. Gay. *User-Friendly Functional Programming for Web Mashups.* In R. Hinze and N. Ramsey, eds., *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007,*

Freiburg, Germany, October 1-3, 2007, pp. 223–234 (ACM, 2007). URL `http://doi.acm.org/10.1145/1291151.1291187`.

[126] S. Abiteboul, P. Bourhis, and B. Marinoiu. *Incremental view maintenance for active documents.* In O. Boucelma, M.-S. Hacid, T. Libourel, and J.-M. Petit, eds., *23èmes Journées Bases de Données Avancées, BDA 2007, Marseille, 23-26 Octobre 2007, Actes (Informal Proceedings)* (2007).

[127] S. Abiteboul, P. Bourhis, and B. Marinoiu. *Efficient maintenance techniques for views over active documents.* In *EDBT* (2009).

[128] Balbin and K. Ramamohanarao. *A Generalization of the Differential Approach to Recursive Query Evaluation.* The Journal of Logic Programming **4**(3), 259 (1987).

[129] C. Beeri and R. Ramakrishnan. *On the Power of Magic.* In *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Sys.*, p. 269 (San Diego, CA, 1987).

[130] L. Vieille. *Recursive Query Processing: The Power of Logic.* Theoretical Computer Science **69**(2) (1989).

[131] P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz. *Constraint Query Languages.* In *Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Sys.* (Nashville, TN, 1990).

[132] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases* (Addison-Wesley, Reading, Massachusetts, 1995).

[133] U. Dayal and P. Bernstein. *On the Correct Translation of Update Operations on Relational Views.* ACM Transactions on Database Systems **7**(3) (1982).

[134] U. Dayal and P. A. Bernstein. *On the Updatability of Network Views - Extending Relational View Theory to the Network Model.* Inf. Sys. **7**(1), 29 (1982).

[135] Cosmadakis and Papadimitriou. *Updates of Relational Views.* JACM: Journal of the ACM **31** (1984).

[136] F. Bancilhon and N. Spyratos. *Update semantics of relational views.* ACM Trans. Database Syst. **6**(4) (1981).

[137] F. Bancilhon and N. Spyratos. *Update Semantics of Relational Views.* ACM Trans. on Database Sys. **6**(4), 557 (1981).

[138] W. W. Armstrong. *Dependency Structures of Database Relationships.* In *Proc. IFIP '74*, pp. 580–583 (North Holland, 1974).

[139] E. F. Codd. *A Relational Model of Data for Large Shared Data Banks.* Comm. ACM **13**(6), pages 377 (1970).

[140] A. M. Keller. *Algorithms for Translating View Updates to Database Updates for Views Involving Selections, Projections, and Joins.* ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, Portland OR, ACM SIGACT and SIGMOD (1985).

[141] L. Wang, M. Mulchandani, and E. A. Rundensteiner. *Updating XQuery Views Published over Relational Data: A Roundtrip Case Study.* In Z. Bellahsene, A. B. Chaudhri, E. Rahm, M. Rys, and R. Unland, eds., *Database and XML Technologies, First International XML Database Symposium, XSym 2003, Berlin, Germany, September 8, 2003, Proceedings*, vol. 2824 of *Lecture Notes in Computer Science*, pp. 223–237 (Springer, 2003). URL `http://dx.doi.org/10.1007/978-3-540-39429-7_15`.

[142] X. Zhang, M. Mulchandani, S. Christ, B. Murphy, and E. A. Rundensteiner. *Rainbow: Mapping-Driven XQuery Processing System.* In M. Franklin, B. Moon, and A. Ailamaki, eds., *Proceedings of the ACM SIGMOD International Conference on Management of Data, June 3–6, 2002, Madison, WI, USA*, pp. 614–614 (ACM Press, pub-ACM:adr, 2002).

[143] X. Zhang, K. Dimitrova, L. Wang, M. E. Sayed, B. Murphy, B. Pielech, M. Mulchandani, L. Ding, and E. A. Rundensteiner. *Rainbow: Multi-XQuery Optimization using Materialized XML Views.* In ACM, ed., *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data 2003, San Diego, California, June 09–12, 2003*, pp. 671–671 (ACM Press, pub-ACM:adr, 2003).

[144] V. P. Braganholo, S. B. Davidson, and C. A. Heuser. *From XML View Updates to Relational View Updates: Old Solutions to a New Problem.* In M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley, and K. B. Schiefer, eds., *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, pp. 276–287 (Morgan Kaufmann, 2004). URL http://www.vldb.org/conf/2004/RS7P3.PDF.

[145] V. P. Braganholo, S. B. Davidson, and C. A. Heuser. *On the Updatability of XML Views over Relational Databases.* In V. Christophides and J. Freire, eds., *International Workshop on Web and Databases, San Diego, California, June 12-13, 2003*, pp. 31–36 (2003). URL http://www.cse.ogi.edu/webdb03/papers/06.pdf.

[146] L. Wang, E. A. Rundensteiner, and M. Mani. *U-Filter: A Lightweight XML View Update Checker.* In L. Liu, A. Reuter, K.-Y. Whang, and J. Zhang, eds., *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, p. 126 (IEEE Computer Society, 2006). URL http://doi.ieeecomputersociety.org/10.1109/ICDE.2006.163.

[147] L. Wang and E. A. Rundensteiner. *On the Updatability of XML Views Published over Relational Data.* In P. Atzeni, W. W. Chu, H. Lu, S. Zhou, and T. W. Ling, eds., *Conceptual Modeling - ER 2004, 23rd International Conference on Conceptual Modeling, Shanghai, China, November 2004, Proceedings*, vol. 3288 of *Lecture Notes in Computer Science*, pp. 795–809 (Springer, 2004). URL http://dx.doi.org/10.1007/978-3-540-30464-7_59.

[148] L. Wang, E. A. Rundensteiner, and M. Mani. *Updating XML Views Published over Relational Databases: Towards the Existence of a Correct Update Mapping.* Data Knowl. Eng **58**(3), 263 (2006). URL http://dx.doi.org/10.1016/j.datak.2005.07.003.

[149] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. *Relational Databases for Querying XML Documents: Limitations and Opportunities.* In *Proceedings of VLDB* (Edinburgh, UK, 1999).

[150] L. Chen, A. Gupta, and M. E. Kurul. *Stack-Based Algorithms for Pattern Matching on DAGs.* In K. Böhm, C. S. Jensen, L. M. Haas, M. L. Kersten, P.-Å. Larson, and B. C. Ooi, eds., *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pp. 493–504 (ACM, 2005). URL http://www.vldb2005.org/program/paper/wed/p493-chen.pdf.

[151] R. Schenkel, A. Theobald, and G. Weikum. *Efficient Creation and Incremental Maintenance of the HOPI Index for Complex XML Document Collections.* In K. Aberer, M. J. Franklin, and S. Nishio, eds., *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*, pp. 360–371 (IEEE Computer Society, 2005). URL http://doi.ieeecomputersociety.org/10.1109/ICDE.2005.57.

[152] Microsoft. *SQL Server. MSDN Library.* msdn.microsoft.com/en-us/sqlserver/.

[153] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. *Tree pattern query minimization.* VLDB J. **11**(4) (2002).

[154] A. Arion, V. Benzaken, and I. Manolescu. *XML access modules: Towards physical data independence in XML databases.* In *XIME-P* (2005).

[155] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. *Structural joins: A primitive for efficient XML query pattern matching.* In *ICDE* (2002).

[156] F. Cavalieri, G. Guerrini, and M. Mesiti. *Dynamic reasoning on xml updates.* In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT/ICDT '11, pp. 165–176 (ACM, New York, NY, USA, 2011). URL http://doi.acm.org/10.1145/1951365.1951387.

[157] M. Benedikt and J. Cheney. *Destabilizers and independence of XML updates.* In *VLDB* (2010).

[158] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. *XMark: A benchmark for XML data management.* In *VLDB* (2002).

[159] M. Franceschet. *XPathMark: An XPath benchmark for the XMark generated data.* In *XSym* (2005).

A

Exploiting Schema Information for Pruning

In some situations XML schemas may be available. In these cases it can't be expected that all updates adhere to the schema. Therefore, if a schema is present it can be used to reject view maintenance updates which violate the schema or optimise the propagation of violation free updates.

DTDs are represented as extended context-free grammars (CFGs). Each rule's right-hand side consists of a regular expression from an alphabet of terminal and non-terminal symbols. For example, Figure A.1 describes two DTDs. In this Figure, a, b, c, d1, d2 and x are terminal symbols and AS and BS are non-terminal symbols. DTD d1 (a) contains mandatory edges, whereas DTD d2 (b) contains concatenation, disjunction and recursion.

$$\begin{array}{lcl}
\mathsf{d1} & \rightarrow & \mathsf{AS} \\
\mathsf{AS} & \rightarrow & \mathsf{a+} \\
\mathsf{a} & \rightarrow & \mathsf{BS} \\
\mathsf{BS} & \rightarrow & \mathsf{b+} \\
\mathsf{b} & \rightarrow & \mathsf{c} \\
\mathsf{c} & \rightarrow & \epsilon
\end{array}$$

(a) DTD d1

$$\begin{array}{lcl}
\mathsf{d2} & \rightarrow & \mathsf{AS} \\
\mathsf{AS} & \rightarrow & \mathsf{(a,\ b,\ c)+} \\
\mathsf{a} & \rightarrow & \mathsf{BS} \\
\mathsf{BS} & \rightarrow & \mathsf{x}\ |\epsilon \\
\mathsf{x} & \rightarrow & \mathsf{x}\ |\epsilon \\
\mathsf{b} & \rightarrow & \epsilon \\
\mathsf{c} & \rightarrow & \epsilon
\end{array}$$

(b) DTD d2

FIGURE A.1: **Sample DTDs, expressed as CFGs.**

A.1 Examples

Example A.1.1. *Consider the view v_1 from Example 4.1.2 and an insertion u_5, adding the following XML fragment:*

$$xml_5 = \langle a\rangle\langle b\rangle\langle /b\rangle\langle /a\rangle$$

Applying the update would make the document invalid with respect to the DTD in Figure A.1(a), since a c element is missing under b. More generally, from the DTD d1*, one can derive that the following statement must hold for any newly inserted XML tree:*

$$\Delta_c^+ = \emptyset \Rightarrow \Delta_b^+ = \emptyset$$

Since this does not hold on the update u_5, it is rejected due to its attempted schema violation.

The same consideration applies to Figure A.1(b), in which a $d2$ element must have as children the concatenation of a, b and c. Therefore, any insertion of an a element under the root $d2$ must occur with b and c elements.

Example A.1.2. *The DTD in Figure A.1(b) implies that the following statement must hold on any XML forest inserted under a given node:*

$$\Delta_a^+ \neq \emptyset \Rightarrow (\Delta_b^+ \neq \emptyset \wedge \Delta_c^+ \neq \emptyset)$$

More generally, from the DTD rules, one can infer a set of constraints on the Δ^+ tables, and check them before applying the update. If any constraint is violated, the update is rejected.

B
Test Set

These updates have been used to test the different kinds of XPath expressions that could be present in identifying the target node(s) for deletions or insertions. These updates were largely inspired by the XPathMark benchmark [159] and the views were created such that they would be affected by these updates. These updates are the expressions that can be represented within XML access modules [154], which are used to represent views. The following kinds of XPath expressions are supported:

- L: Linear path expression
- LB: Linear with Boolean filter
- A: AND predicate (pipeline the filters)
- O: OR predicate (union the paths)

- AO: AND + OR predicate (combination of the two former cases)

B.1 Linear Path Expression Updates

X1_L: insert_name

– For each person add a new name

```
let $c:= doc (“auction.xml”)
for $person in $c/site/people/person
insert ⟨name⟩ Martin
        ⟨name⟩ and ⟨/name⟩
        ⟨name⟩ some ⟨/name⟩
        ⟨name⟩ test ⟨/name⟩
        ⟨name⟩ nodes ⟨/name⟩
      ⟨/name⟩
```

X2 _L: insert_increase

– For each bidder add a new increase

```
let $c:= doc (“auction.xml”)
for $bidder in $c//open_auction/bidder
insert ⟨increase⟩ inserted 100.00
        ⟨increase⟩ and ⟨/increase⟩
        ⟨increase⟩ some ⟨/increase⟩
        ⟨increase⟩ test ⟨/increase⟩
        ⟨increase⟩ nodes ⟨/increase⟩
      ⟨/increase⟩
```

E6 _L: insert_item

– For each item insert a new item inside it

```
let $c:= doc (“auction.xml”)
for $item in $c/site/regions/*/item
insert ⟨item⟩
          ⟨location⟩ Unknown ⟨/location⟩
          ⟨quantity⟩ 1 ⟨/quantity⟩
          ⟨name⟩ E6_L Item ⟨/name⟩
          ⟨payment⟩ Creditcard, Personal Check, Cash ⟨/payment⟩
       ⟨/item⟩
```

X17 _L: insert_item

– For each item insert a new item inside it

```
let $c:= doc (“auction.xml”)
for $item in $c/site/regions//item
insert ⟨item⟩
          ⟨location⟩ Unknown ⟨/location⟩
          ⟨quantity⟩ 1 ⟨/quantity⟩
          ⟨name⟩ X17_L Item ⟨/name⟩
          ⟨payment⟩ Creditcard, Personal Check, Cash ⟨/payment⟩
          ⟨description⟩ Test description ⟨/description⟩
       ⟨/item⟩
```

B.2 Linear with Boolean Filter Updates

B7 _LB: insert_name

– For each person with a profile with a salary attribute add a new name

```
let $c:= doc (“auction.xml”)
```

```
for $name in $c//person[profile/@income]
insert ⟨name⟩ Jim
        ⟨name⟩ and ⟨/name⟩
        ⟨name⟩ some ⟨/name⟩
        ⟨name⟩ test ⟨/name⟩
        ⟨name⟩ nodes ⟨/name⟩
      ⟨/name⟩
```

B3 _LB: insert_name

– For each open auction with a reserve add a new name

```
let $c:= doc (“auction.xml”)
for $increase in $c/site/open_auctions/open_auction[reserve]/bidder
insert ⟨increase⟩ inserted 4.50
        ⟨increase⟩ and ⟨/increase⟩
        ⟨increase⟩ some ⟨/increase⟩
        ⟨increase⟩ test ⟨/increase⟩
        ⟨increase⟩ nodes ⟨/increase⟩
      ⟨/increase⟩
```

B5 _LB: insert_item

– For each item insert a new item inside it

```
let $c:= doc (“auction.xml”)
for $item in $c/site/regions/*/item[name]
insert ⟨item⟩
        ⟨location⟩ Unknown ⟨/location⟩
        ⟨quantity⟩ 1 ⟨/quantity⟩
        ⟨name⟩ B5_LB Item ⟨/name⟩
        ⟨payment⟩ Creditcard, Personal Check, Cash ⟨/payment⟩
```

```
    ⟨/item⟩
```

B.3 AND Predicate Updates

– For each person with a profile with a gender and a profile with an age add a new name

X6 _A: insert_name

```
let $c:= doc (“auction.xml”)
for $name in $c/site/people/person[phone and homepage]
insert ⟨name⟩ Mimma
        ⟨name⟩ and ⟨/name⟩
        ⟨name⟩ some ⟨/name⟩
        ⟨name⟩ test ⟨/name⟩
        ⟨name⟩ nodes ⟨/name⟩
    ⟨/name⟩
```

X3 _A: insert_increase

– For each open auction with privacy and a bidder add an increase

```
let $c:= doc (“auction.xml”)
for $increase in $c/site/open_auctions/open_auction[privacy and bidder]/bidder
insert ⟨increase⟩ inserted 150.00
        ⟨increase⟩ and ⟨/increase⟩
        ⟨increase⟩ some ⟨/increase⟩
        ⟨increase⟩ test ⟨/increase⟩
        ⟨increase⟩ nodes ⟨/increase⟩
    ⟨/increase⟩
```

B1 _A: insert_item

– For each item from North America and South America insert a new item inside it.

```
let $c:= doc ("auction.xml")
for $item in $c/site/regions[namerica and samerica]//item
insert ⟨item⟩
          ⟨location⟩ Canada ⟨/location⟩
          ⟨quantity⟩ 1 ⟨/quantity⟩
          ⟨name⟩ B1_A Item ⟨/name⟩
          ⟨payment⟩ Creditcard, Personal Check, Cash ⟨/payment⟩
       ⟨/item⟩
```

X20 _A: insert_item

– For each item with a description and a name insert a new item inside it

```
let $c:= doc ("auction.xml")
for $item in $c/site/regions//item[description and name]
insert ⟨item⟩
          ⟨location⟩ Japan ⟨/location⟩
          ⟨quantity⟩ 1 ⟨/quantity⟩
          ⟨name⟩ X20_A Item ⟨/name⟩
          ⟨payment⟩ Creditcard, Personal Check, Cash ⟨/payment⟩
          ⟨description⟩ Test description ⟨/description⟩
       ⟨/item⟩
```

B.4 OR Predicate Updates

A7 _O: insert_name

– For each person with a phone or homepage add a new name

```
let $c:= doc ("auction.xml")
```

```
for $name in $c/site/people/person[phone or homepage]
insert ⟨name⟩ Ioana
        ⟨name⟩ and ⟨/name⟩
        ⟨name⟩ some ⟨/name⟩
        ⟨name⟩ test ⟨/name⟩
        ⟨name⟩ nodes ⟨/name⟩
    ⟨/name⟩
```

X4 _O: insert_increase

– For each open auction with a bidder or privacy add a new increase

```
let $c:= doc (“auction.xml”)
for $increase in $c/site/open_auctions/open_auction[bidder or privacy]/bidder
insert ⟨increase⟩ inserted 200.00
        ⟨increase⟩ and ⟨/increase⟩
        ⟨increase⟩ some ⟨/increase⟩
        ⟨increase⟩ test ⟨/increase⟩
        ⟨increase⟩ nodes ⟨/increase⟩
    ⟨/increase⟩
```

X7 _O: insert_item

– For each item with a description or a name insert a new item inside it

```
let $c:= doc (“auction.xml”)
for $item in $c/site/regions//item[description or name]
insert ⟨item⟩
        ⟨location⟩ Unknown ⟨/location⟩
        ⟨quantity⟩ 1 ⟨/quantity⟩
        ⟨name⟩ X7_O Item ⟨/name⟩
        ⟨payment⟩ Creditcard, Personal Check, Cash ⟨/payment⟩
        ⟨/item⟩
```

B1 _O: insert_item

– For each item with a description or a name insert a new item inside it

```
let $c:= doc ("auction.xml")
for $item in $c/site/regions[namerica or samerica]//item
insert ⟨item⟩
           ⟨location⟩ Canada ⟨/location⟩
           ⟨quantity⟩ 1 ⟨/quantity⟩
           ⟨name⟩ B1_O Item ⟨/name⟩
           ⟨payment⟩ Creditcard, Personal Check, Cash ⟨/payment⟩
           ⟨description⟩ Test description ⟨/description⟩
        ⟨/item⟩
```

B.5 AND + OR Predicate Updates

A8 _AO: insert_name

– For each person with an address AND (phone OR homepage) AND (creditcard OR profile)

```
let $c:= doc ("auction.xml")
for $name in $c/site/people/person[address and (phone or homepage) and (creditcard or
profile)]
insert ⟨name⟩ Angela
           ⟨name⟩ and ⟨/name⟩
           ⟨name⟩ some ⟨/name⟩
           ⟨name⟩ test ⟨/name⟩
           ⟨name⟩ nodes ⟨/name⟩
        ⟨/name⟩
```

X5 _AO: insert_increase

– For each open auction with a current AND (bidder OR reserve) add a new increase

```
let $c:= doc (“auction.xml”)
for $increase in $c/site/open_auctions/open_auction[current and (bidder or reserve)]/bidder
insert ⟨increase⟩ inserted 250.00
        ⟨increase⟩ and ⟨/increase⟩
        ⟨increase⟩ some ⟨/increase⟩
        ⟨increase⟩ test ⟨/increase⟩
        ⟨increase⟩ nodes ⟨/increase⟩
    ⟨/increase⟩
```

X8 _AO: insert_item

– For each item with a description AND (name OR mailbox) insert a new item inside it

```
let $c:= doc (“auction.xml”)
for $item in $c/site/regions//item[description and (name or mailbox)]
insert ⟨item⟩
        ⟨location⟩ New Zealand ⟨/location⟩
        ⟨quantity⟩ 1 ⟨/quantity⟩
        ⟨name⟩ X8_AO Item ⟨/name⟩
        ⟨payment⟩ Creditcard, Personal Check, Cash ⟨/payment⟩
    ⟨/item⟩
```

C

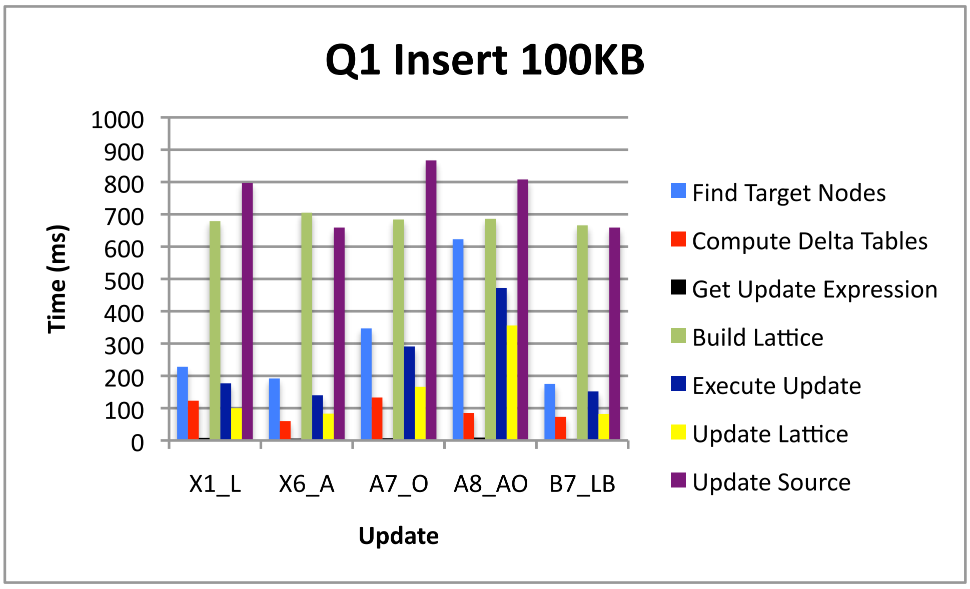
Experiments

C.1 View Maintenance

C.1.1 Insert

Query	TN	DT	UE	BL	EU	UL	US
X1_L	228	123	8	679	177	100	797
X6_A	192	60	6	705	140	83	659
A7_O	347	133	7	684	291	166	867
A8_AO	623	85	9	686	472	356	808
B7_LB	175	73	5	666	152	82	659

Table C.1: Q1 Insert 100Kb



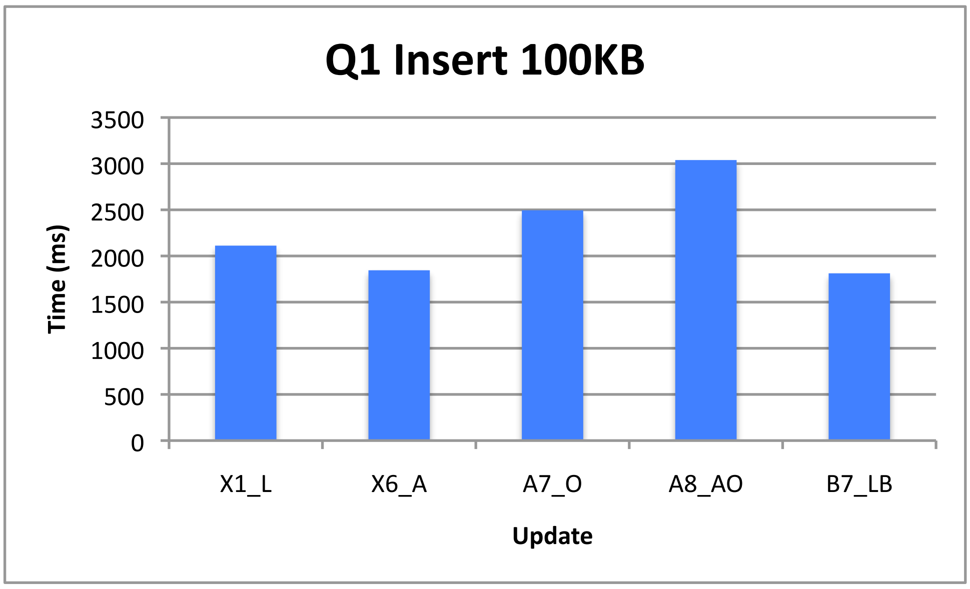


FIGURE C.1: Q1 Insert Graph 100KB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	789	363	6	1360	338	160	1104
X6_A	604	104	9	1425	188	121	1034
A7_O	1173	371	10	1388	399	224	1416
A8_AO	1832	199	6	1446	621	293	1899
B7_LB	606	260	11	1378	270	127	1079

TABLE C.2: Q1 Insert 500Kb

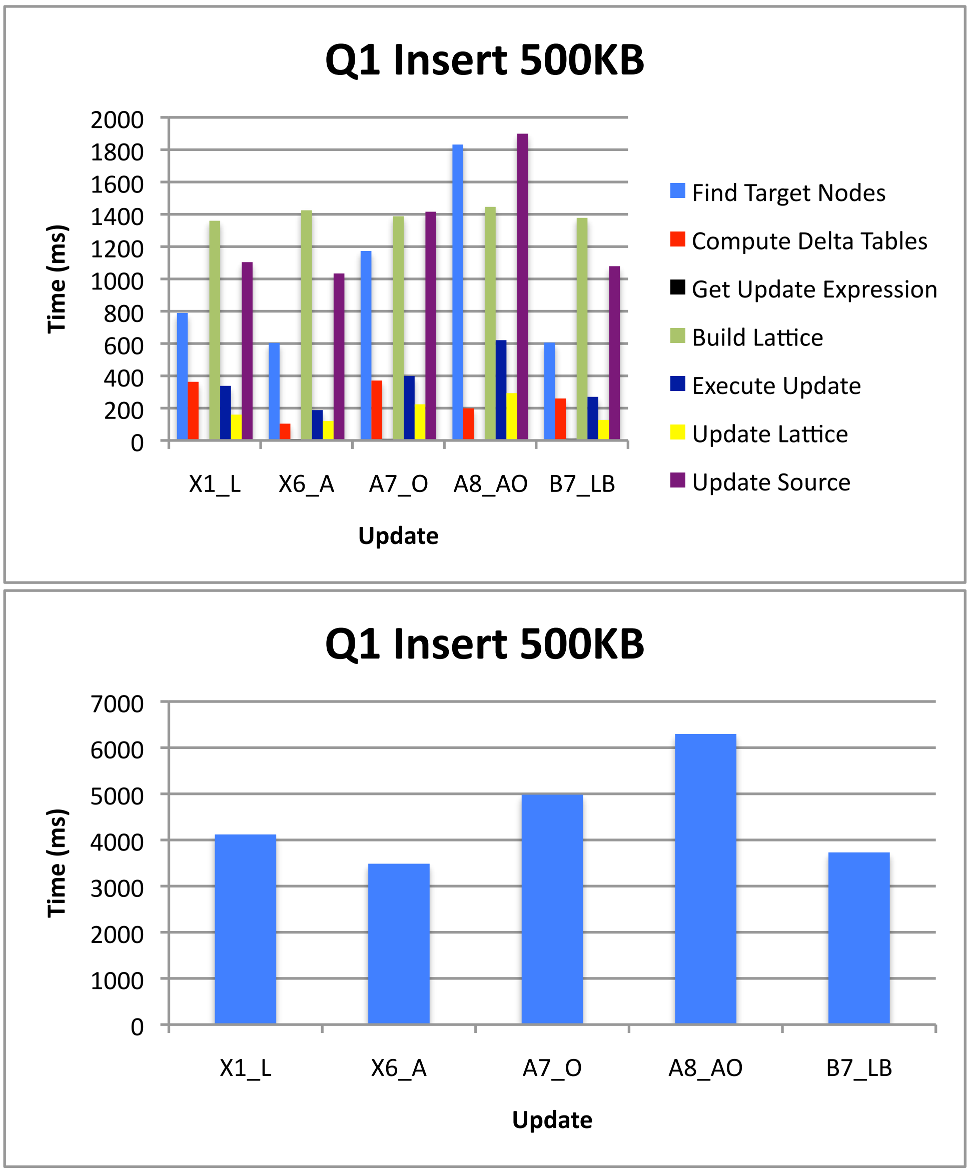
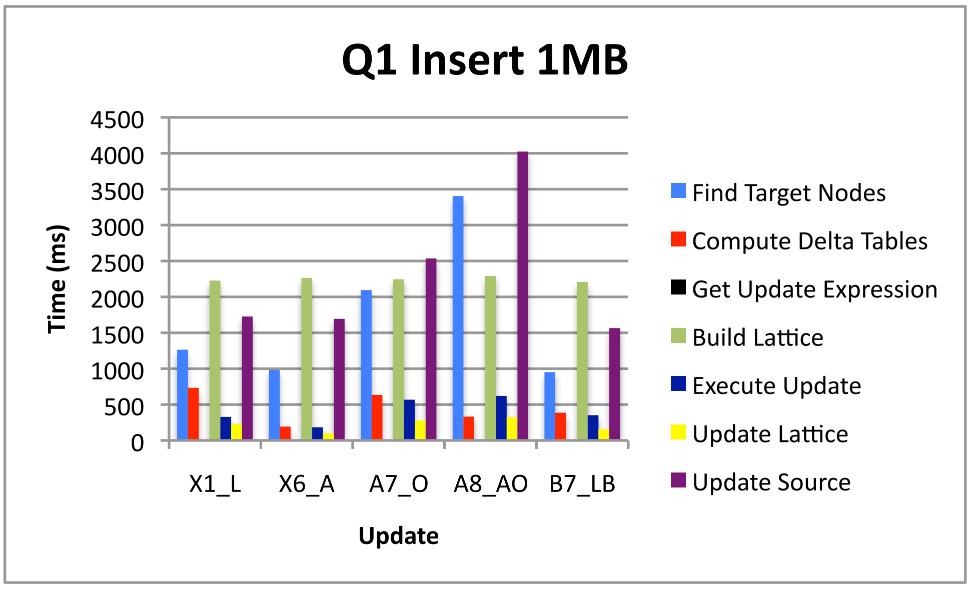


Figure C.2: Q1 Insert Graph 500KB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	1264	733	8	2227	328	231	1727
X6_A	982	194	6	2264	184	99	1694
A7_O	2095	634	8	2246	568	280	2536
A8_AO	3404	332	9	2291	620	324	4024
B7_LB	952	386	6	2208	352	157	1565

Table C.3: Q1 Insert 1Mb



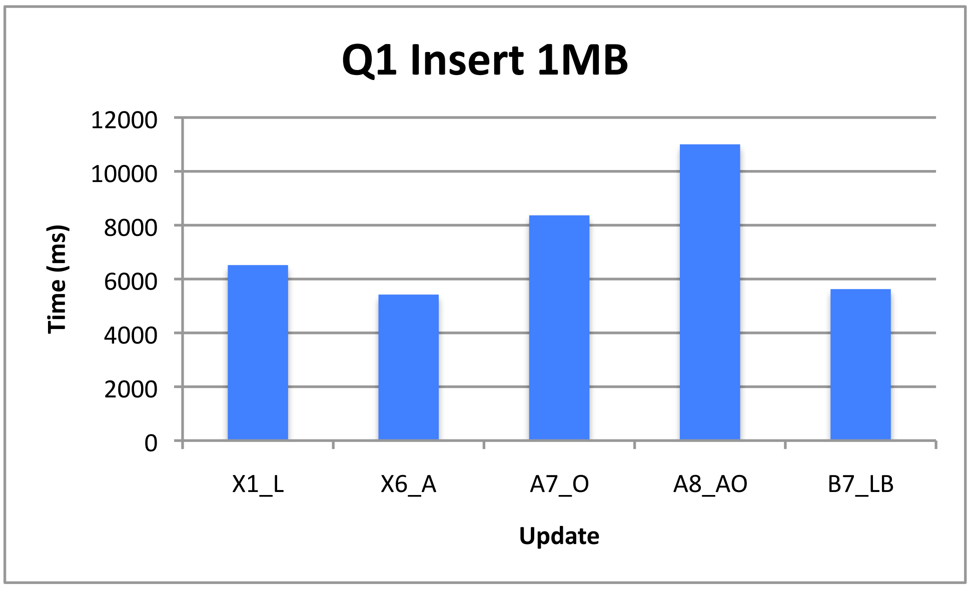


FIGURE C.3: Q1 Insert Graph 1MB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	22231	5696	10	17241	3539	2431	69552
X6_A	11244	1592	9	19007	1222	853	72600
A7_O	25274	5673	13	17261	4132	2694	137199
A8_AO	37412	3028	9	17156	3542	2898	258804
B7_LB	10663	2996	11	17223	1818	1332	66565

TABLE C.4: Q1 Insert 10Mb

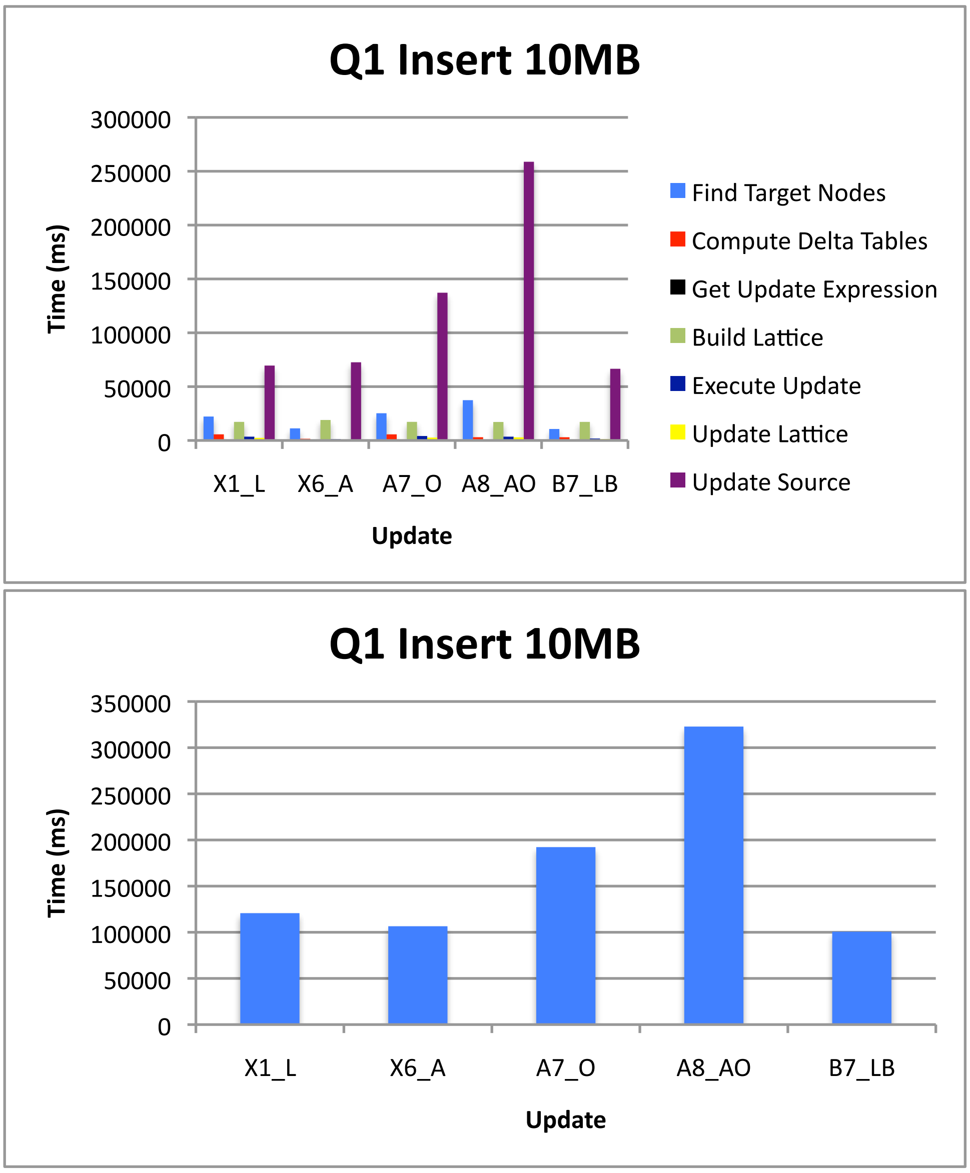


Figure C.4: Q1 Insert Graph 10MB

Query	TN	DT	UE	BL	EU	UL	US
X2_L	262	331	13	483	281	0	791
X3_A	266	160	6	546	218	0	664
X4_O	552	404	5	472	519	0	841
X5_AO	538	407	7	477	458	0	852
B3_LB	252	156	6	538	215	0	659

Table C.5: Q2 Insert 100Kb

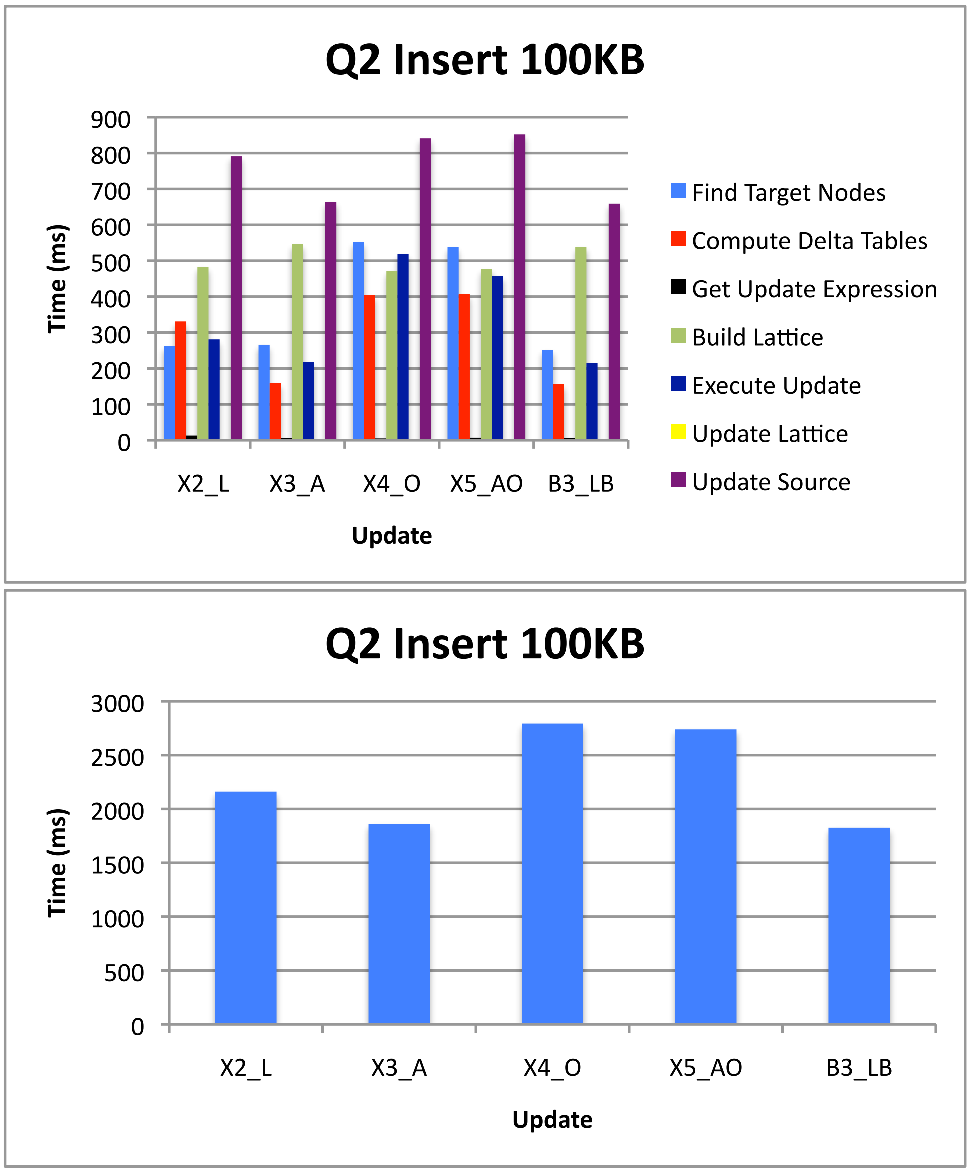
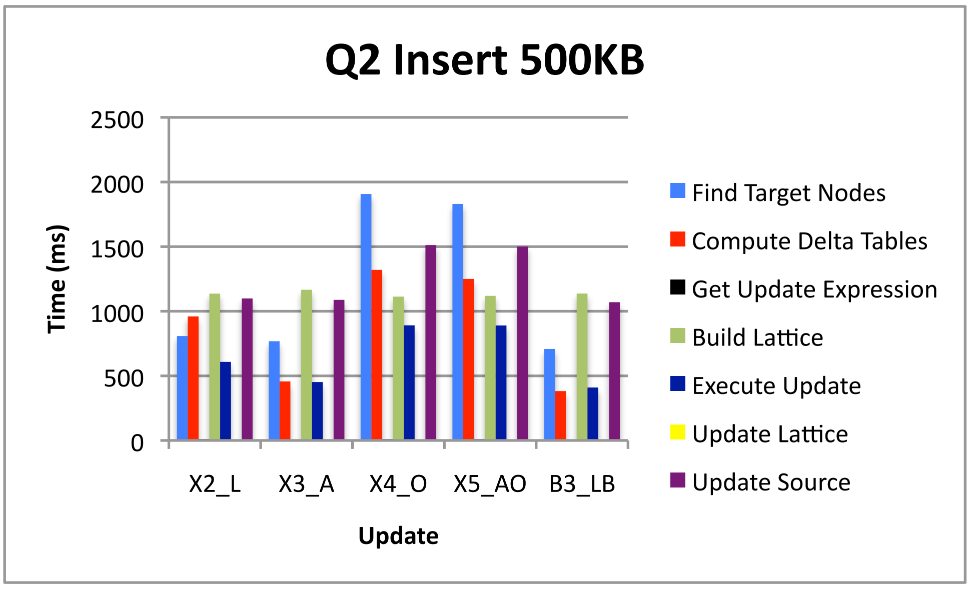


FIGURE C.5: Q2 Insert Graph 100KB

Query	TN	DT	UE	BL	EU	UL	US
X2_L	808	960	10	1136	608	0	1099
X3_A	768	457	6	1166	452	0	1088
X4_O	1907	1320	10	1113	891	0	1512
X5_AO	1830	1250	10	1119	890	0	1501
B3_LB	708	382	6	1137	410	1	1070

TABLE C.6: Q2 Insert 500Kb



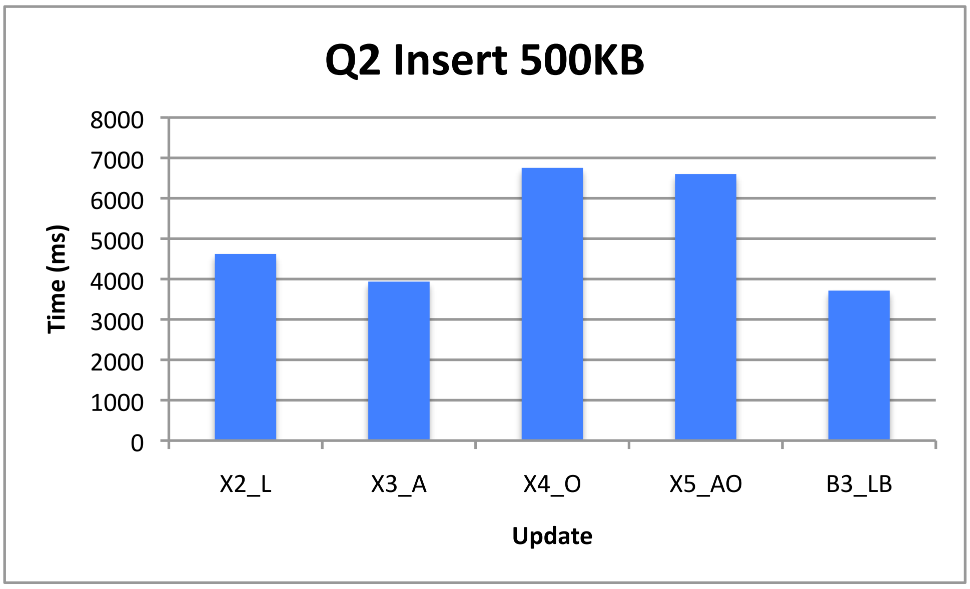
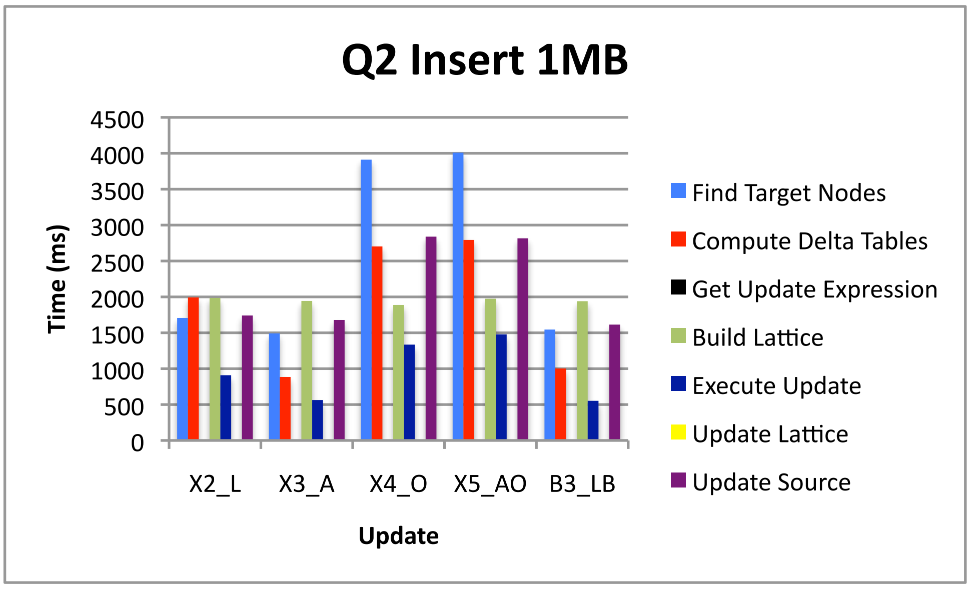


Figure C.6: Q2 Insert Graph 500KB

Query	TN	DT	UE	BL	EU	UL	US
X2_L	1706	1993	9	1985	909	0	1741
X3_A	1492	884	8	1943	563	0	1678
X4_O	3911	2703	10	1887	1335	0	2839
X5_AO	4009	2793	10	1974	1478	1	2816
B3_LB	1545	1003	10	1940	552	0	1614

Table C.7: Q2 Insert 1Mb



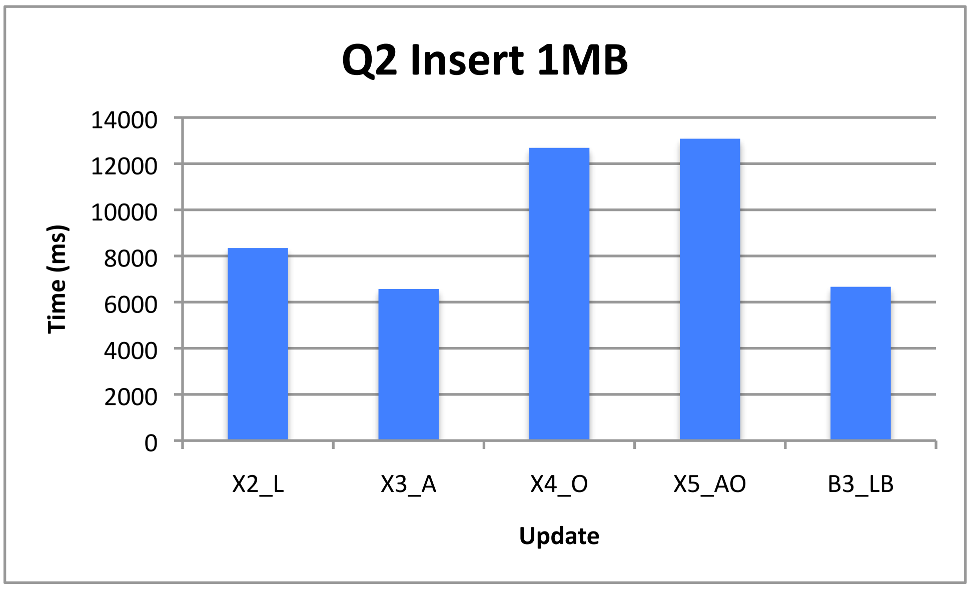


FIGURE C.7: Q2 Insert Graph 1MB

Query	TN	DT	UE	BL	EU	UL	US
X2_L	41096	16463	11	15969	12797	0	73958
X3_A	22918	7418	10	16126	5776	0	68689
X4_O	87674	23972	13	15546	30654	0	151364
X5_AO	85894	23378	15	16107	33810	0	156075
B3_LB	23211	8108	10	15980	6069	0	68845

TABLE C.8: Q2 Insert 10Mb

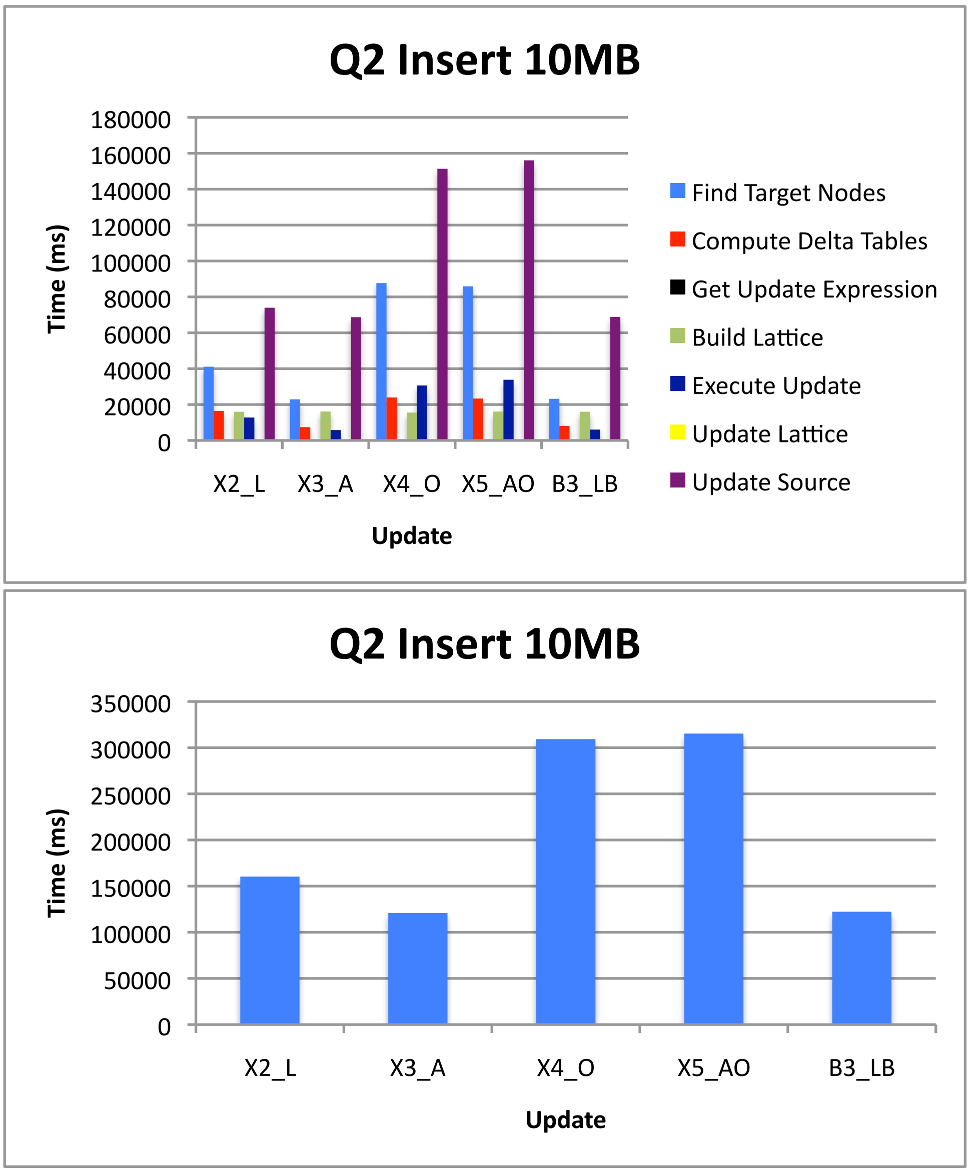


Figure C.8: Q2 Insert Graph 10MB

Query	TN	DT	UE	BL	EU	UL	US
X2_L	268	97	6	553	178	0	683
X3_A	267	66	6	561	167	0	667
X4_O	638	123	7	558	318	0	752
X5_AO	637	123	7	562	371	0	761
B3_LB	261	62	6	552	166	0	661

Table C.9: Q3 Insert 100Kb

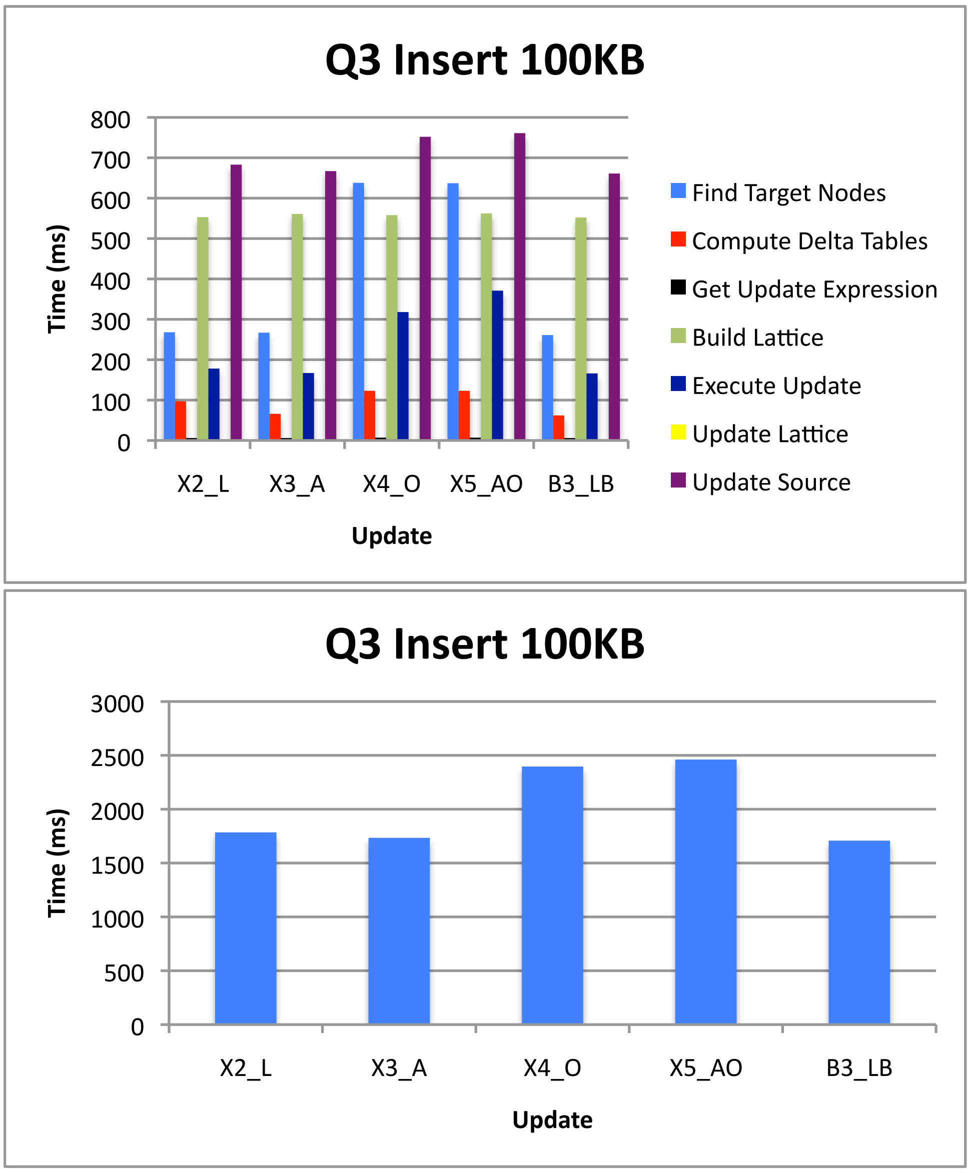
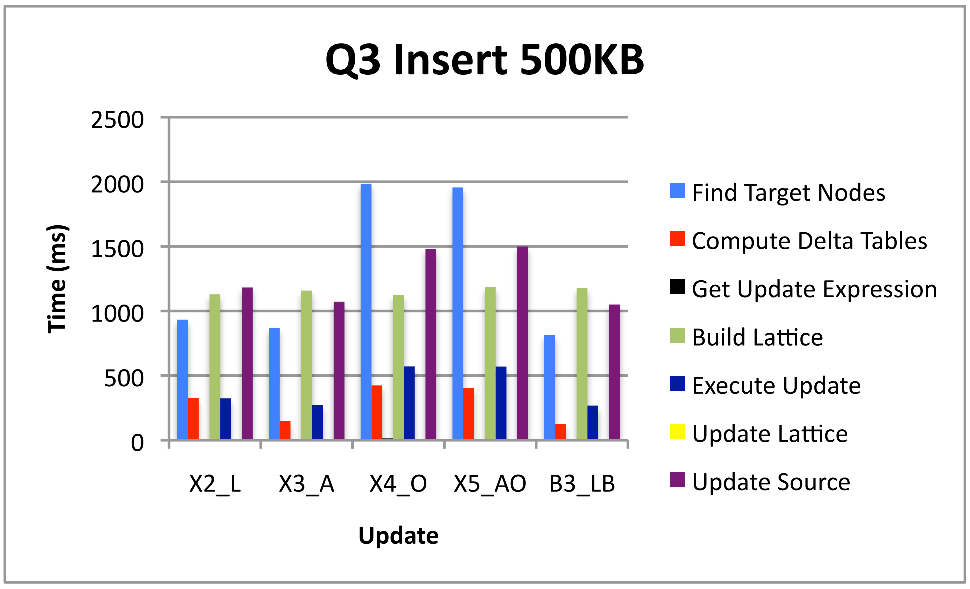


FIGURE C.9: Q3 Insert Graph 100KB

Query	TN	DT	UE	BL	EU	UL	US
X2_L	933	326	10	1129	324	0	1182
X3_A	869	149	6	1158	274	0	1072
X4_O	1985	424	14	1122	571	0	1481
X5_AO	1956	402	10	1186	570	0	1499
B3_LB	815	125	6	1177	268	0	1050

TABLE C.10: Q3 Insert 500Kb



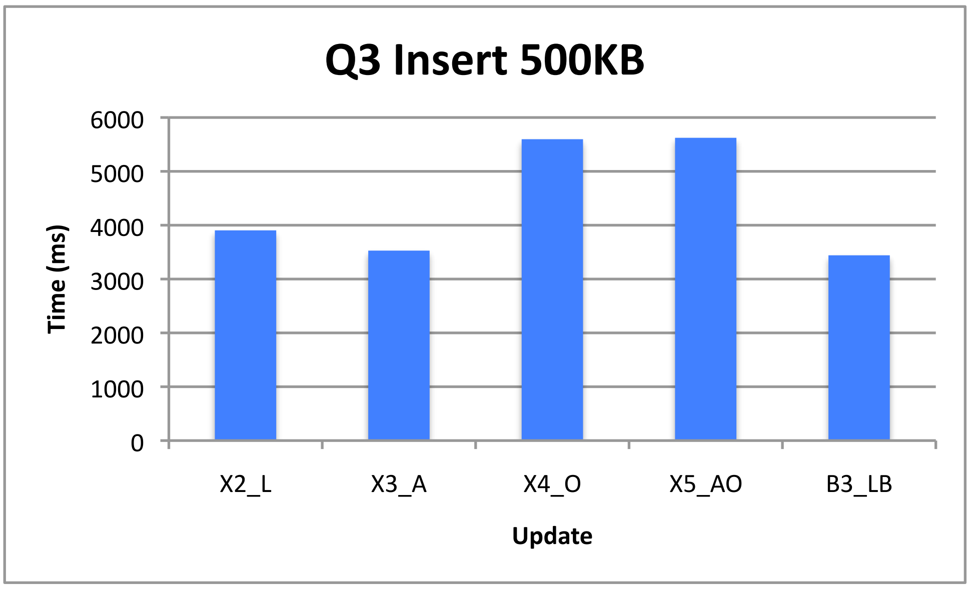


Figure C.10: Q3 Insert Graph 500KB

Query	TN	DT	UE	BL	EU	UL	US
X2_L	1737	672	9	1883	482	0	1740
X3_A	1525	338	9	1964	458	0	1521
X4_O	3855	861	10	1942	741	0	2493
X5_AO	4008	879	10	1946	724	1	2490
B3_LB	1537	361	10	1872	463	0	1544

Table C.11: Q3 Insert 1Mb

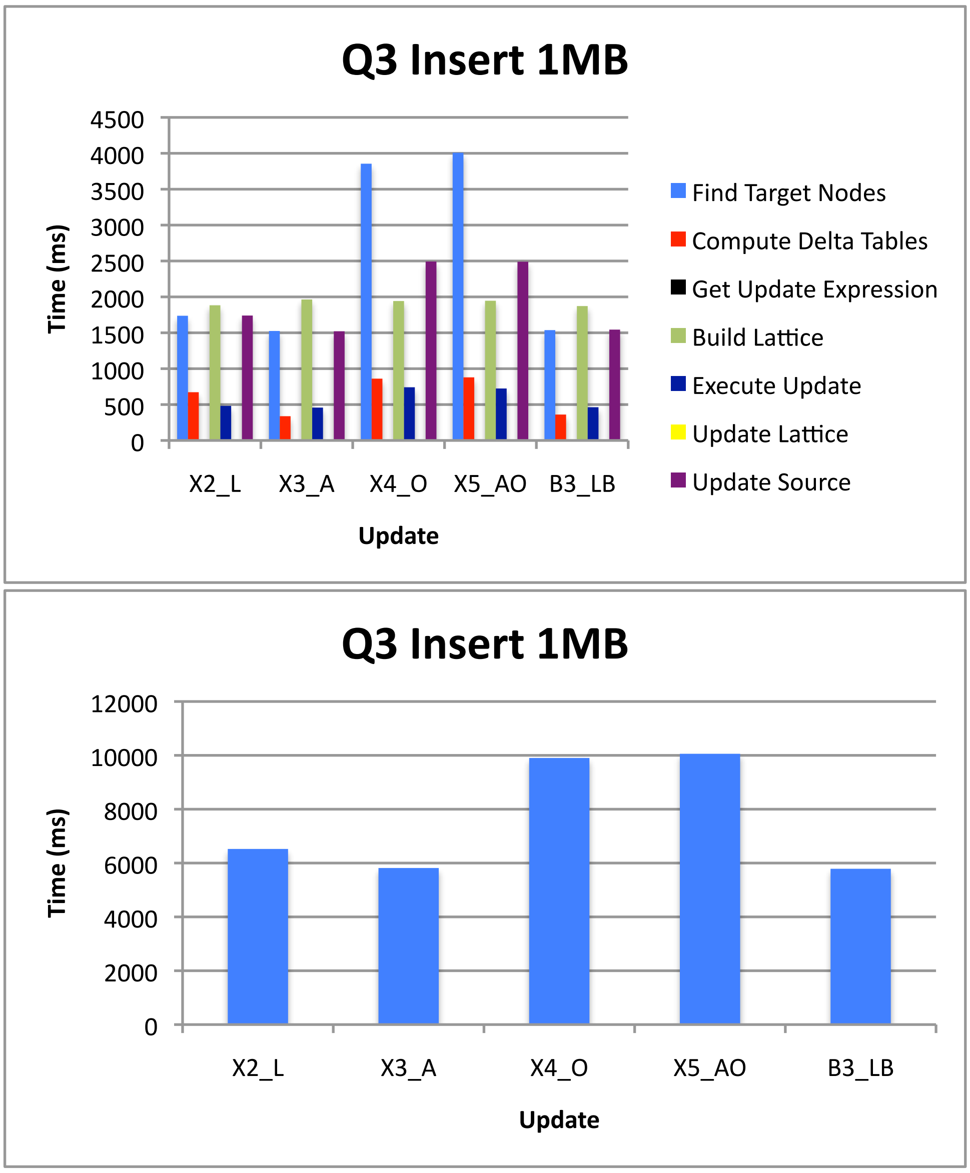
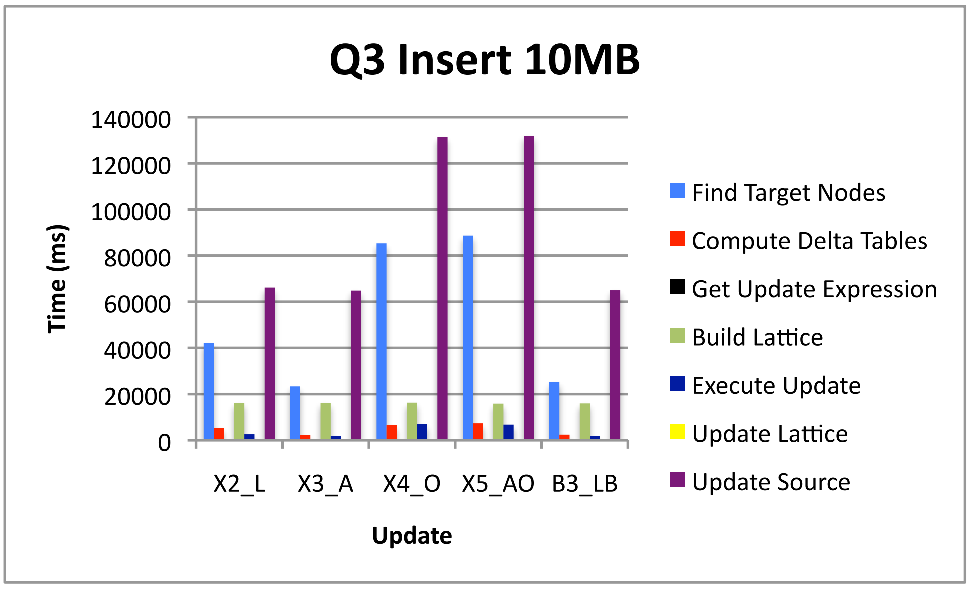


FIGURE C.11: Q3 Insert Graph 1MB

Query	TN	DT	UE	BL	EU	UL	US
X2_L	42158	5341	12	16185	2577	0	66155
X3_A	23332	2191	11	16174	1806	0	64843
X4_O	85310	6576	13	16284	6987	0	131294
X5_AO	88670	7331	14	15863	6761	0	131892
B3_LB	25284	2409	10	15958	1804	0	64983

TABLE C.12: Q3 Insert 10Mb



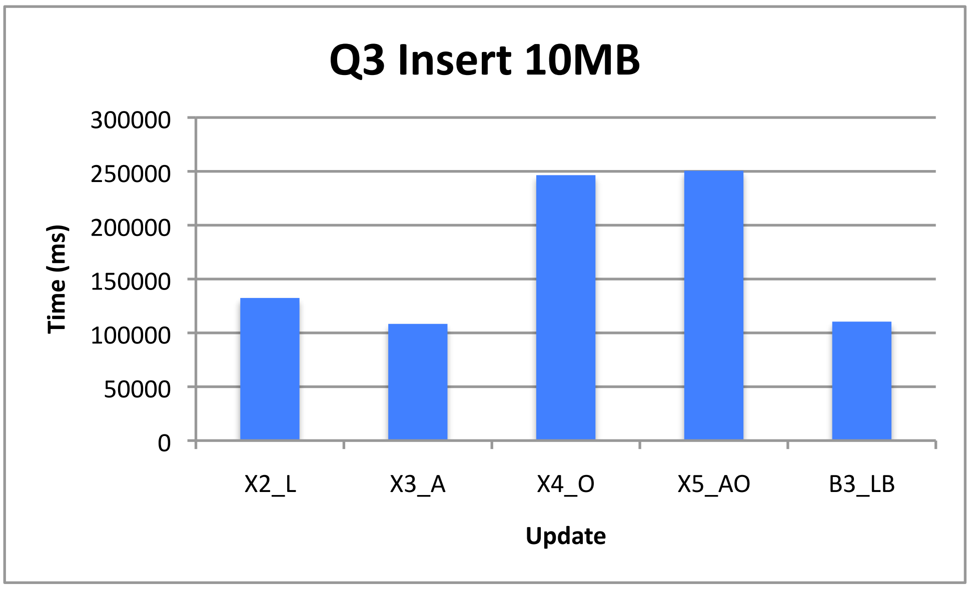
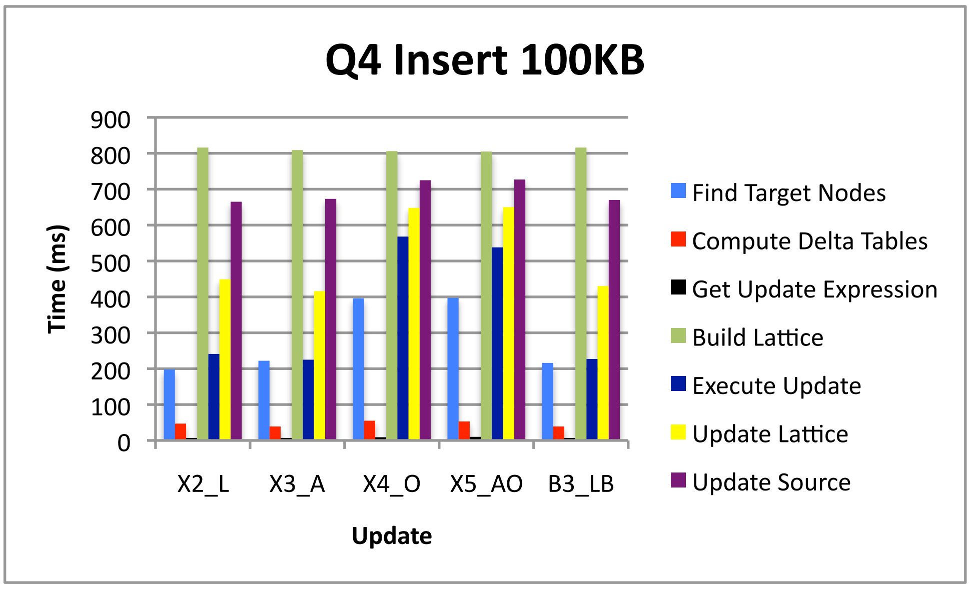


Figure C.12: Q3 Insert Graph 10MB

Query	TN	DT	UE	BL	EU	UL	US
X2_L	198	47	7	816	241	449	665
X3_A	222	39	7	809	225	416	673
X4_O	396	55	9	806	568	648	725
X5_AO	397	53	10	805	538	650	727
B3_LB	216	39	7	816	227	430	670

Table C.13: Q4 Insert 100Kb



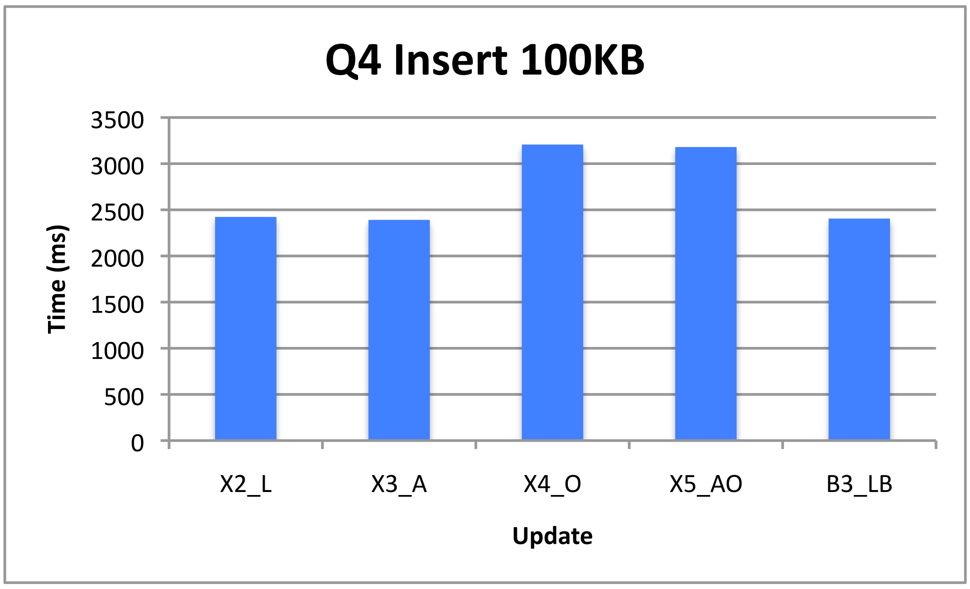
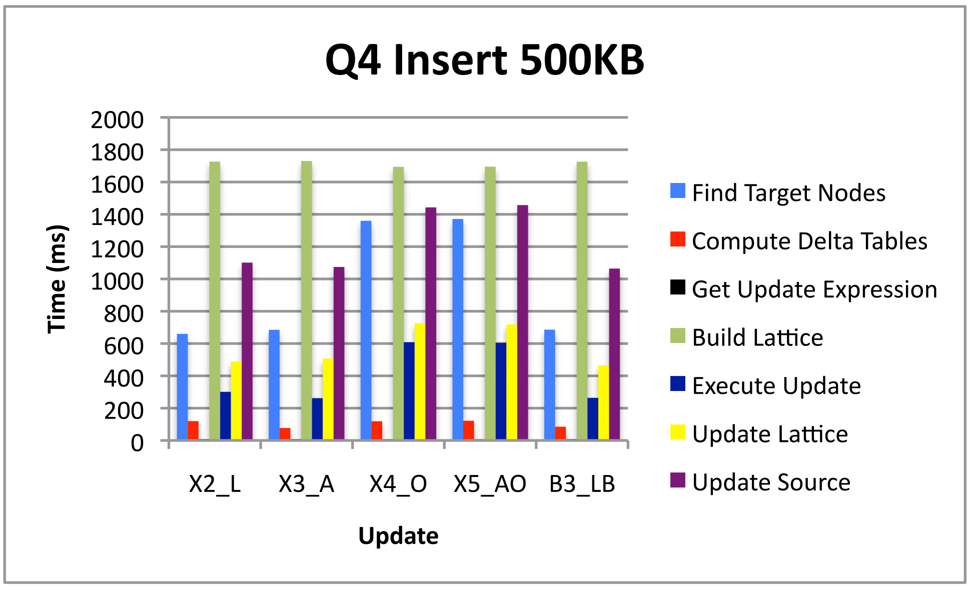


Figure C.13: Q4 Insert Graph 100KB

Query	TN	DT	UE	BL	EU	UL	US
X2_L	660	120	9	1726	301	488	1101
X3_A	685	77	9	1730	262	507	1074
X4_O	1360	119	8	1694	608	725	1443
X5_AO	1371	122	8	1695	604	718	1457
B3_LB	686	85	9	1726	264	464	1064

Table C.14: Q4 Insert 500Kb



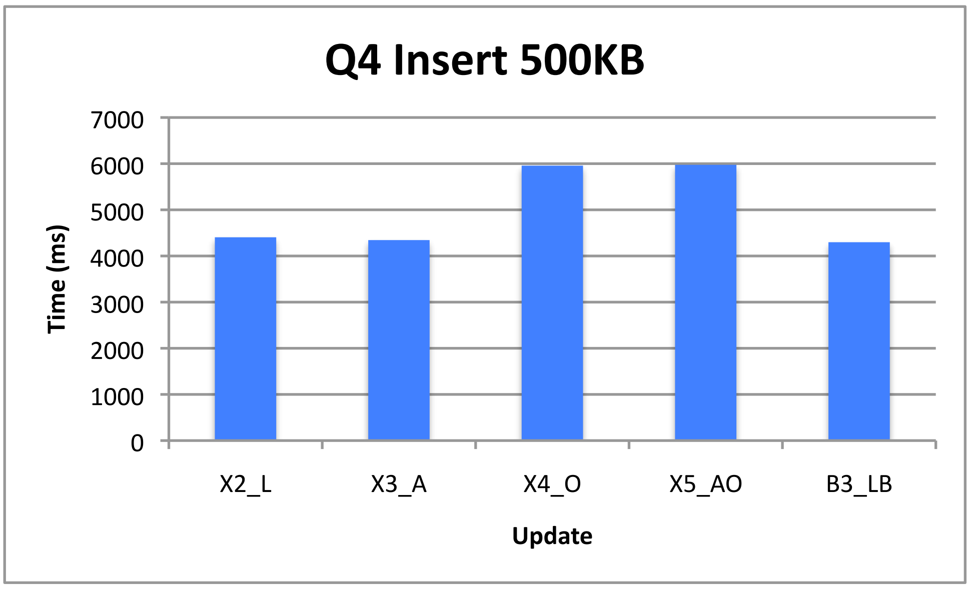
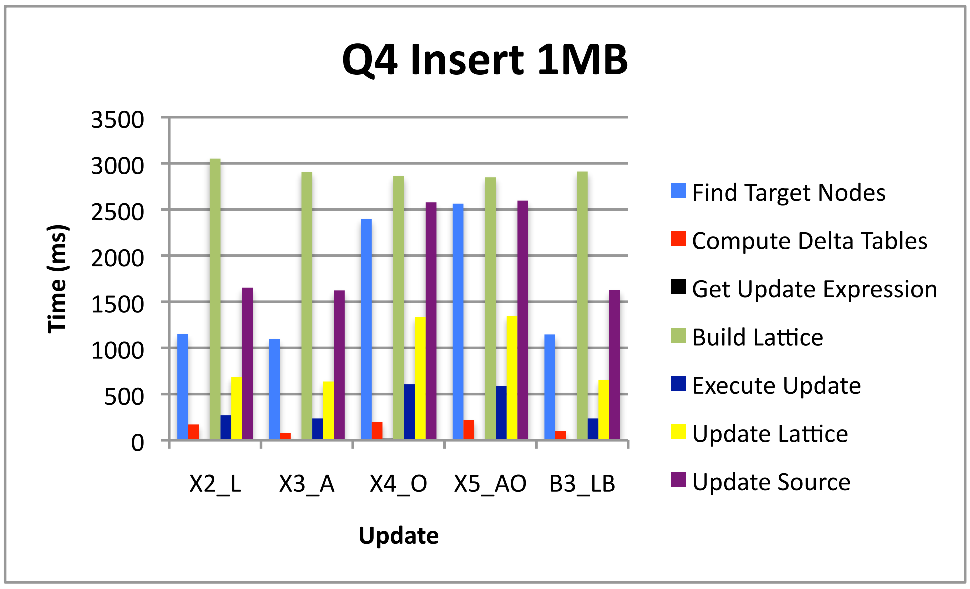


Figure C.14: Q4 Insert Graph 500KB

Query	TN	DT	UE	BL	EU	UL	US
X2_L	1149	171	16	3051	270	684	1653
X3_A	1098	78	7	2907	236	636	1623
X4_O	2397	200	18	2861	606	1335	2577
X5_AO	2563	219	12	2848	589	1344	2596
B3_LB	1146	101	7	2911	236	651	1630

Table C.15: Q4 Insert 1Mb



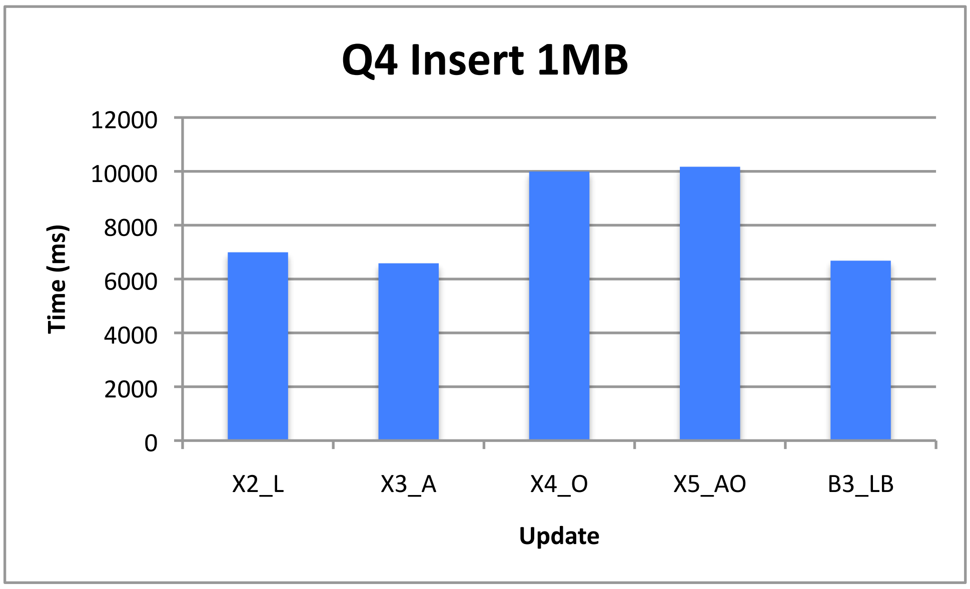


FIGURE C.15: Q4 Insert Graph 1MB

Query	TN	DT	UE	BL	EU	UL	US
X2_L	100413	1301	21	22343	48141	41412	49651
X3_A	106368	1335	20	22099	49558	42508	54950
X4_O	100565	1372	20	23512	47977	42489	49548
X5_AO	106817	1291	21	22431	49582	42057	49632
B3_LB	107402	1292	21	21950	49314	42944	49578

TABLE C.16: Q4 Insert 10Mb

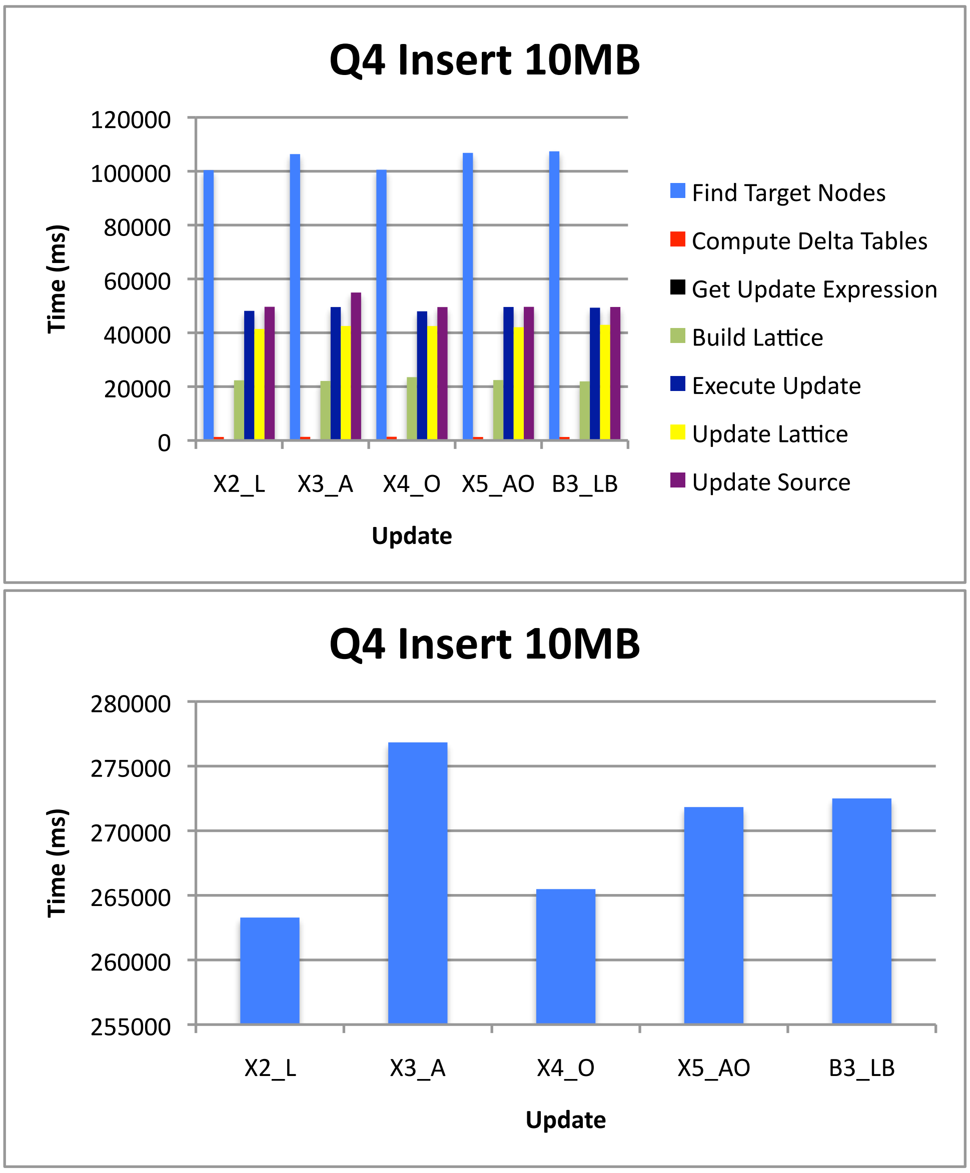
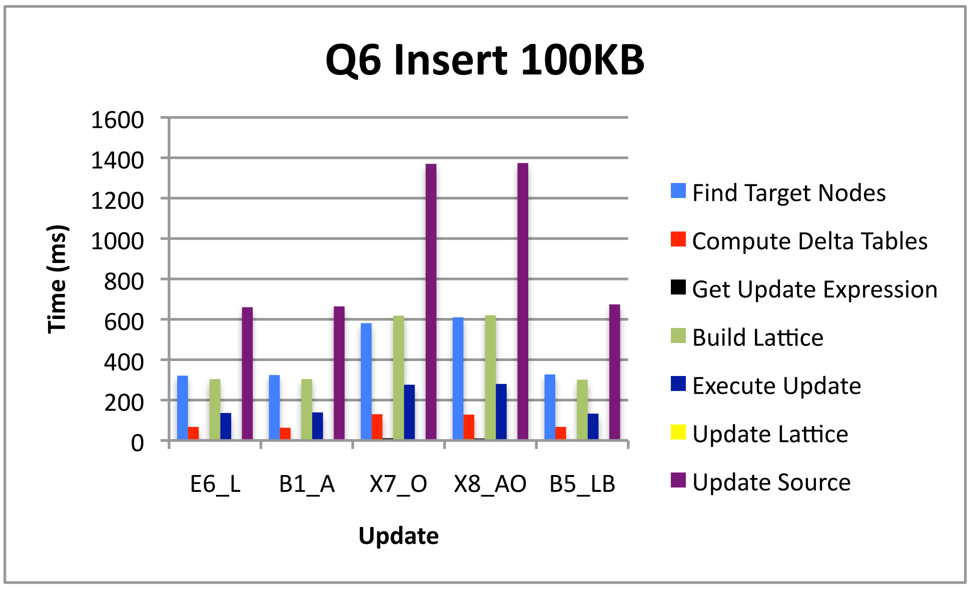


Figure C.16: Q4 Insert Graph 10MB

Query	TN	DT	UE	BL	EU	UL	US
E6_L	321	67	5	304	136	0	660
B1_A	324	63	5	304	139	0	664
X7_O	581	130	11	618	276	0	1370
X8_AO	610	128	10	620	280	0	1374
B5_LB	327	67	5	301	133	0	674

Table C.17: Q6 Insert 100Kb



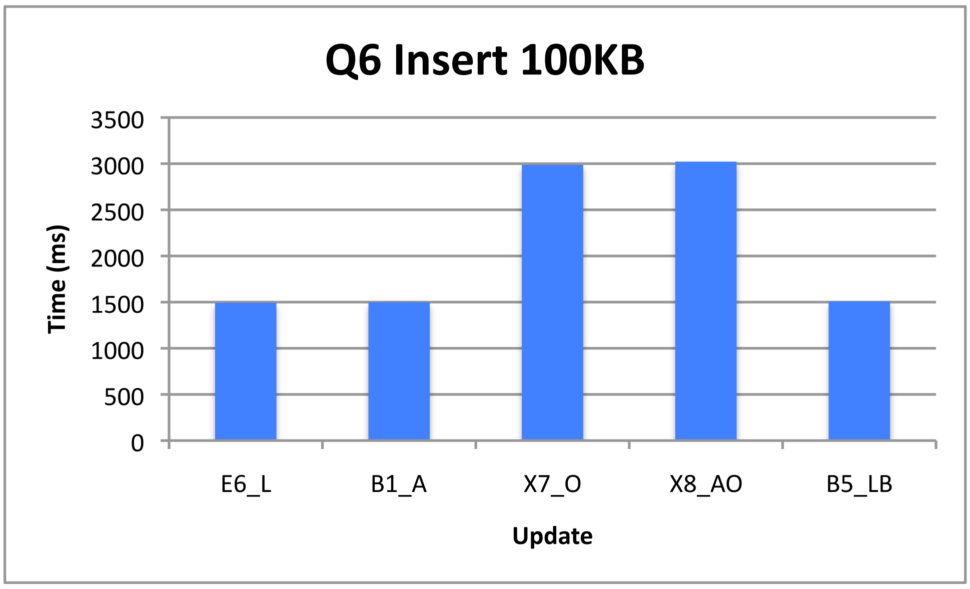
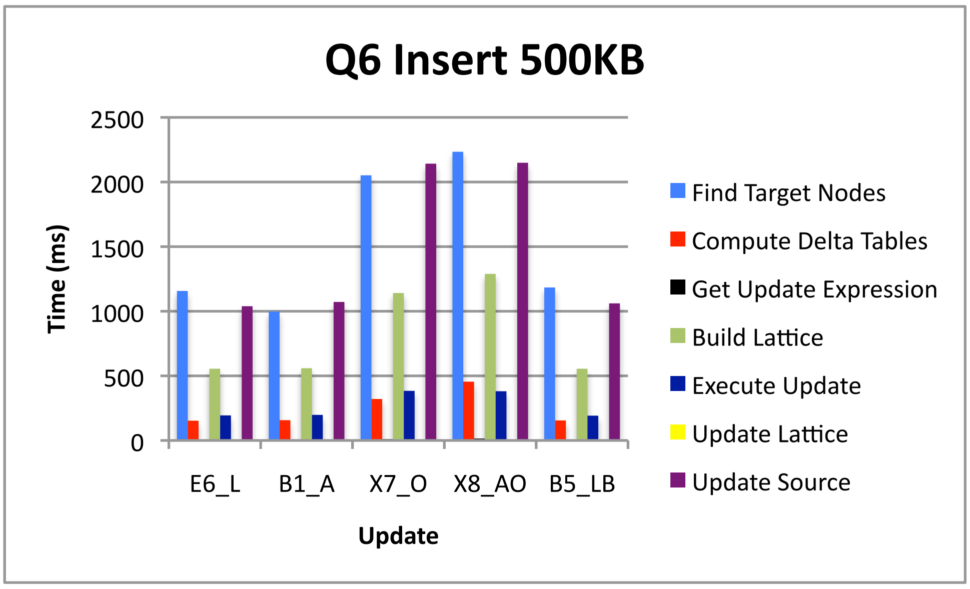


FIGURE C.17: Q6 Insert Graph 100KB

Query	TN	DT	UE	BL	EU	UL	US
E6_L	1157	153	6	555	194	0	1039
B1_A	998	157	5	559	198	0	1072
X7_O	2052	321	11	1141	384	0	2142
X8_AO	2234	455	15	1289	381	1	2149
B5_LB	1184	155	6	555	192	0	1061

TABLE C.18: Q6 Insert 500Kb



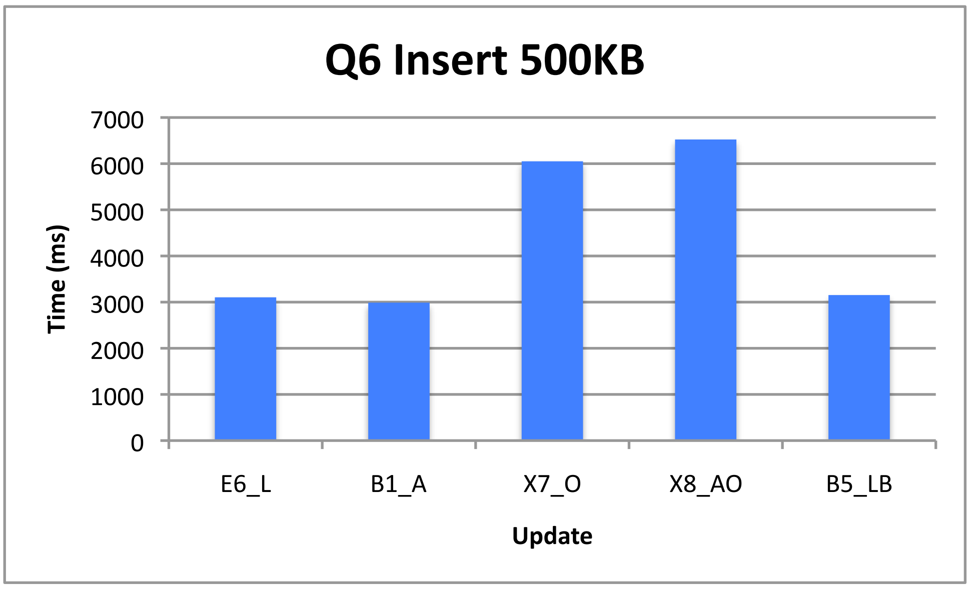
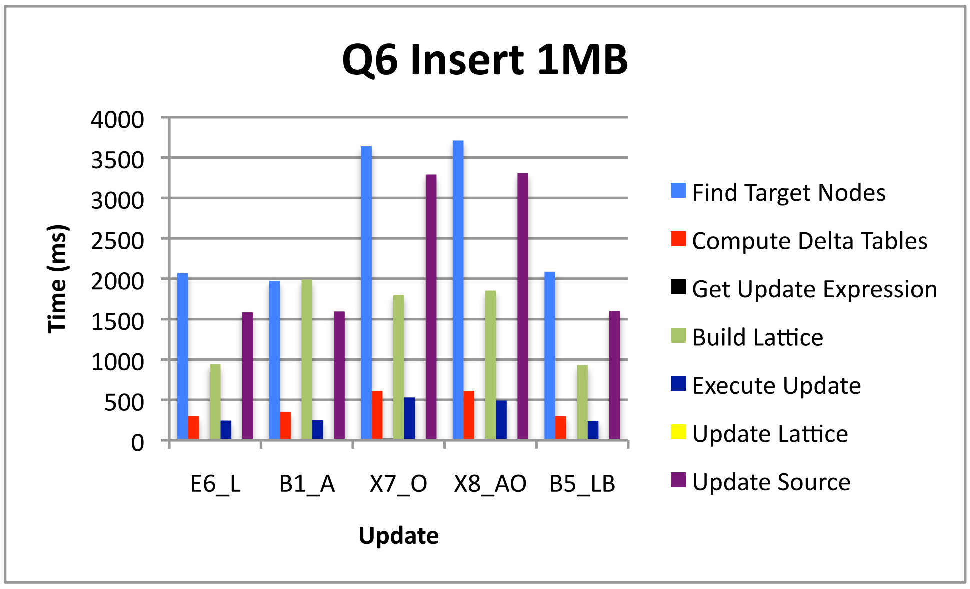


Figure C.18: Q6 Insert Graph 500KB

Query	TN	DT	UE	BL	EU	UL	US
E6_L	2069	302	8	944	244	0	1584
B1_A	1973	353	11	1991	247	0	1595
X7_O	3640	611	21	1800	530	0	3290
X8_AO	3711	612	17	1853	492	0	3307
B5_LB	2087	299	10	931	241	0	1599

Table C.19: Q6 Insert 1Mb



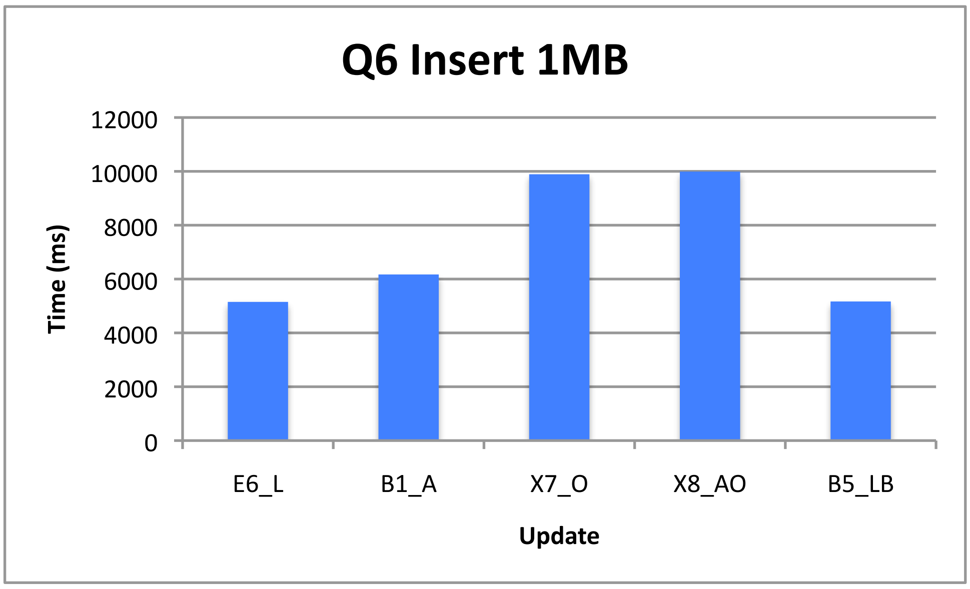


FIGURE C.19: Q6 Insert Graph 1MB

Query	TN	DT	UE	BL	EU	UL	US
E6_L	33374	1947	10	8525	1097	0	65813
B1_A	30235	2010	9	8392	1014	0	66032
X7_O	66709	4272	19	17612	1778	0	141274
X8_AO	64022	3989	18	17222	1678	1	141283
B5_LB	42555	1963	9	8390	1077	0	66143

TABLE C.20: Q6 Insert 10Mb

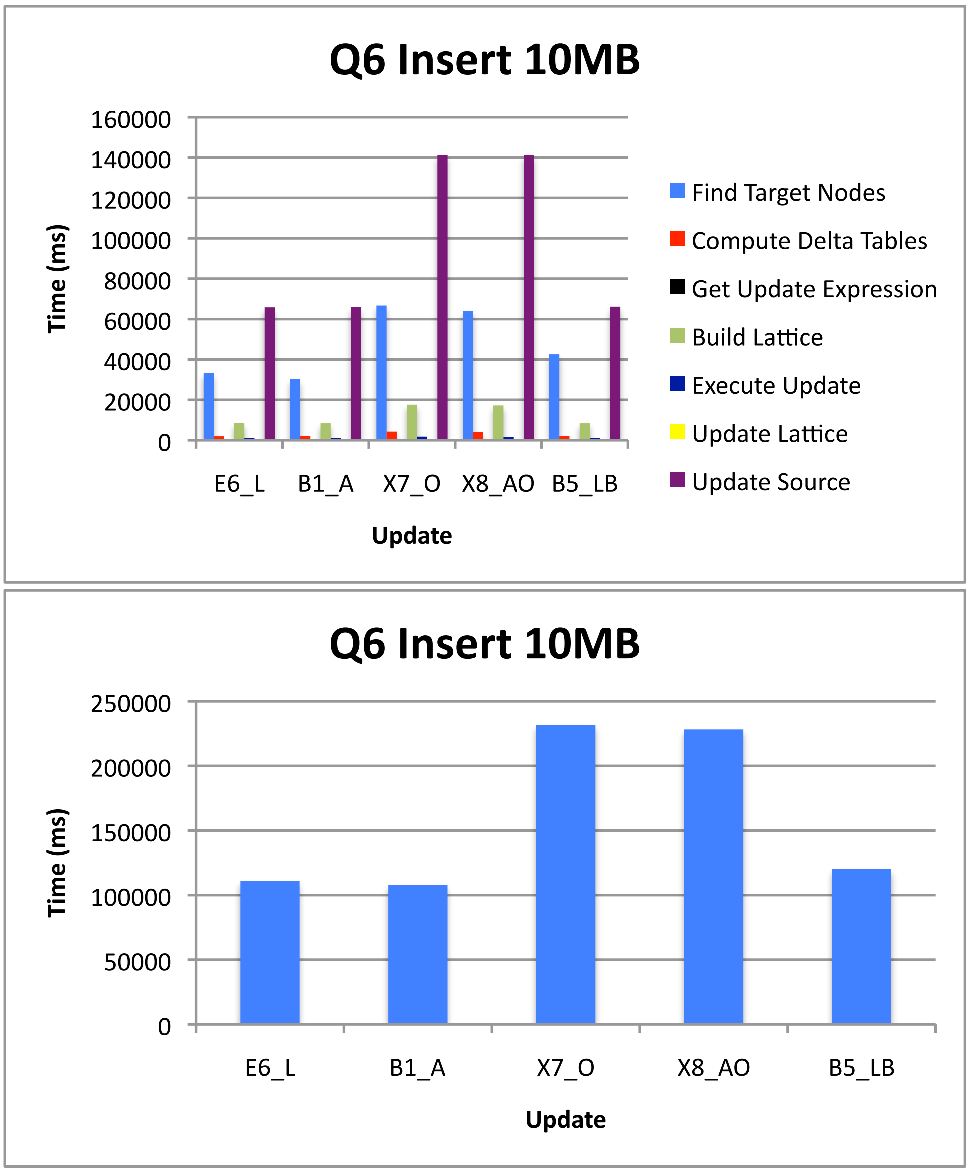
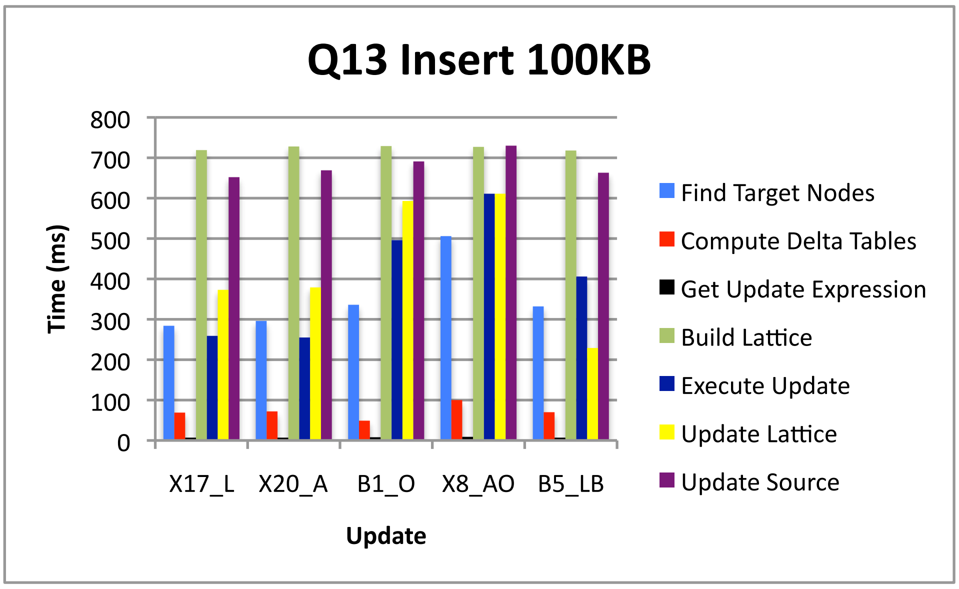


Figure C.20: Q6 Insert Graph 10MB

Query	TN	DT	UE	BL	EU	UL	US
X17_L	284	69	7	719	259	373	652
X20_A	296	72	7	728	255	379	669
B1_O	336	49	8	729	496	593	691
X8_AO	506	100	9	727	611	611	730
B5_LB	332	70	7	718	406	229	663

Table C.21: Q13 Insert 100Kb



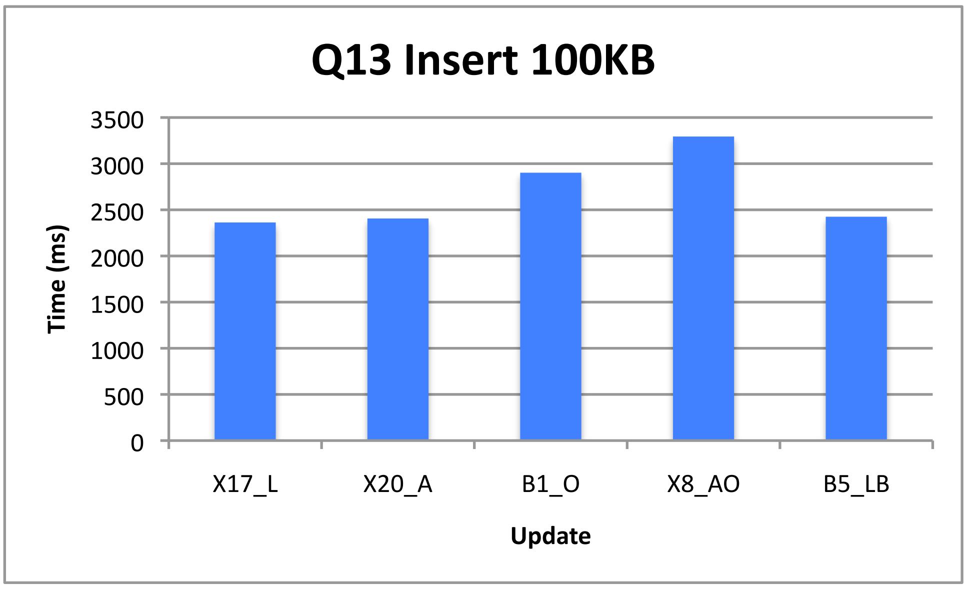
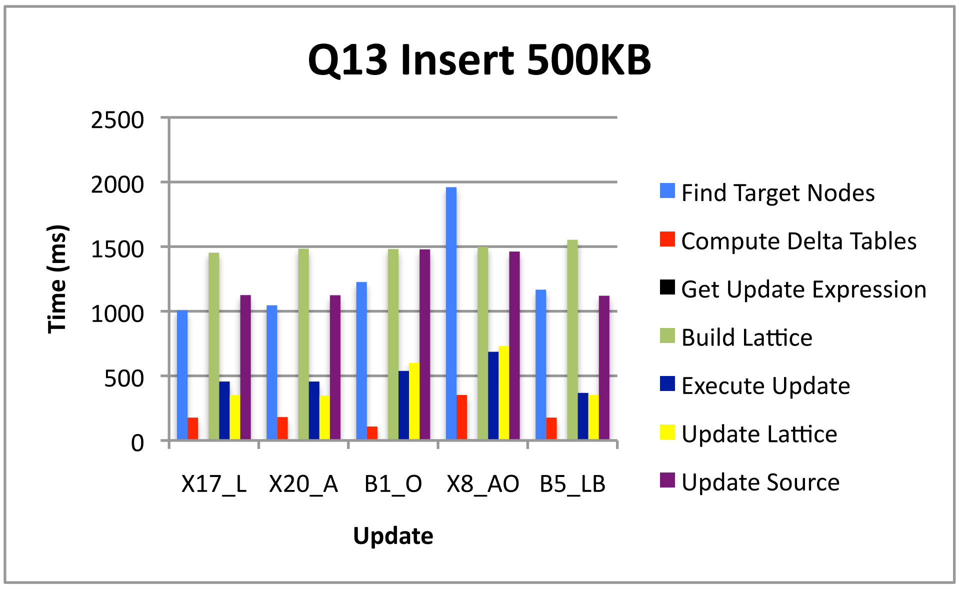


FIGURE C.21: Q13 Insert Graph 100KB

Query	TN	DT	UE	BL	EU	UL	US
X17_L	1007	177	7	1453	456	351	1125
X20_A	1046	181	7	1484	456	347	1124
B1_O	1226	108	7	1481	538	600	1479
X8_AO	1960	352	9	1498	686	730	1462
B5_LB	1167	177	7	1553	368	353	1120

TABLE C.22: Q13 Insert 500Kb



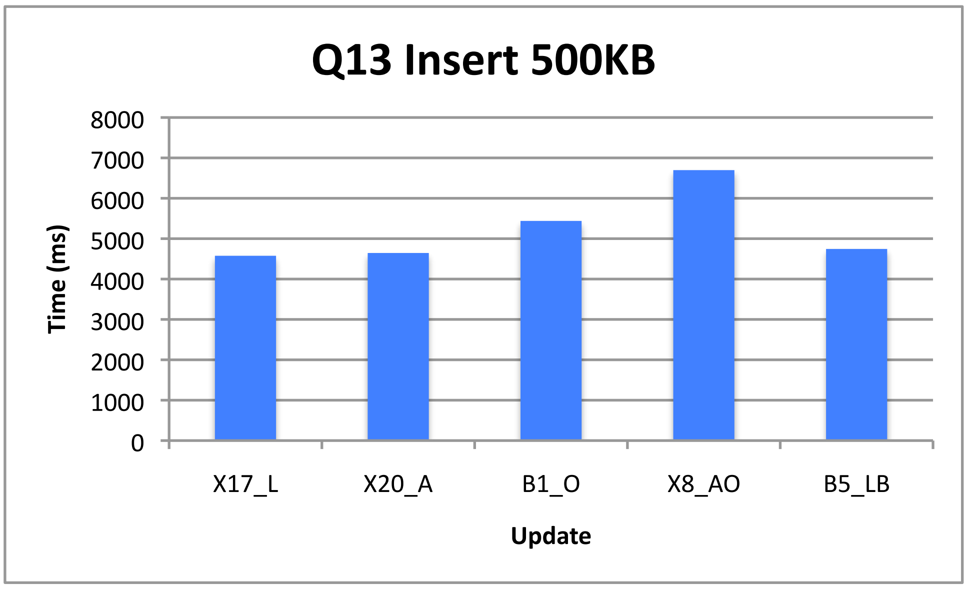
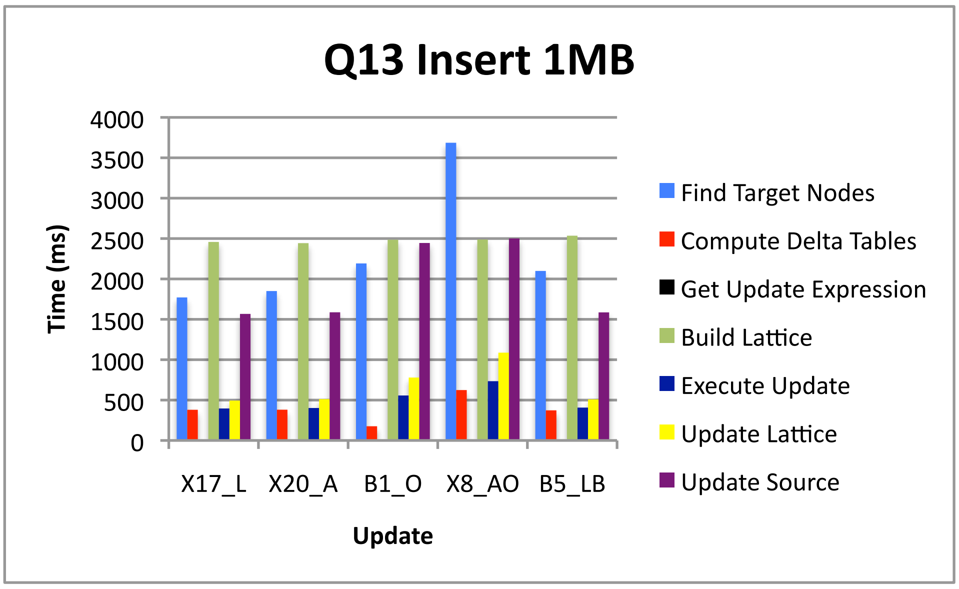


Figure C.22: Q13 Insert Graph 500KB

Query	TN	DT	UE	BL	EU	UL	US
X17_L	1771	380	10	2458	396	494	1567
X20_A	1850	381	10	2443	402	512	1587
B1_O	2192	176	8	2485	557	779	2445
X8_AO	3686	624	12	2488	734	1087	2501
B5_LB	2099	373	10	2535	407	509	1586

Table C.23: Q13 Insert 1Mb



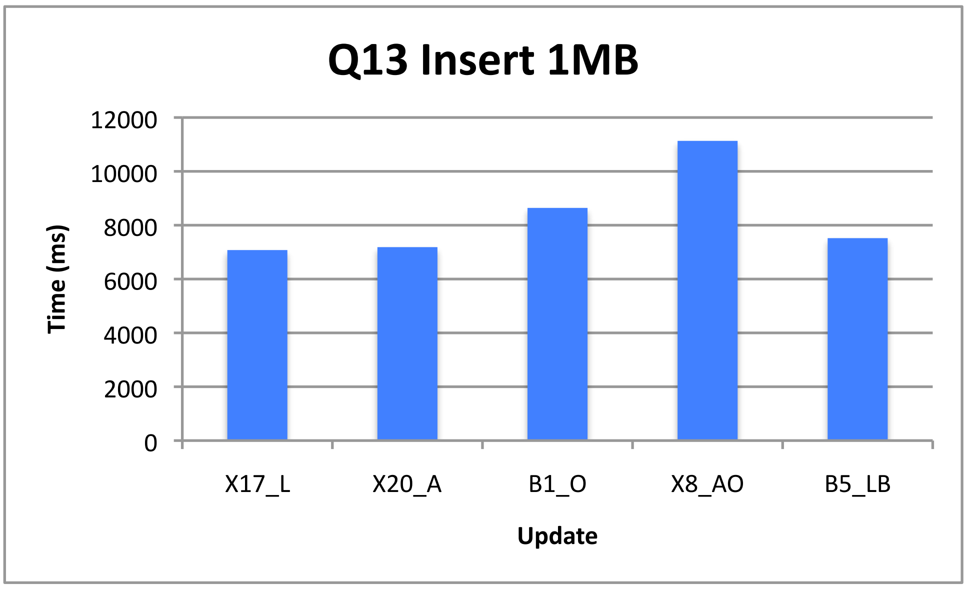
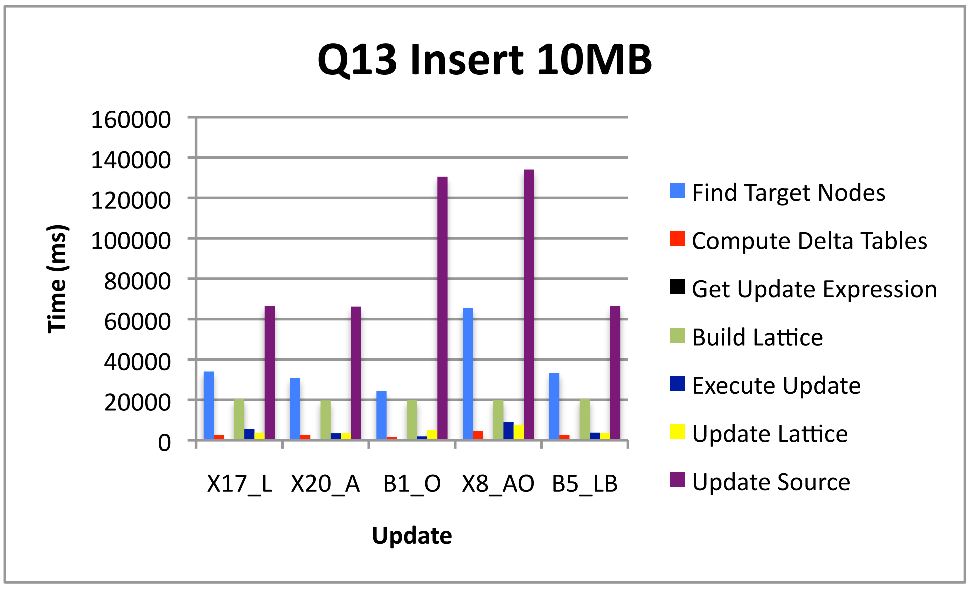


FIGURE C.23: Q13 Insert Graph 1MB

Query	TN	DT	UE	BL	EU	UL	US
X17_L	34034	2694	12	20306	5575	3499	66372
X20_A	30763	2541	13	19729	3436	3407	66182
B1_O	24310	1471	11	19821	1899	5079	130518
X8_AO	65430	4542	17	19993	8897	7521	134060
B5_LB	33256	2576	13	20291	3751	3486	66349

TABLE C.24: Q13 Insert 10Mb



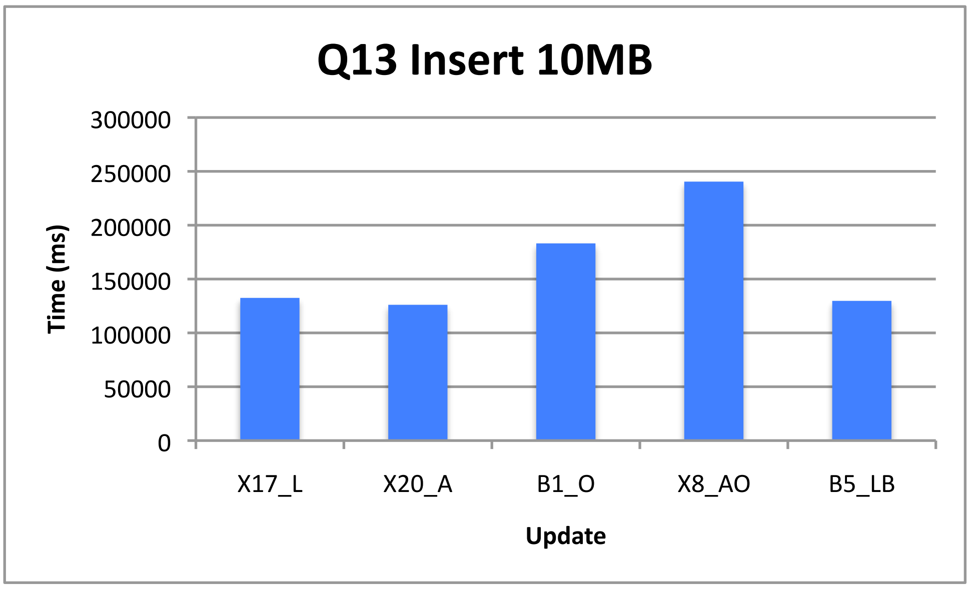


Figure C.24: Q13 Insert Graph 10MB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	231	121	8	638	165	108	760
X6_A	194	60	5	638	139	77	648
A7_O	345	126	6	660	325	165	821
A8_AO	690	99	11	979	510	273	1297
B7_LB	175	71	5	649	145	79	655

Table C.25: Q17 Insert 100Kb

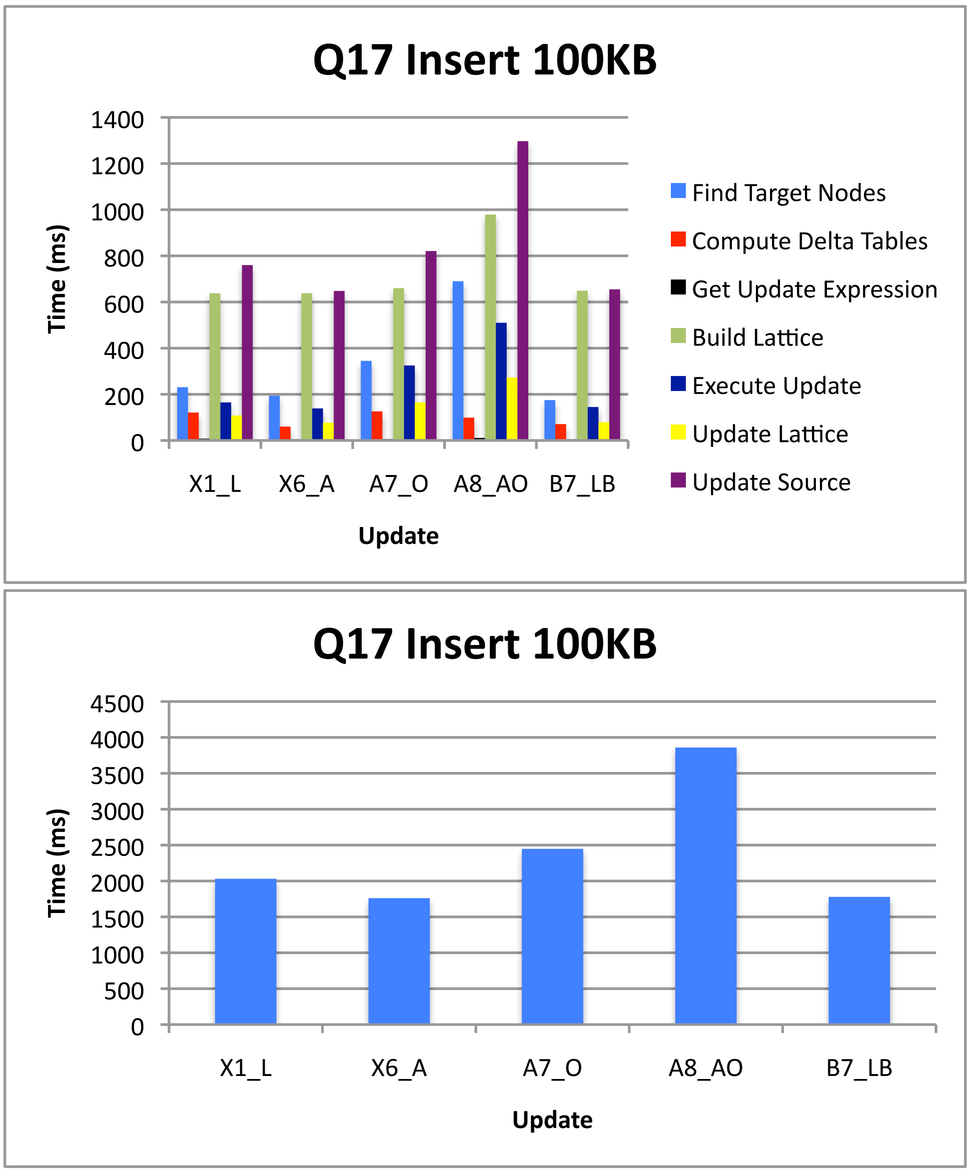


FIGURE C.25: Q17 Insert Graph 100KB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	875	431	6	1288	322	193	1071
X6_A	625	103	6	1368	161	118	1017
A7_O	1221	366	10	1316	366	261	1363
A8_AO	1868	204	8	1360	529	335	1939
B7_LB	625	264	8	1313	231	181	1059

TABLE C.26: Q17 Insert 500Kb

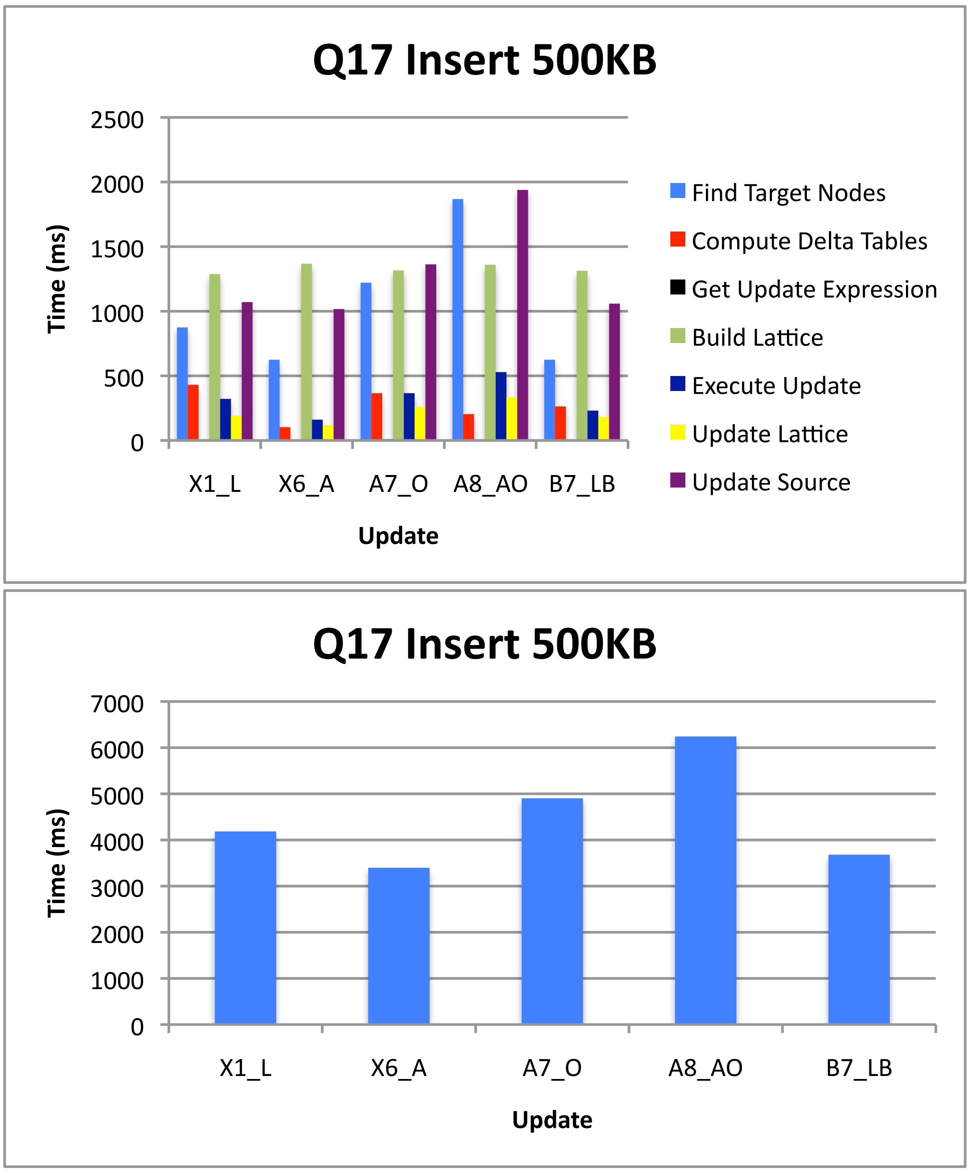
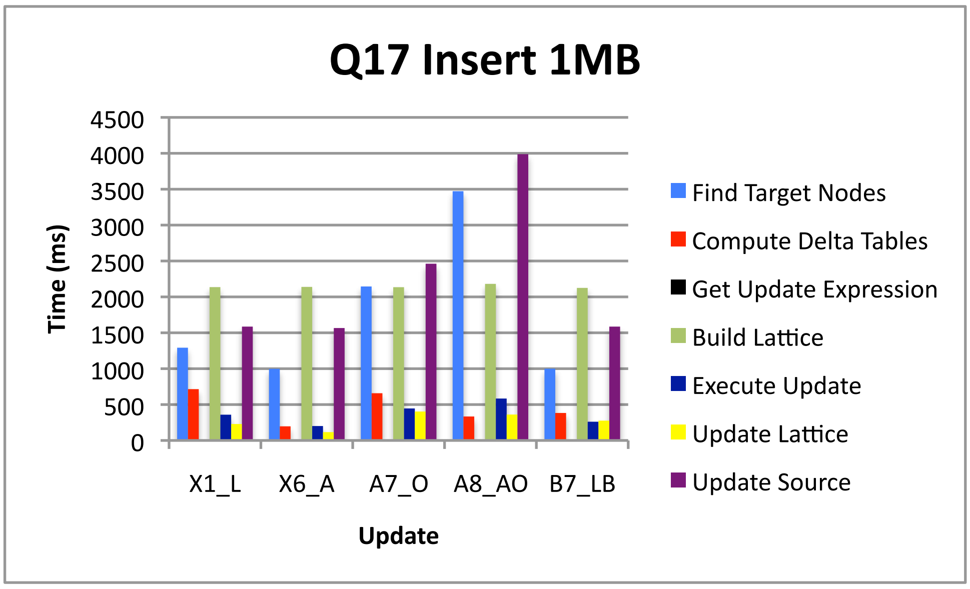


Figure C.26: Q17 Insert Graph 500KB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	1292	715	9	2136	359	229	1586
X6_A	998	197	6	2139	201	116	1566
A7_O	2145	658	6	2135	446	403	2462
A8_AO	3472	334	9	2181	584	361	3988
B7_LB	1000	383	5	2125	261	276	1586

Table C.27: Q17 Insert 1Mb



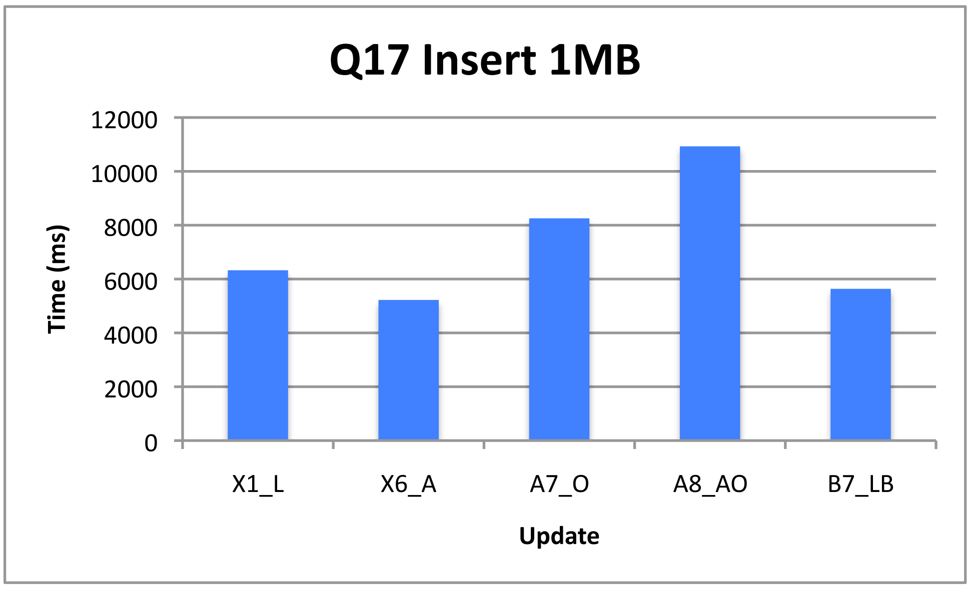


FIGURE C.27: Q17 Insert Graph 1MB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	22616	5740	11	16364	3607	2372	69329
X6_A	10352	1514	9	16542	947	829	65298
A7_O	26928	5639	10	16622	3874	2605	134678
A8_AO	40571	2928	12	18035	2732	2617	261230
B7_LB	10769	2973	10	16575	1663	1345	66468

TABLE C.28: Q17 Insert 10Mb

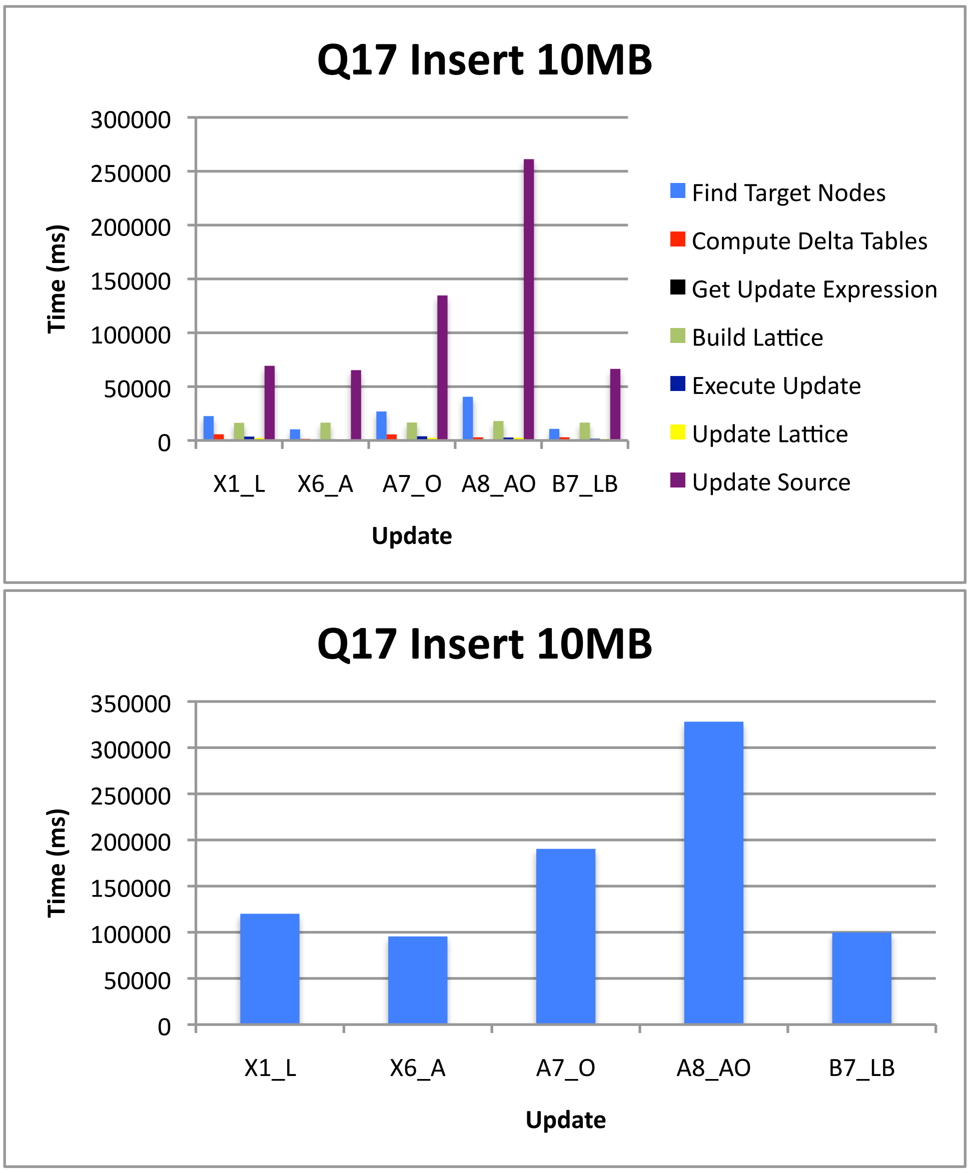
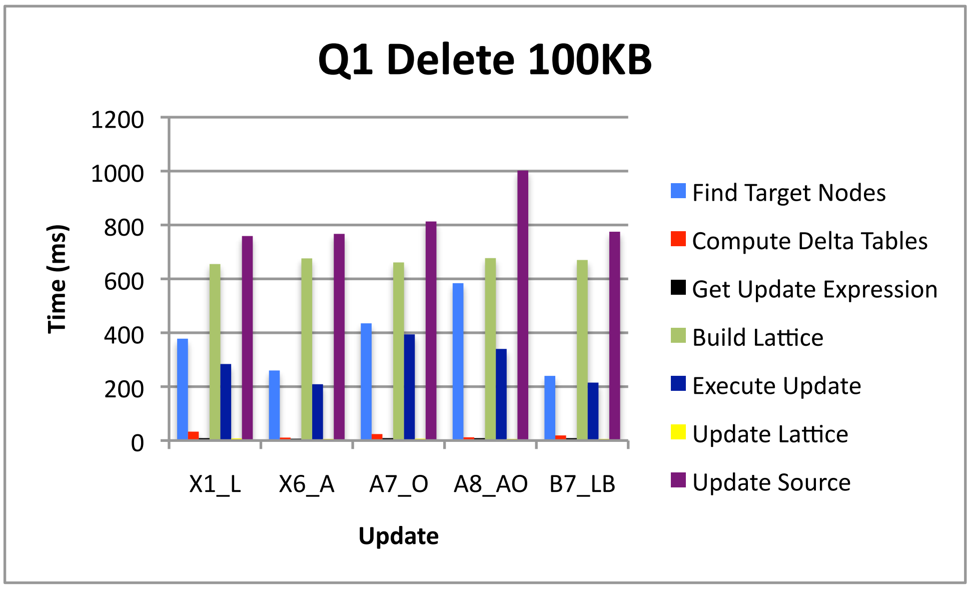


Figure C.28: Q17 Insert Graph 10MB

C.1.2 Delete

Query	TN	DT	UE	BL	EU	UL	US
X1_L	378	33	9	655	284	8	759
X6_A	260	11	7	676	209	6	767
A7_O	435	24	9	661	394	7	813
A8_AO	584	12	9	677	340	6	1002
B7_LB	240	19	9	670	215	6	775

TABLE C.29: Q1 Delete 100Kb



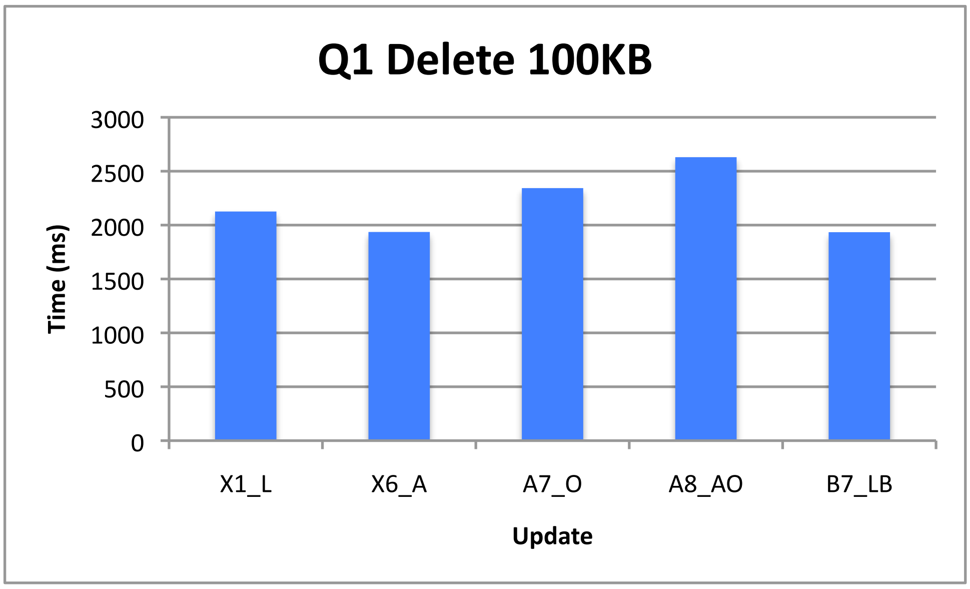
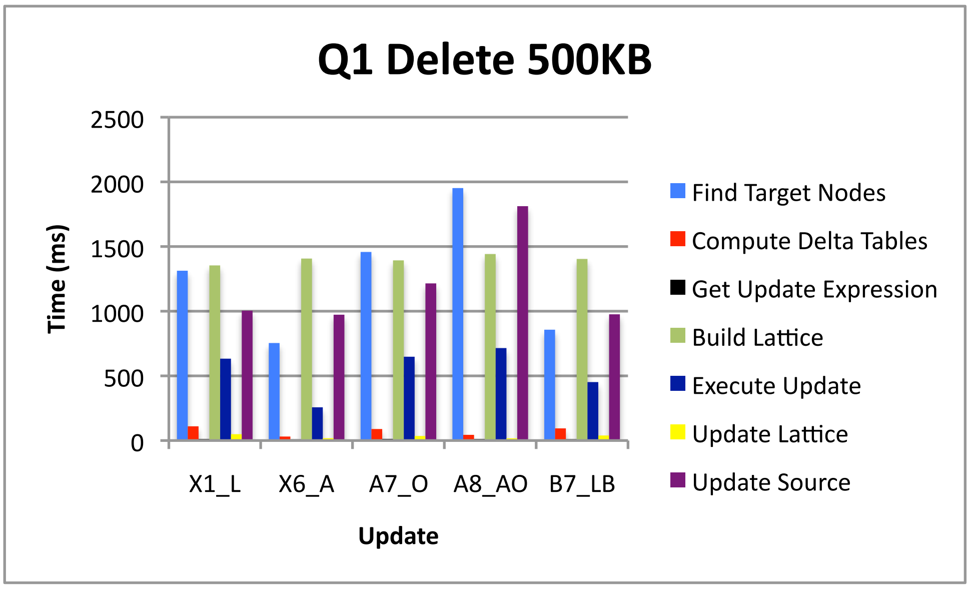


Figure C.29: Q1 Delete Graph 100KB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	1313	110	12	1354	633	49	1004
X6_A	754	31	10	1407	257	18	973
A7_O	1458	89	13	1393	648	35	1215
A8_AO	1952	44	12	1442	715	17	1812
B7_LB	857	94	11	1404	452	39	976

Table C.30: Q1 Delete 500Kb



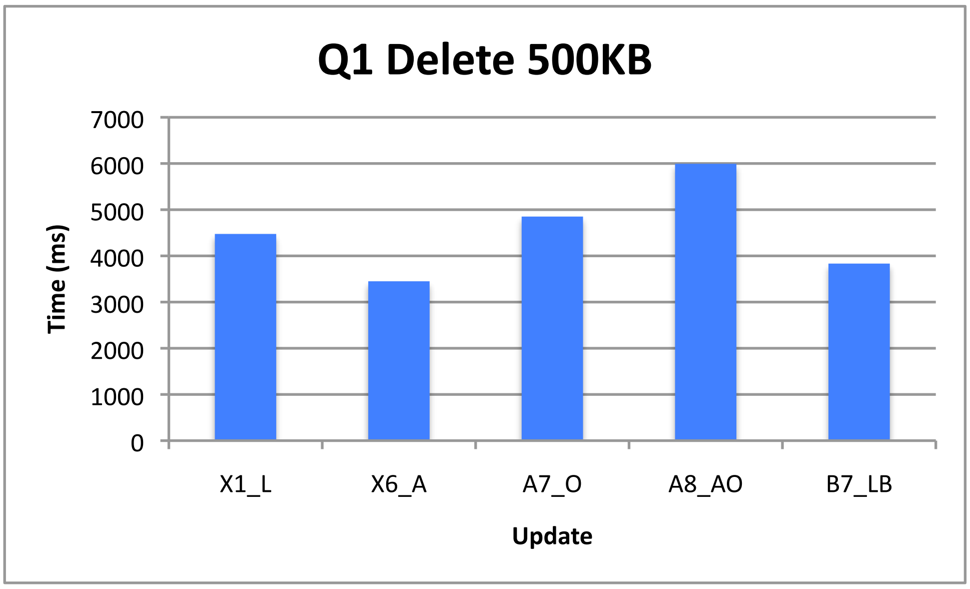
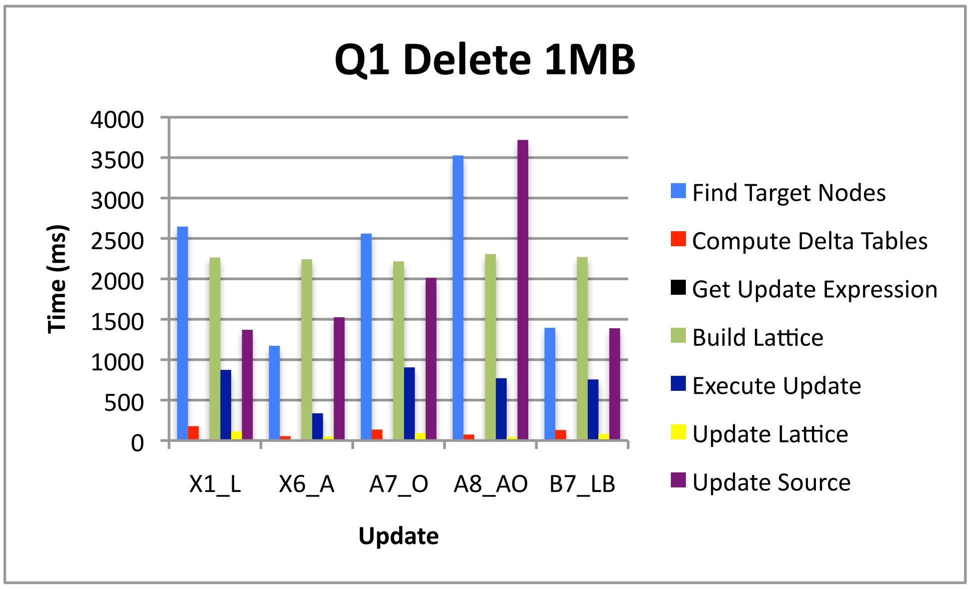


FIGURE C.30: Q1 Delete Graph 500KB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	2647	178	14	2265	874	114	1370
X6_A	1172	54	9	2243	337	50	1525
A7_O	2560	136	14	2217	905	92	2012
A8_AO	3527	74	14	2307	771	46	3719
B7_LB	1395	130	12	2271	756	82	1389

TABLE C.31: Q1 Delete 1Mb



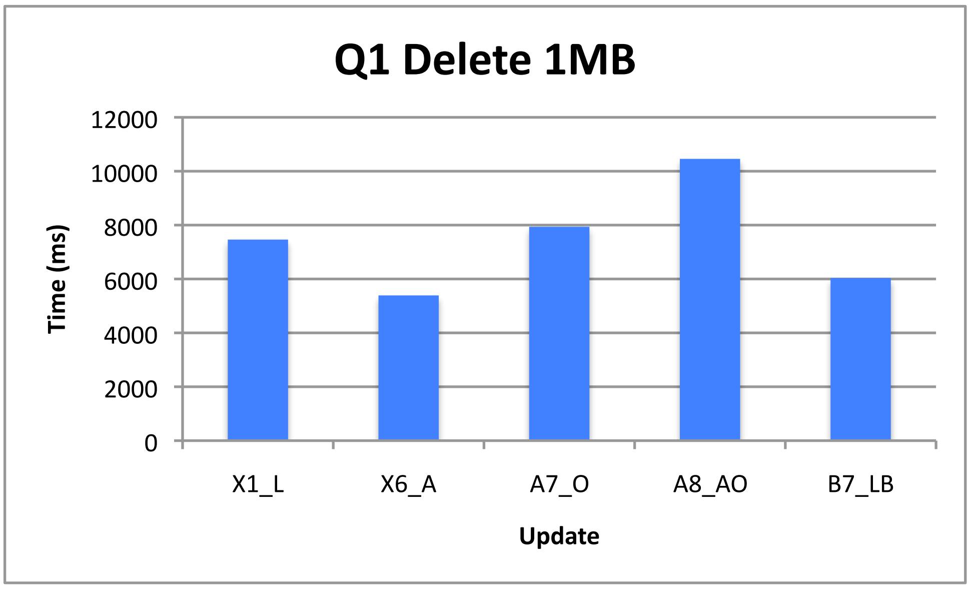
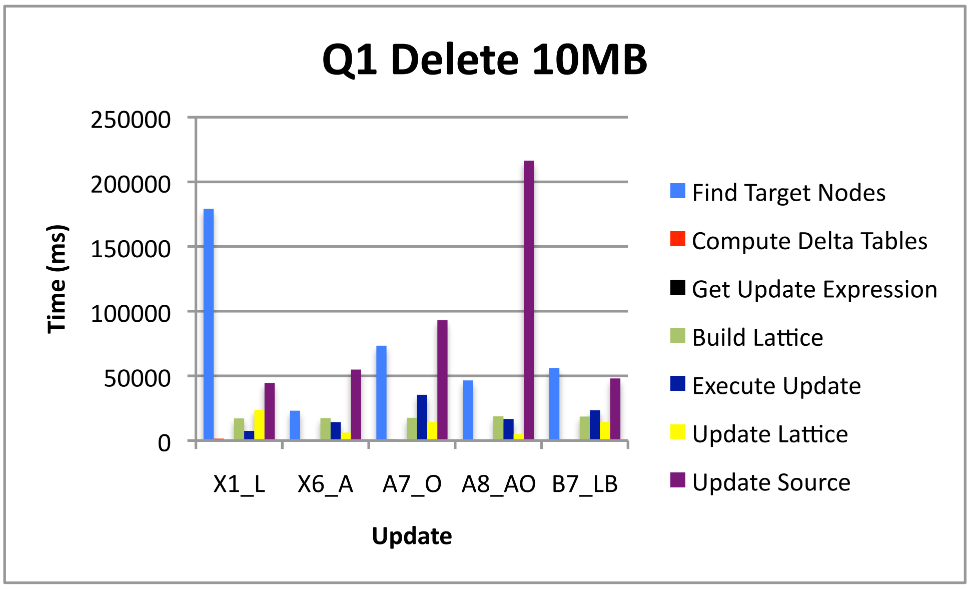


Figure C.31: Q1 Delete Graph 1MB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	179125	1486	40	17109	7472	23622	44573
X6_A	23086	398	17	17360	14236	6148	54872
A7_O	73319	1053	30	17593	35379	14160	93070
A8_AO	46492	580	21	18739	16678	4831	216428
B7_LB	56135	931	23	18553	23433	14273	48020

Table C.32: Q1 Delete 10Mb



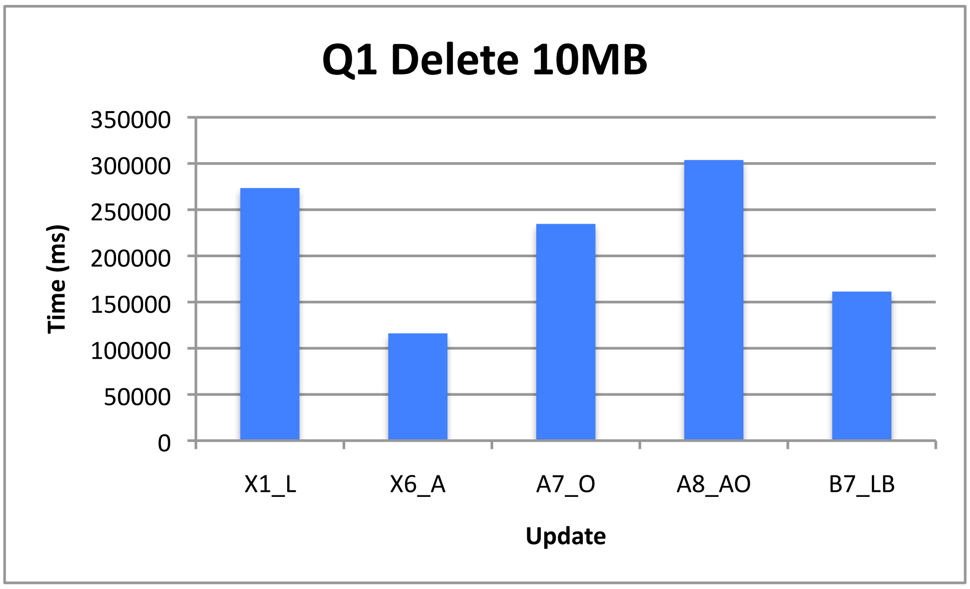
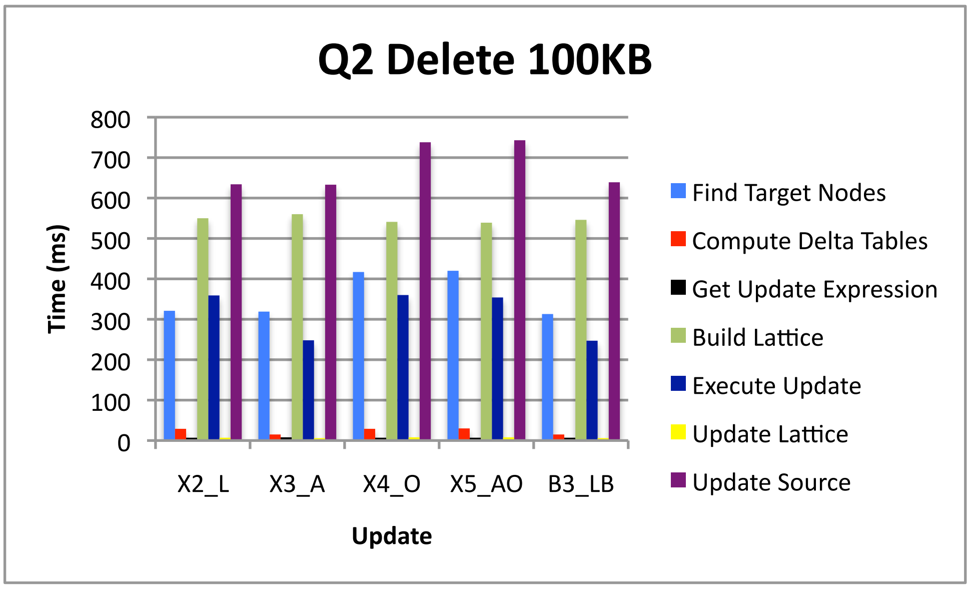


FIGURE C.32: Q1 Delete Graph 10MB

Query	TN	DT	UE	BL	EU	UL	US
X2_L	321	29	7	550	359	7	634
X3_A	319	15	8	560	248	6	633
X4_O	417	29	7	541	360	8	738
X5_AO	420	30	7	539	354	8	743
B3_LB	313	15	7	546	247	6	639

TABLE C.33: Q2 Delete 100Kb



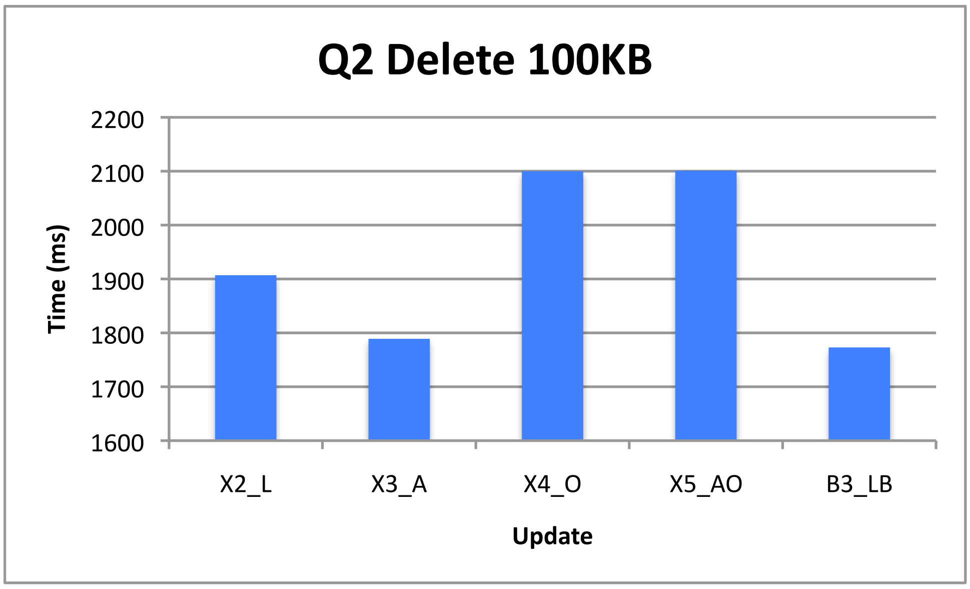
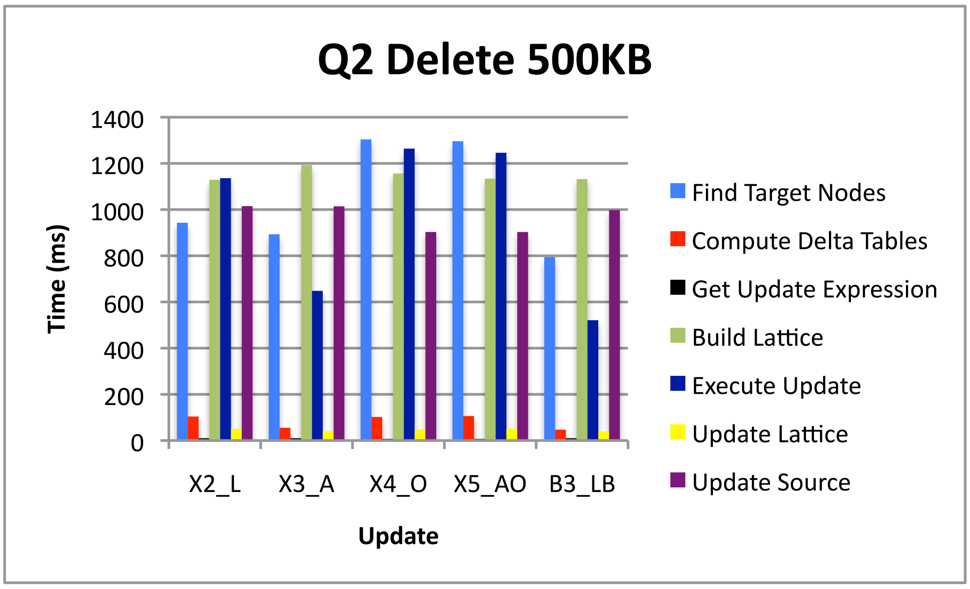


Figure C.33: Q2 Delete Graph 100KB

Query	TN	DT	UE	BL	EU	UL	US
X2_L	943	104	10	1129	1136	50	1015
X3_A	893	55	10	1193	648	41	1014
X4_O	1304	102	7	1156	1264	48	903
X5_AO	1296	106	7	1134	1246	51	903
B3_LB	794	47	10	1132	521	39	998

Table C.34: Q2 Delete 500Kb



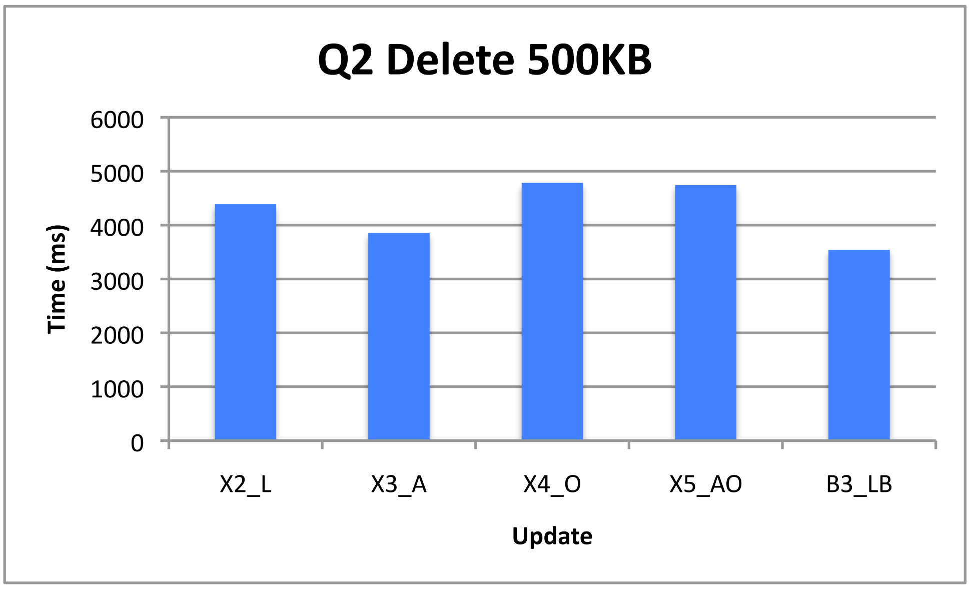
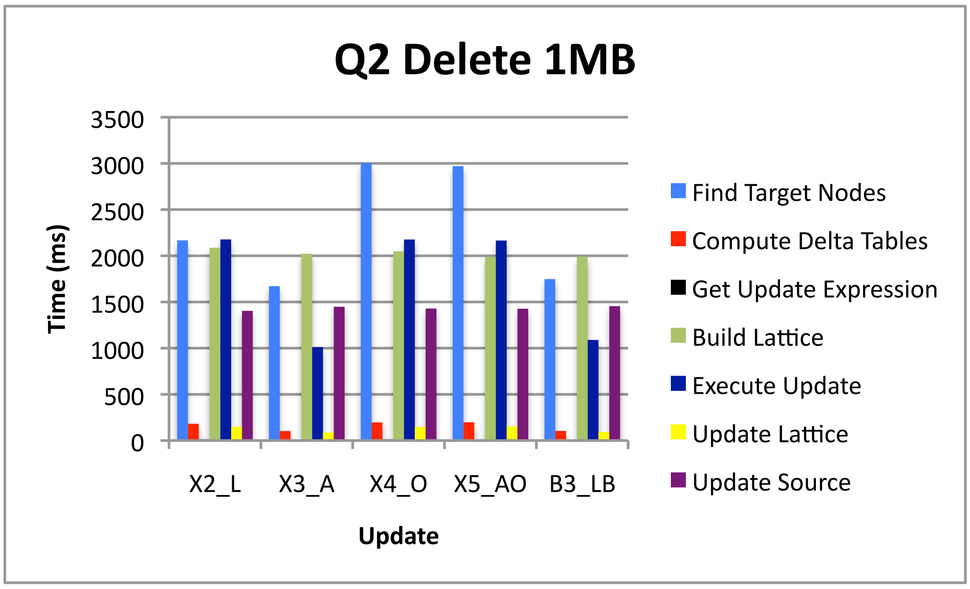


FIGURE C.34: Q2 Delete Graph 500KB

Query	TN	DT	UE	BL	EU	UL	US
X2_L	2167	182	14	2087	2177	148	1404
X3_A	1671	103	11	2022	1011	88	1448
X4_O	3003	196	14	2047	2176	147	1429
X5_AO	2970	198	13	1987	2165	153	1427
B3_LB	1748	104	11	1990	1089	93	1455

TABLE C.35: Q2 Delete 1Mb



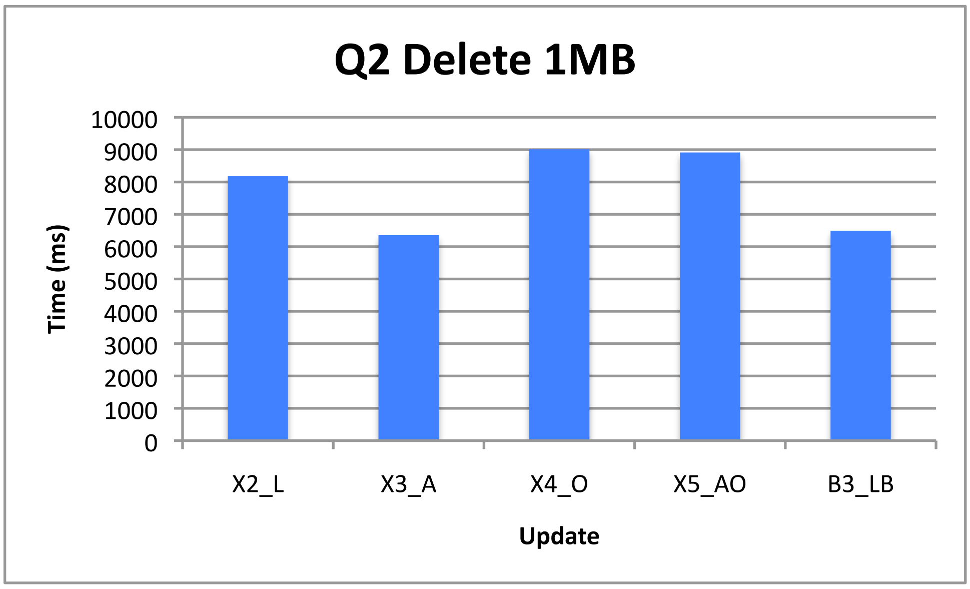
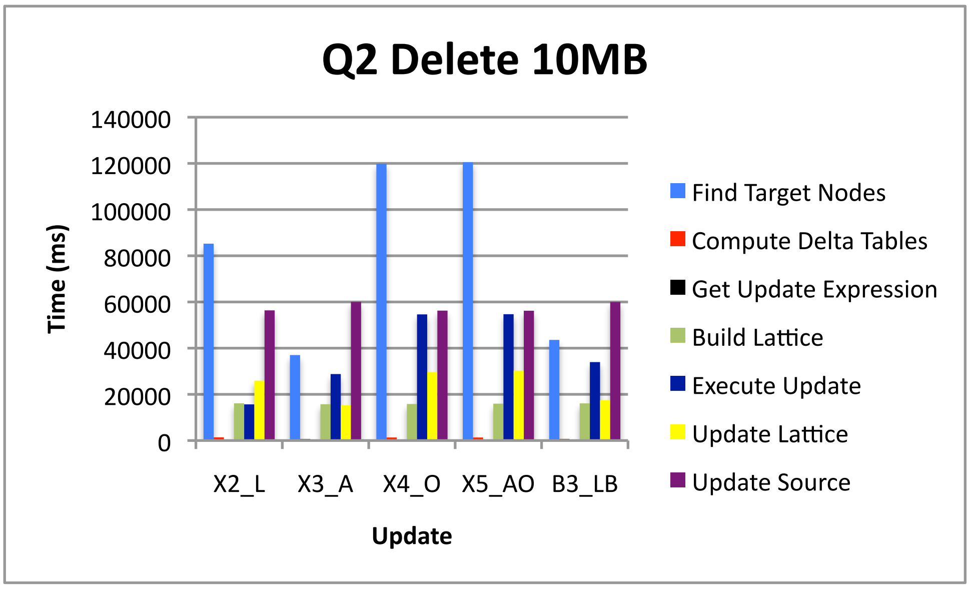


Figure C.35: Q2 Delete Graph 1MB

Query	TN	DT	UE	BL	EU	UL	US
X2_L	85242	1385	28	16119	15660	25916	56364
X3_A	37022	707	20	15726	28790	15270	60053
X4_O	119892	1327	31	15824	54624	29570	56256
X5_AO	120504	1321	30	15967	54702	30244	56214
B3_LB	43574	738	21	16094	33968	17413	60012

Table C.36: Q2 Delete 10Mb



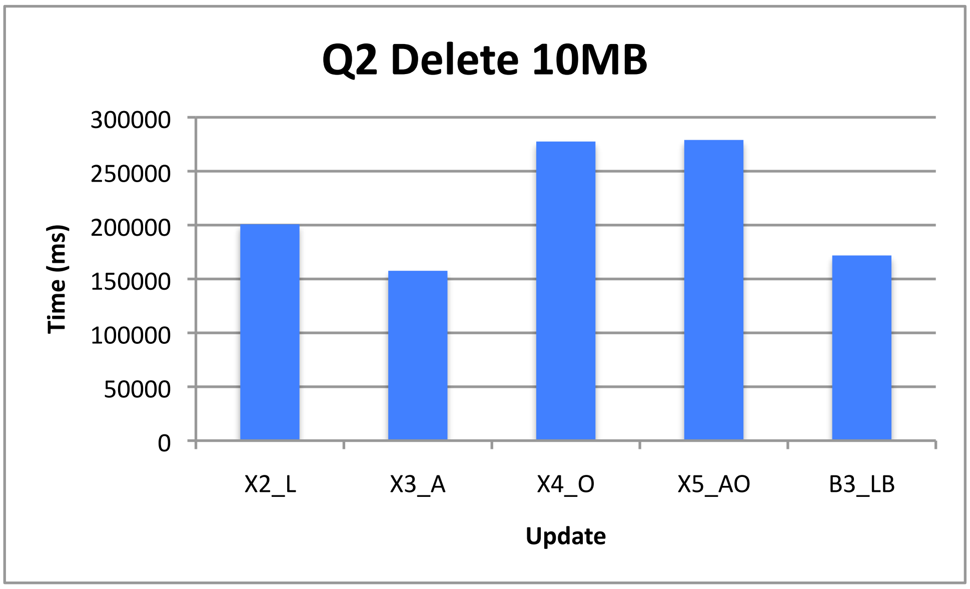
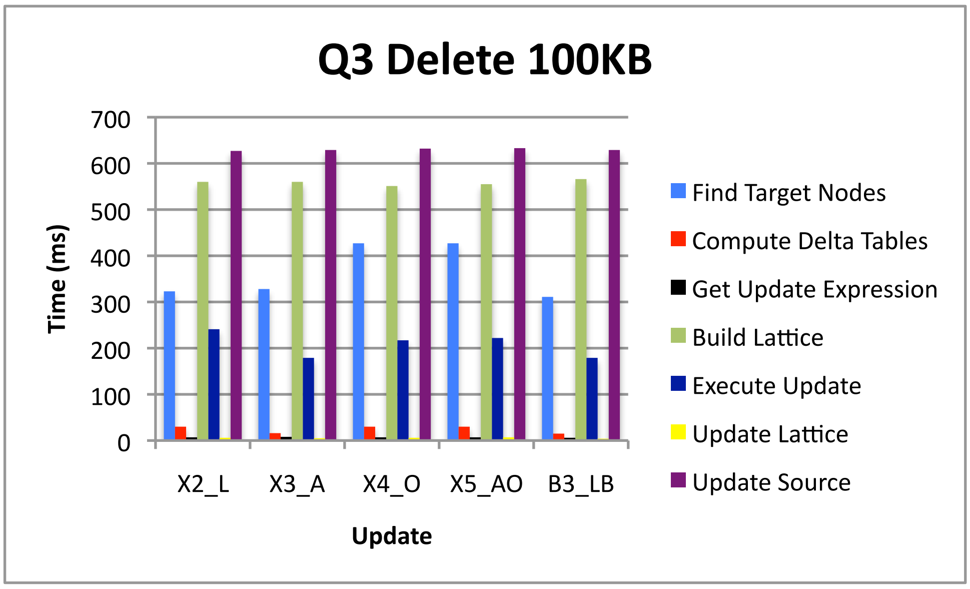


FIGURE C.36: Q2 Delete Graph 10MB

Query	TN	DT	UE	BL	EU	UL	US
X2_L	323	30	7	560	241	6	627
X3_A	328	16	8	560	179	5	629
X4_O	427	30	7	551	217	6	632
X5_AO	427	30	7	555	222	7	633
B3_LB	311	15	6	566	179	4	629

TABLE C.37: Q3 Delete 100Kb



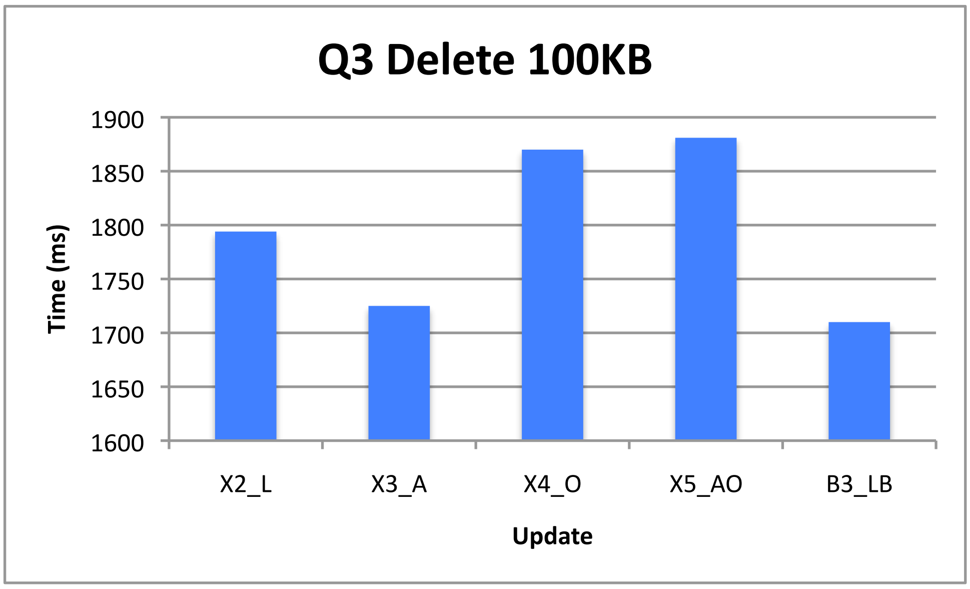
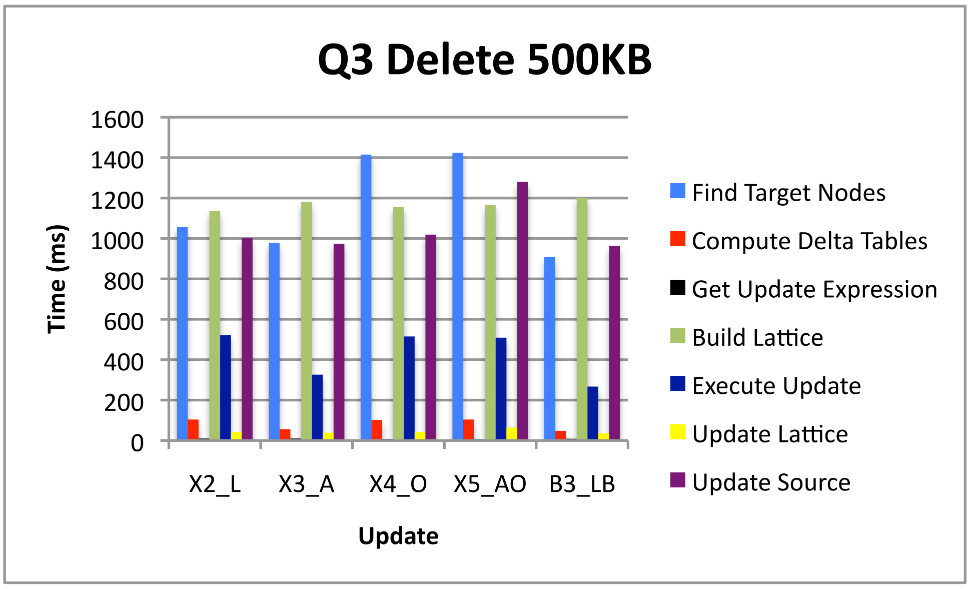


Figure C.37: Q3 Delete Graph 100KB

Query	TN	DT	UE	BL	EU	UL	US
X2_L	1056	104	10	1135	521	43	1001
X3_A	978	56	10	1181	326	39	974
X4_O	1415	102	8	1155	515	43	1019
X5_AO	1423	104	7	1166	509	64	1280
B3_LB	909	48	9	1201	267	35	963

Table C.38: Q3 Delete 500Kb



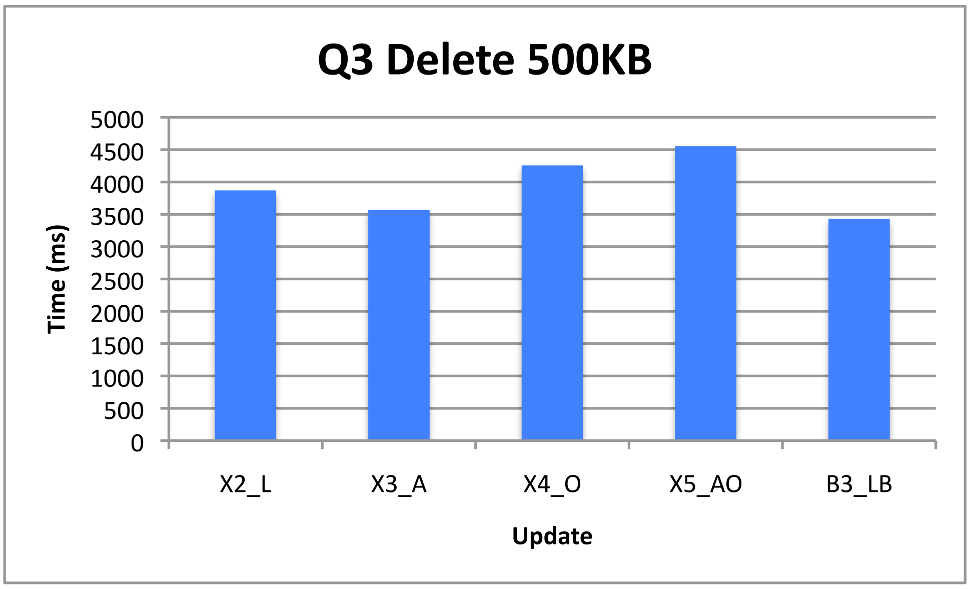
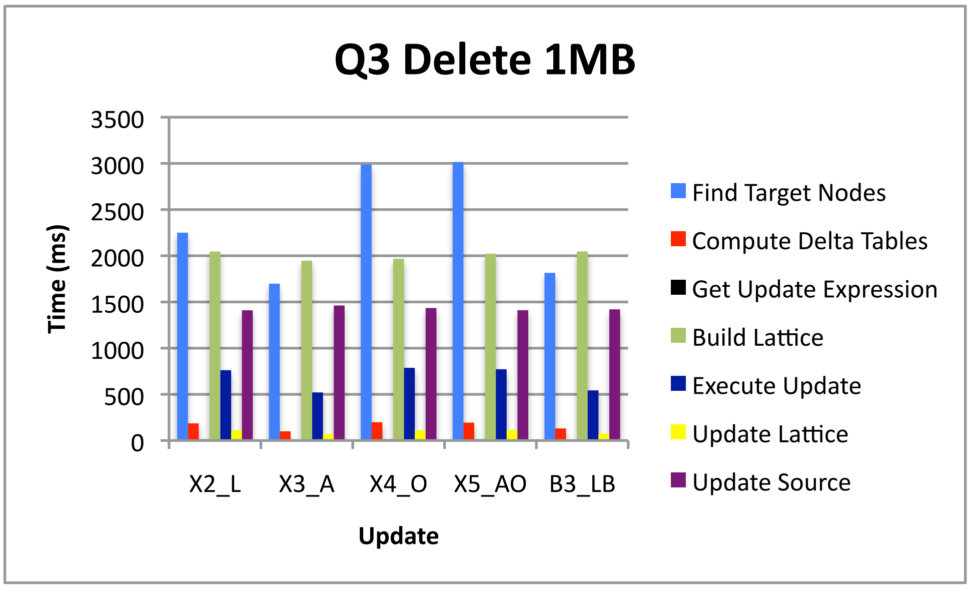


FIGURE C.38: Q3 Delete Graph 500KB

Query	TN	DT	UE	BL	EU	UL	US
X2_L	2250	186	14	2046	762	110	1410
X3_A	1698	100	8	1947	521	69	1462
X4_O	2988	198	14	1967	787	109	1435
X5_AO	3015	194	14	2023	772	112	1411
B3_LB	1815	131	10	2047	543	75	1420

TABLE C.39: Q3 Delete 1Mb



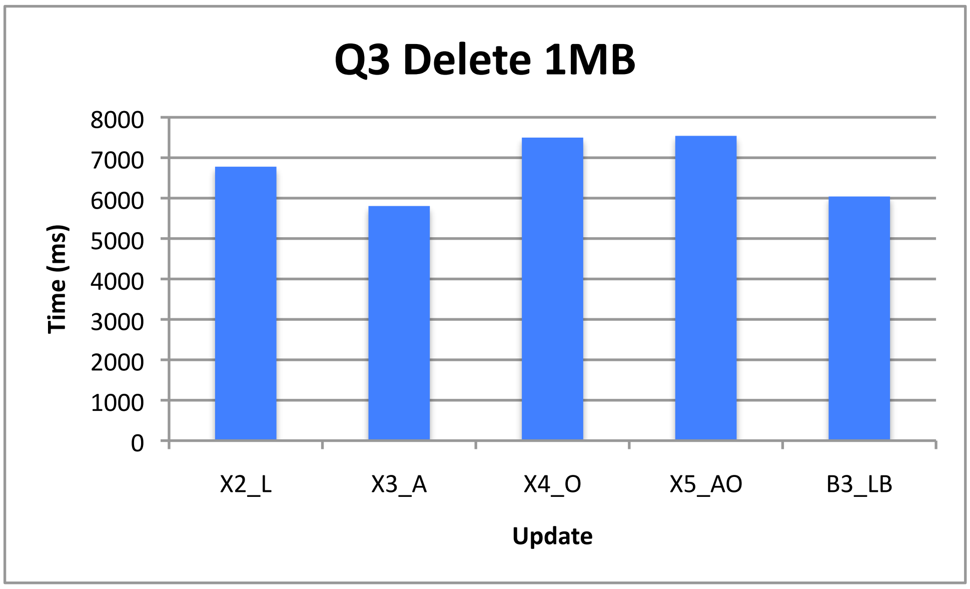
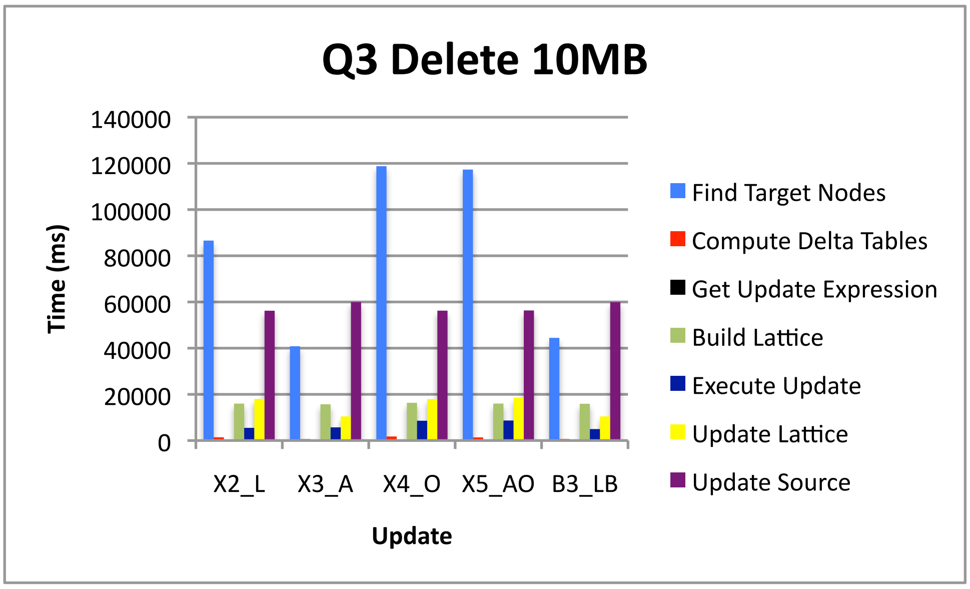


Figure C.39: Q3 Delete Graph 1MB

Query	TN	DT	UE	BL	EU	UL	US
X2_L	86574	1411	30	15983	5492	17999	56214
X3_A	40830	626	20	15691	5710	10454	59975
X4_O	118784	1761	30	16335	8574	17938	56237
X5_AO	117323	1359	30	16004	8675	18554	56328
B3_LB	44454	666	21	15909	4959	10452	59941

Table C.40: Q3 Delete 10Mb



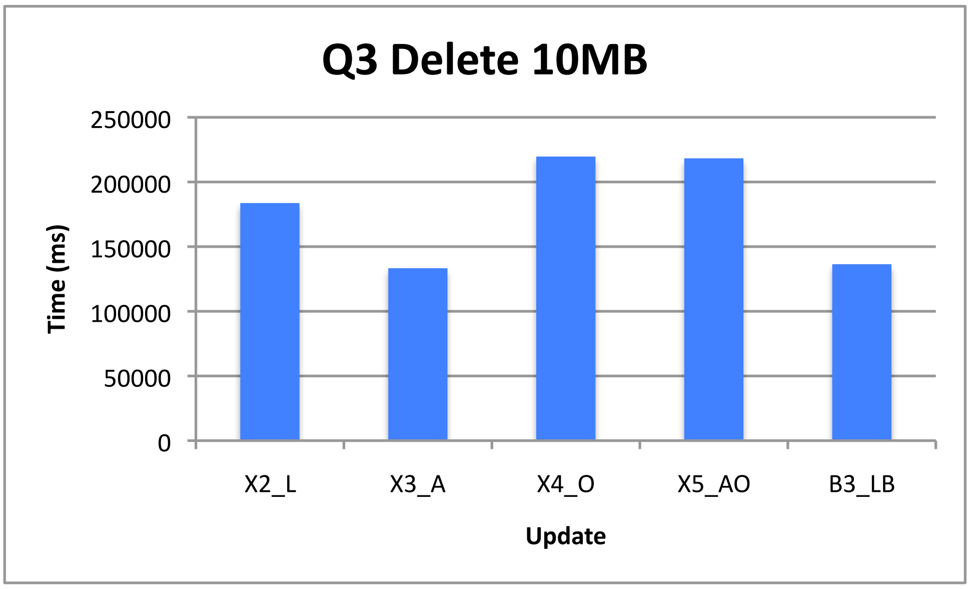
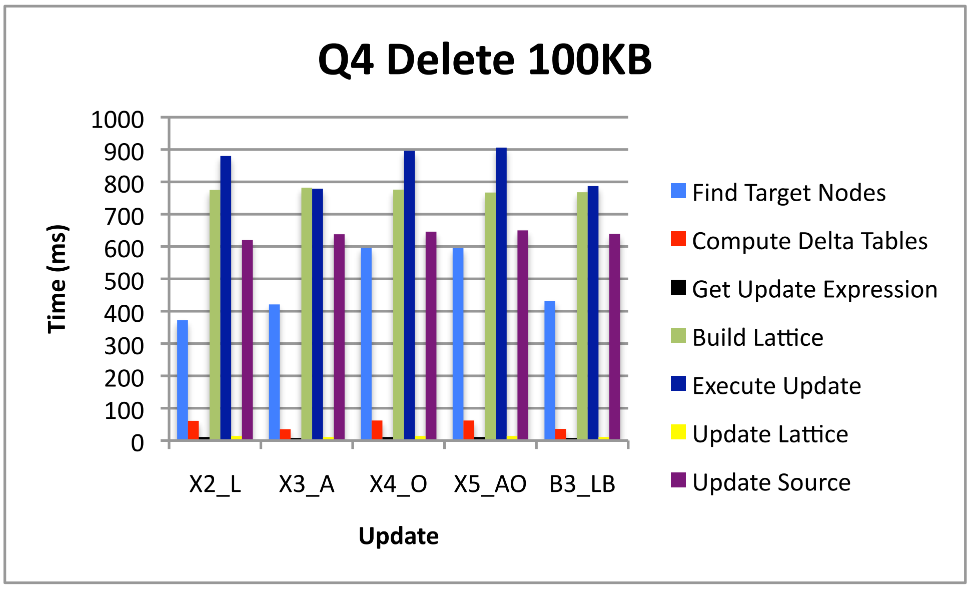


FIGURE C.40: Q3 Delete Graph 10MB

Query	TN	DT	UE	BL	EU	UL	US
X2_L	372	61	11	775	880	14	620
X3_A	421	35	8	782	779	11	638
X4_O	596	62	11	776	896	14	646
X5_AO	595	62	11	767	906	14	650
B3_LB	432	36	8	768	787	11	639

TABLE C.41: Q4 Delete 100Kb



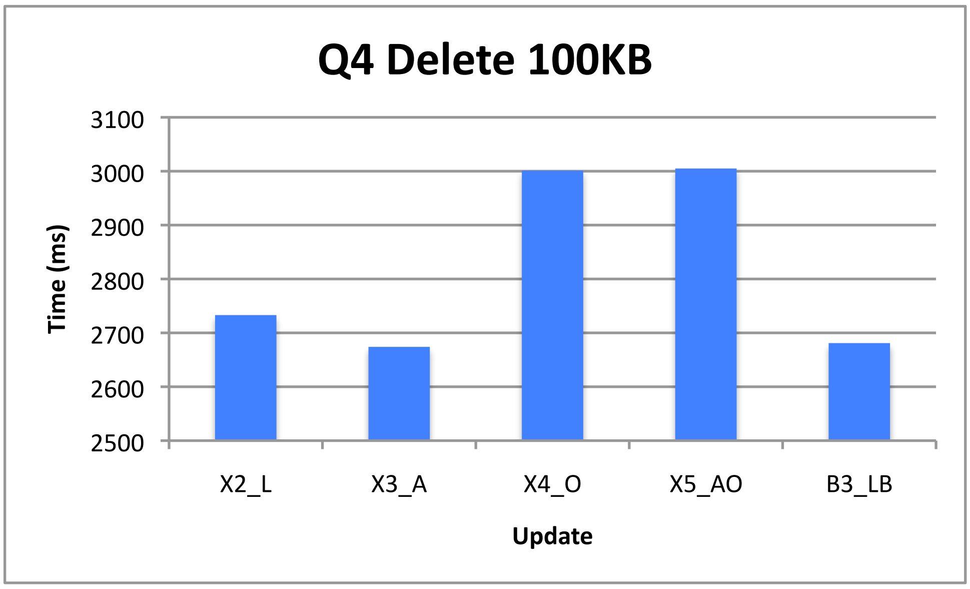
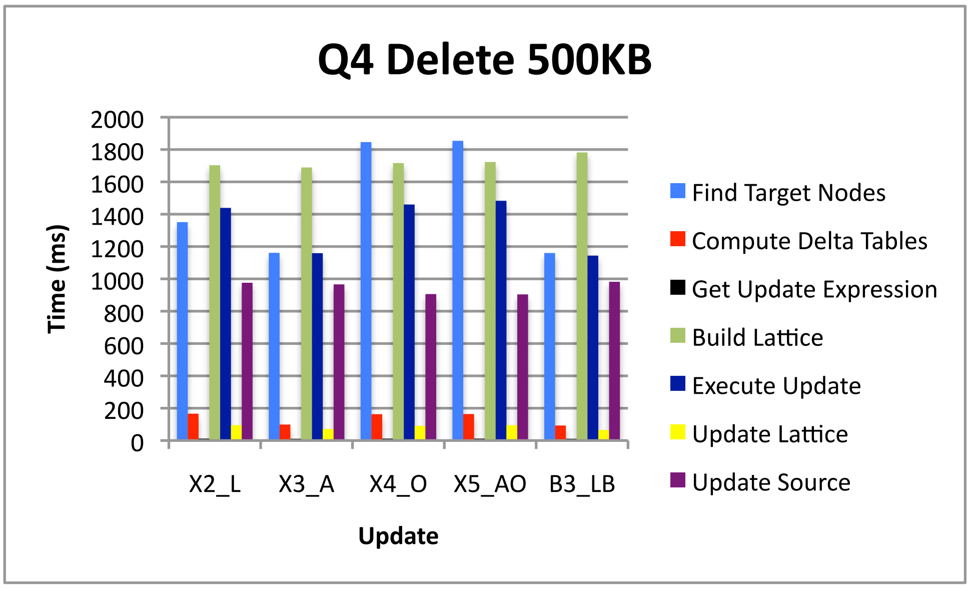


Figure C.41: Q4 Delete Graph 100KB

Query	TN	DT	UE	BL	EU	UL	US
X2_L	1351	166	12	1703	1439	95	976
X3_A	1161	99	11	1689	1159	71	966
X4_O	1846	163	12	1716	1460	91	906
X5_AO	1854	164	12	1723	1483	95	904
B3_LB	1160	93	11	1782	1144	65	982

Table C.42: Q4 Delete 500Kb



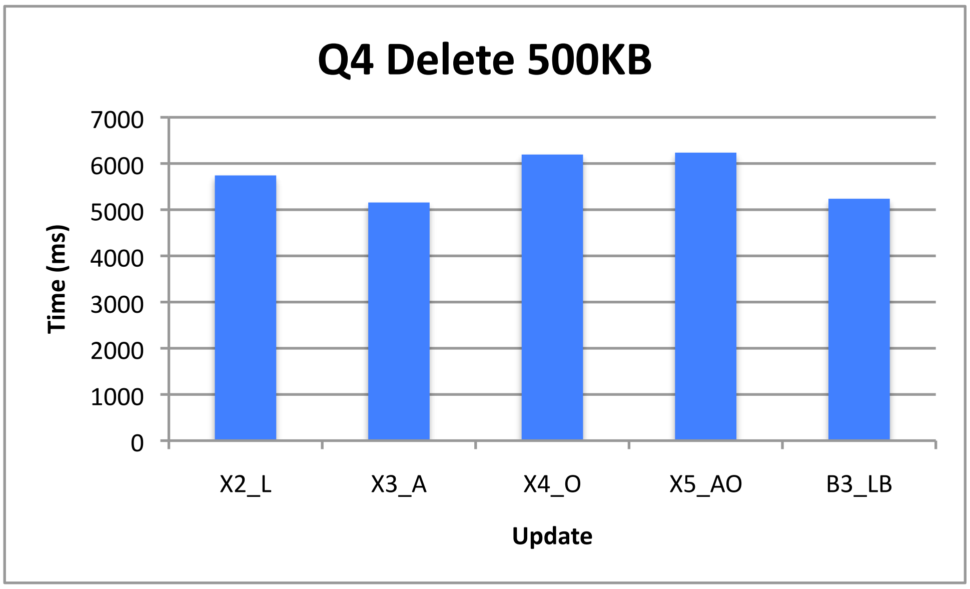
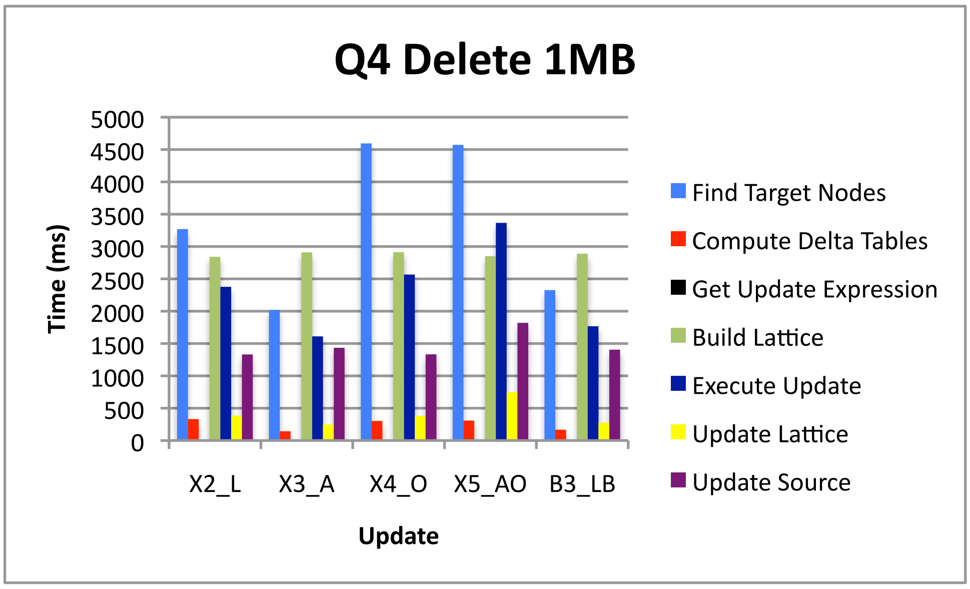


FIGURE C.42: Q4 Delete Graph 500KB

Query	TN	DT	UE	BL	EU	UL	US
X2_L	3270	332	16	2841	2377	385	1331
X3_A	2020	144	12	2910	1611	248	1434
X4_O	4595	303	15	2913	2567	382	1333
X5_AO	4572	309	15	2851	3366	747	1820
B3_LB	2326	167	12	2889	1768	275	1405

TABLE C.43: Q4 Delete 1Mb



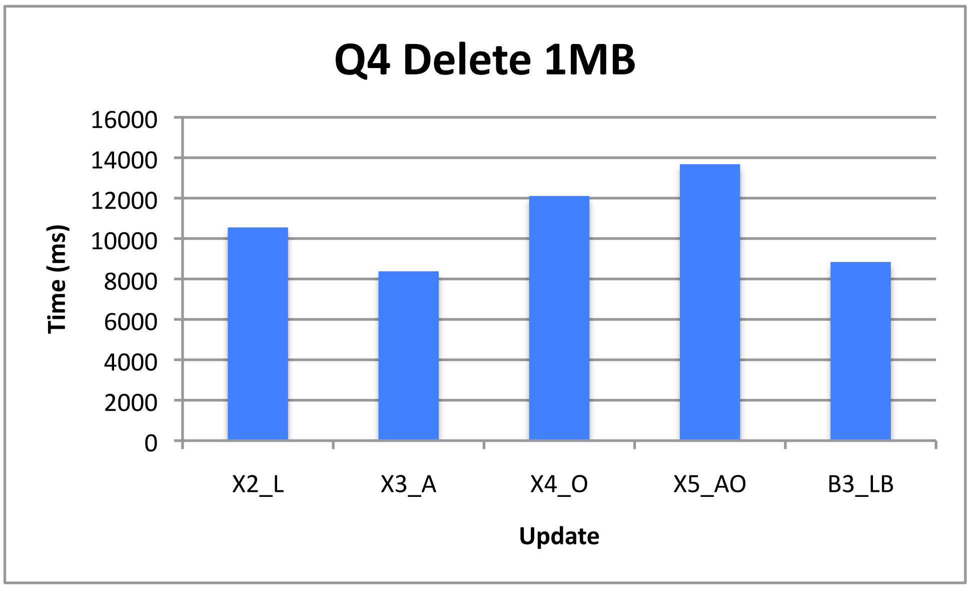
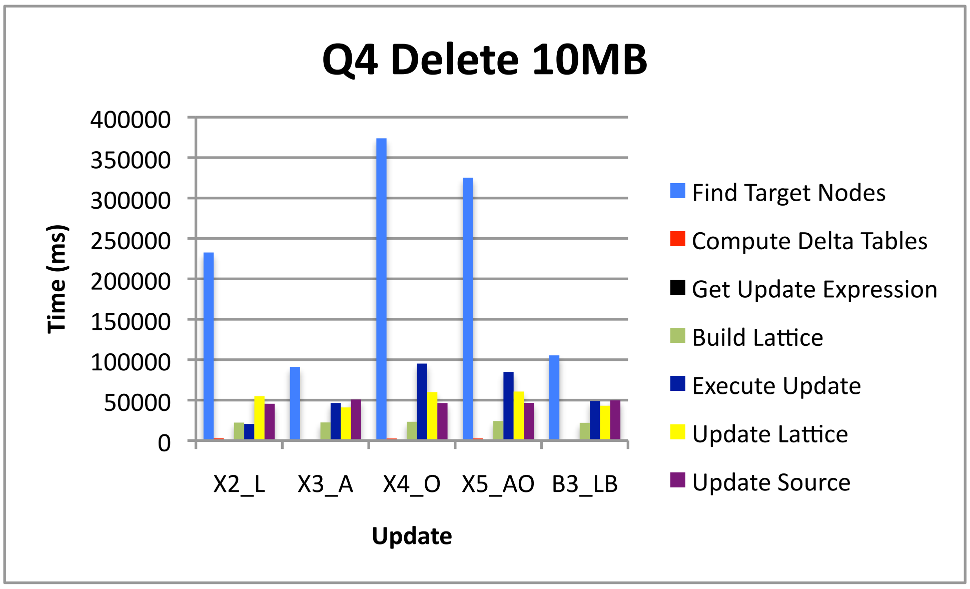


Figure C.43: Q4 Delete Graph 1MB

Query	TN	DT	UE	BL	EU	UL	US
X2_L	232638	2660	28	22206	20548	54874	45512
X3_A	91149	1233	21	22402	46486	40961	50739
X4_O	373899	2524	29	23235	95202	59881	46393
X5_AO	325167	2594	30	24149	84947	60618	46640
B3_LB	105372	1293	21	21957	48908	43071	49689

Table C.44: Q4 Delete 10Mb



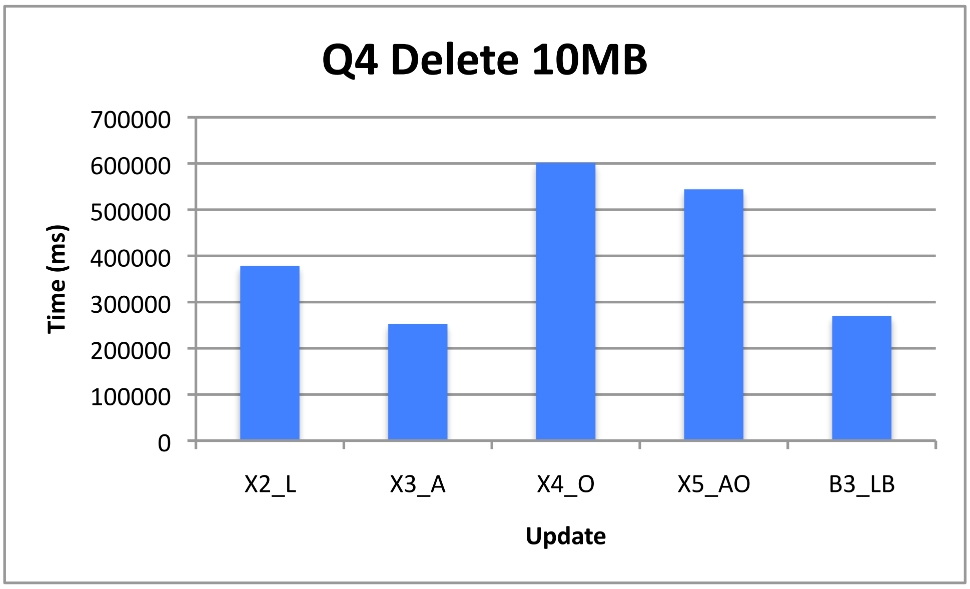
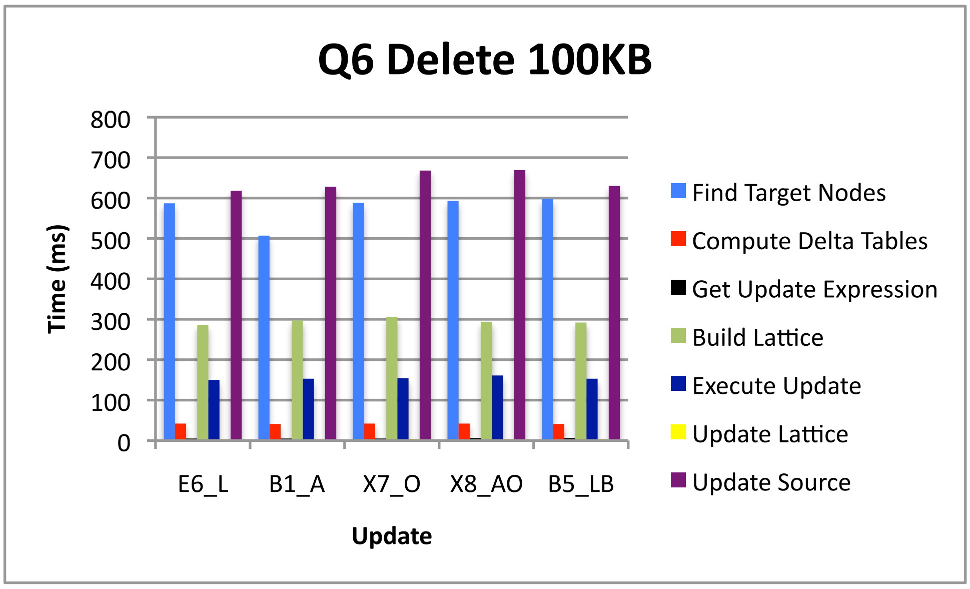


FIGURE C.44: Q4 Delete Graph 10MB

Query	TN	DT	UE	BL	EU	UL	US
E6_L	587	42	5	286	150	3	618
B1_A	507	41	5	297	153	3	628
X7_O	588	42	5	306	154	4	668
X8_AO	593	42	6	294	161	4	669
B5_LB	598	41	6	292	153	4	630

TABLE C.45: Q6 Delete 100Kb



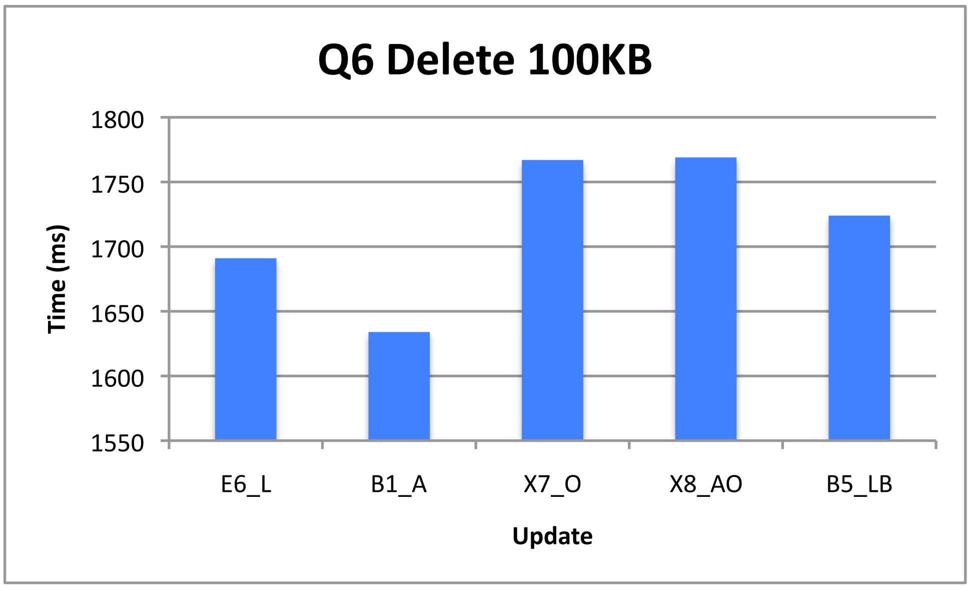
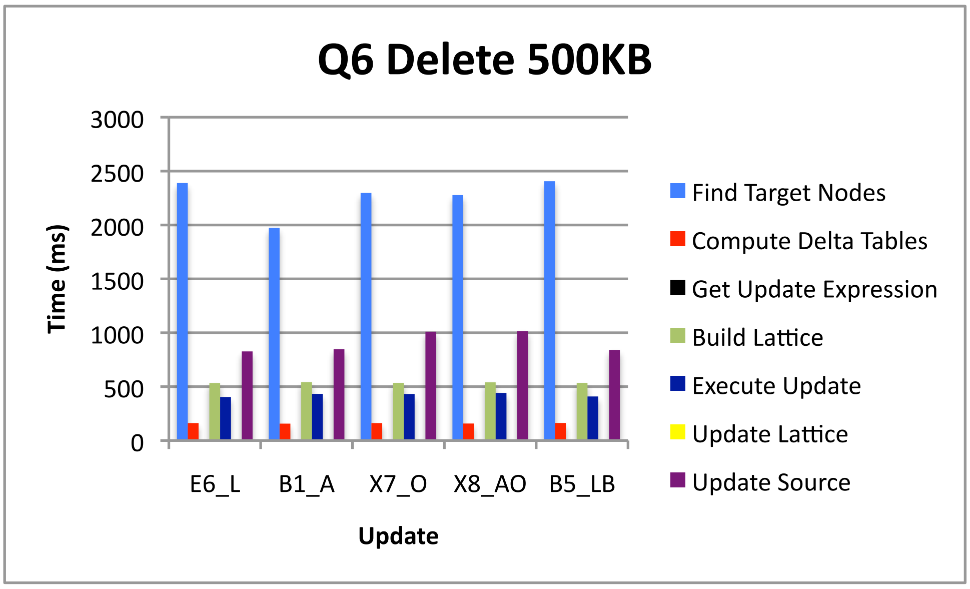


Figure C.45: Q6 Delete Graph 100KB

Query	TN	DT	UE	BL	EU	UL	US
E6_L	2389	162	8	534	404	8	827
B1_A	1974	157	5	542	433	9	847
X7_O	2297	162	6	535	432	8	1010
X8_AO	2277	158	6	540	442	9	1015
B5_LB	2406	163	8	535	409	9	841

Table C.46: Q6 Delete 500Kb



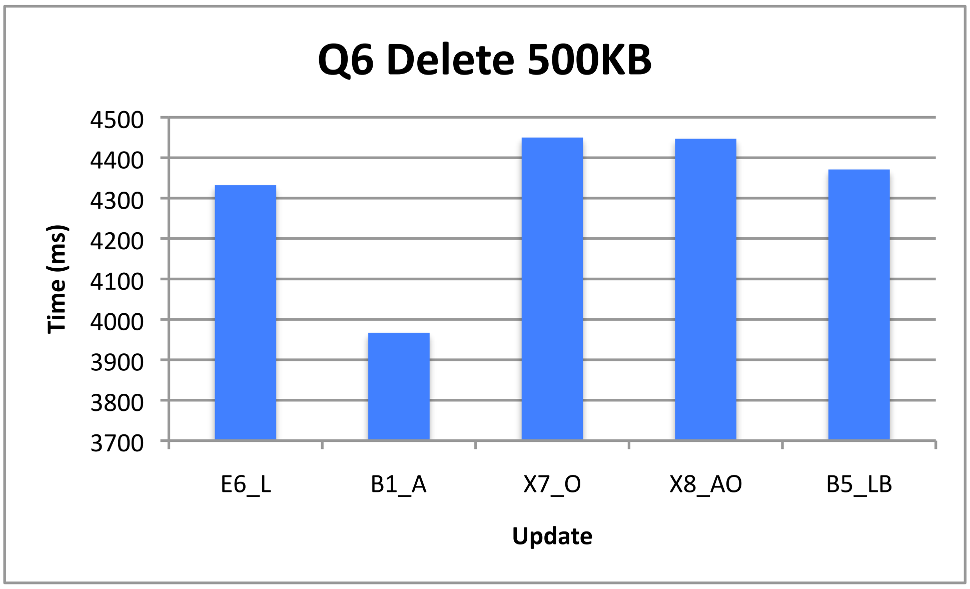
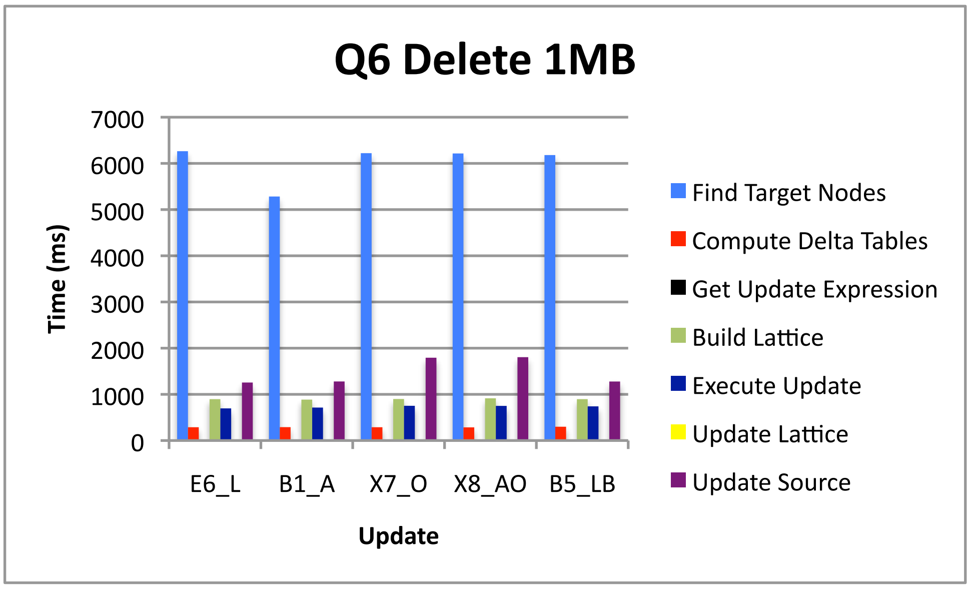


FIGURE C.46: Q6 Delete Graph 500KB

Query	TN	DT	UE	BL	EU	UL	US
E6_L	6264	288	8	896	696	17	1256
B1_A	5283	290	8	886	714	17	1279
X7_O	6221	288	9	899	752	17	1793
X8_AO	6215	285	8	914	750	16	1805
B5_LB	6180	299	8	896	740	17	1278

TABLE C.47: Q6 Delete 1Mb



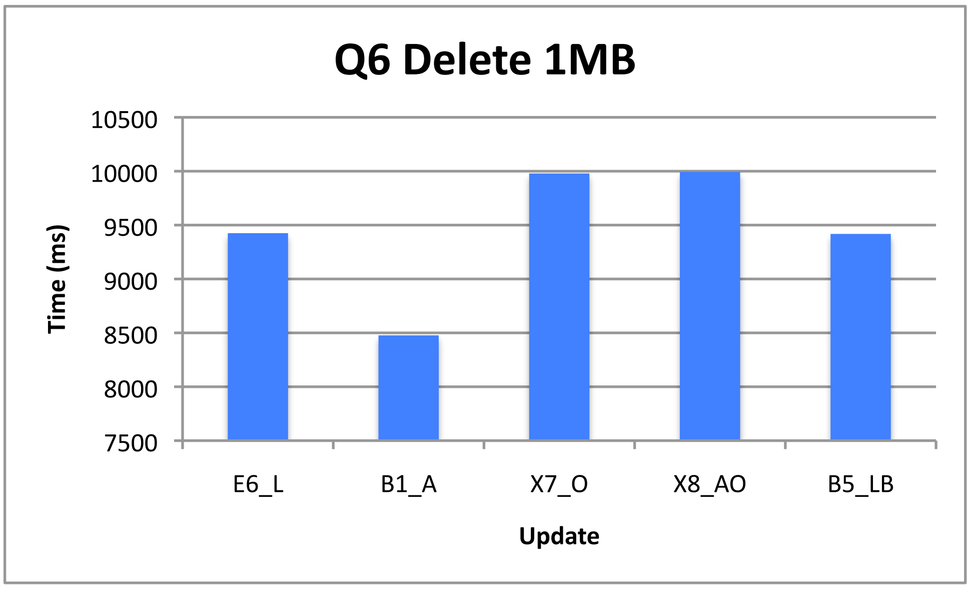
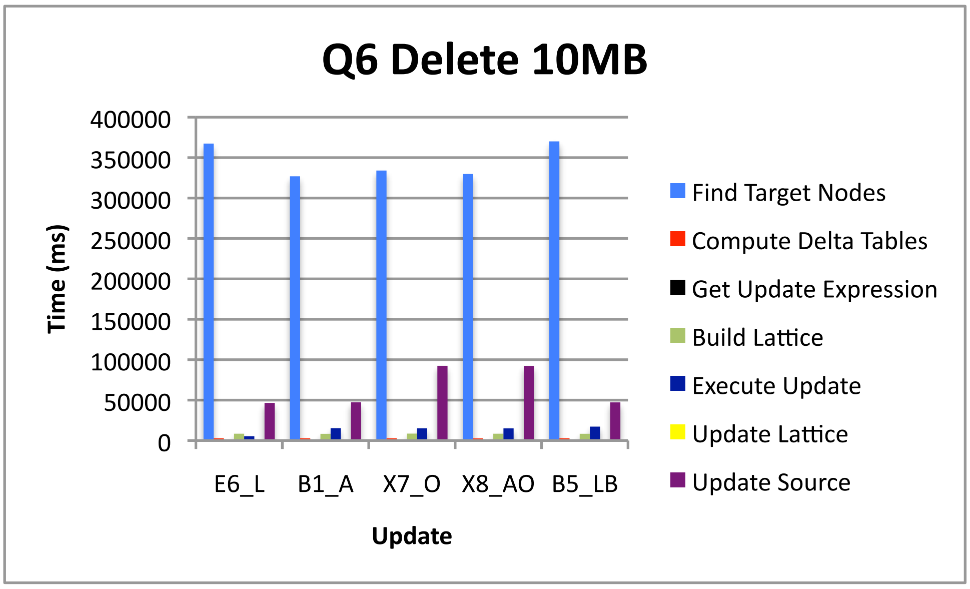


Figure C.47: Q6 Delete Graph 1MB

Query	TN	DT	UE	BL	EU	UL	US
E6_L	367381	2687	8	8457	5336	1055	46556
B1_A	326898	2694	8	8315	15289	1030	47302
X7_O	334017	2679	9	8580	15096	1089	92474
X8_AO	329727	2634	9	8442	15039	1103	92372
B5_LB	370086	2704	8	8365	17229	1059	47219

Table C.48: Q6 Delete 10Mb



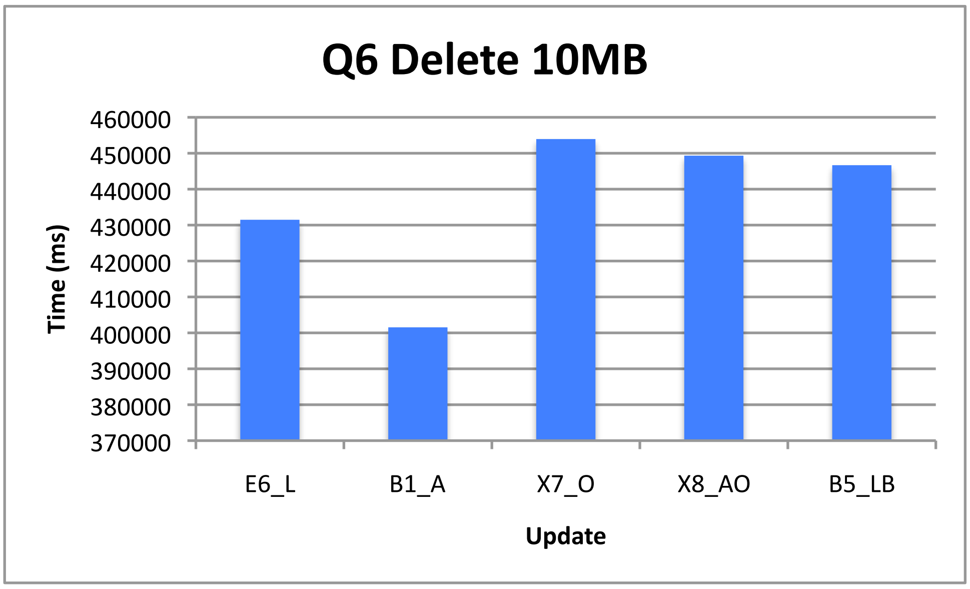
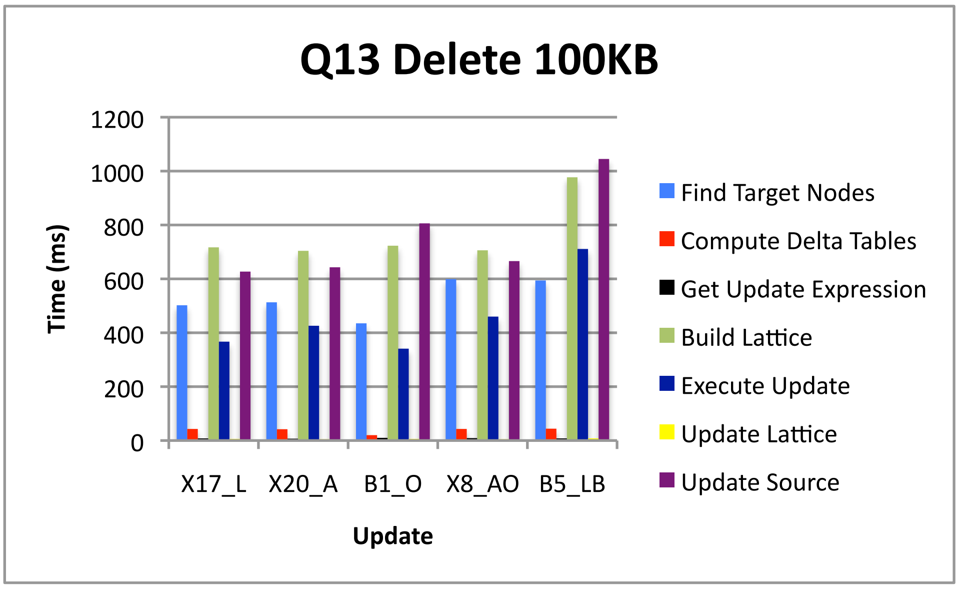


FIGURE C.48: Q6 Delete Graph 10MB

Query	TN	DT	UE	BL	EU	UL	US
X17_L	502	43	8	717	367	6	627
X20_A	513	42	7	704	426	5	643
B1_O	435	20	10	723	341	6	806
X8_AO	599	43	9	706	460	5	666
B5_LB	594	44	8	977	711	8	1045

TABLE C.49: Q13 Delete 100Kb



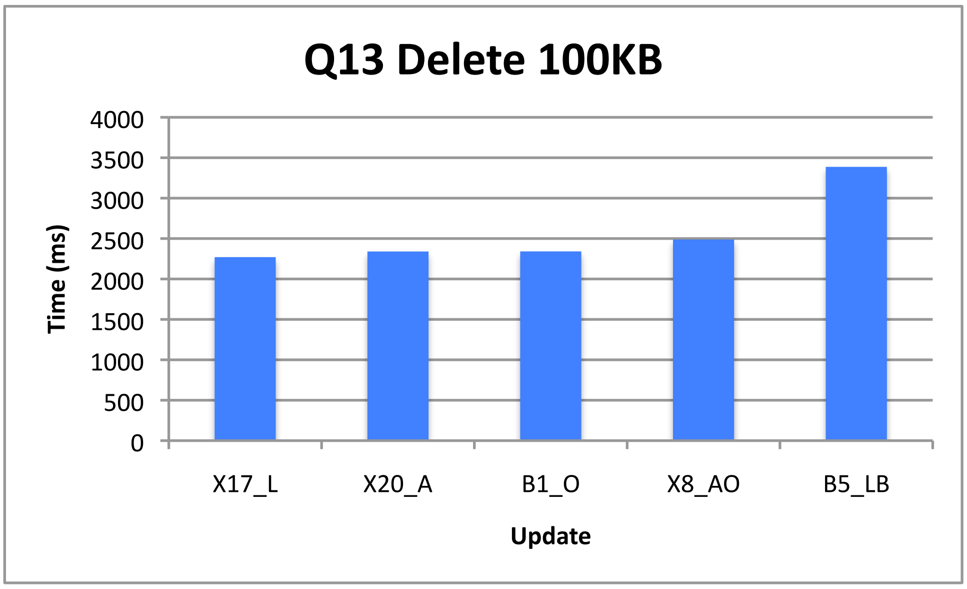
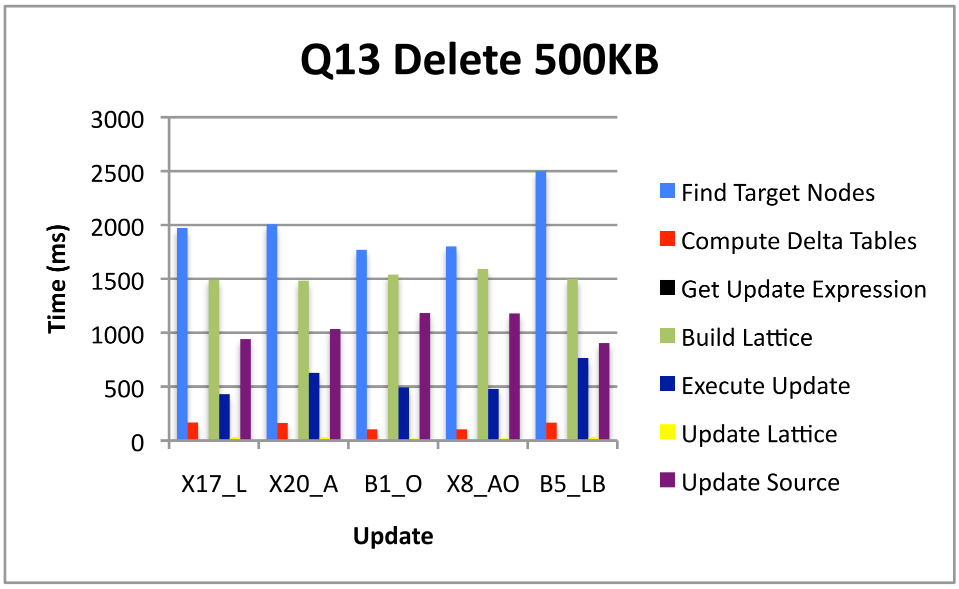


Figure C.49: Q13 Delete Graph 100KB

Query	TN	DT	UE	BL	EU	UL	US
X17_L	1971	167	10	1493	429	29	940
X20_A	2006	164	10	1484	629	29	1035
B1_O	1771	103	11	1540	492	19	1182
X8_AO	1801	103	11	1592	480	20	1179
B5_LB	2500	166	12	1499	767	30	904

Table C.50: Q13 Delete 500Kb



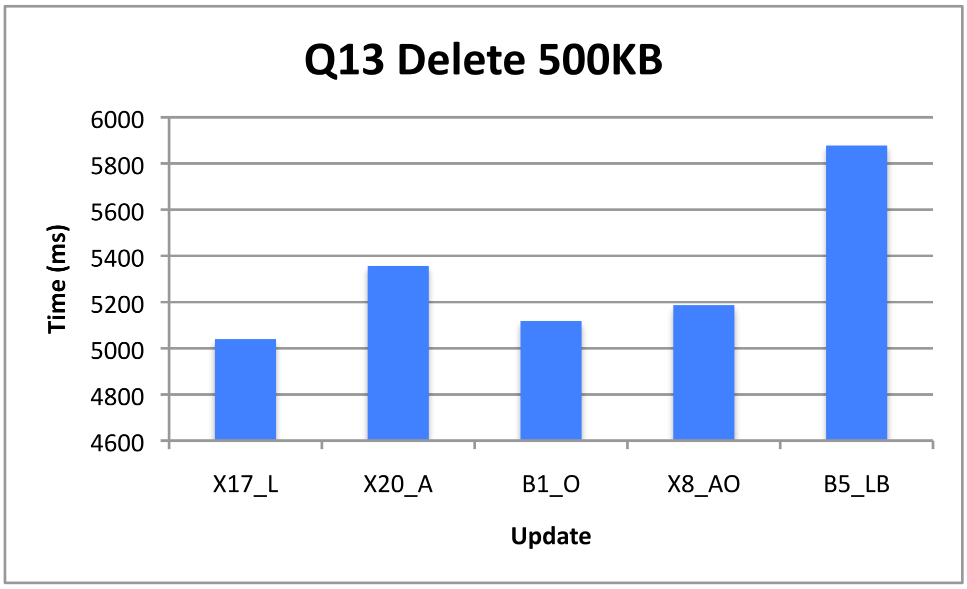
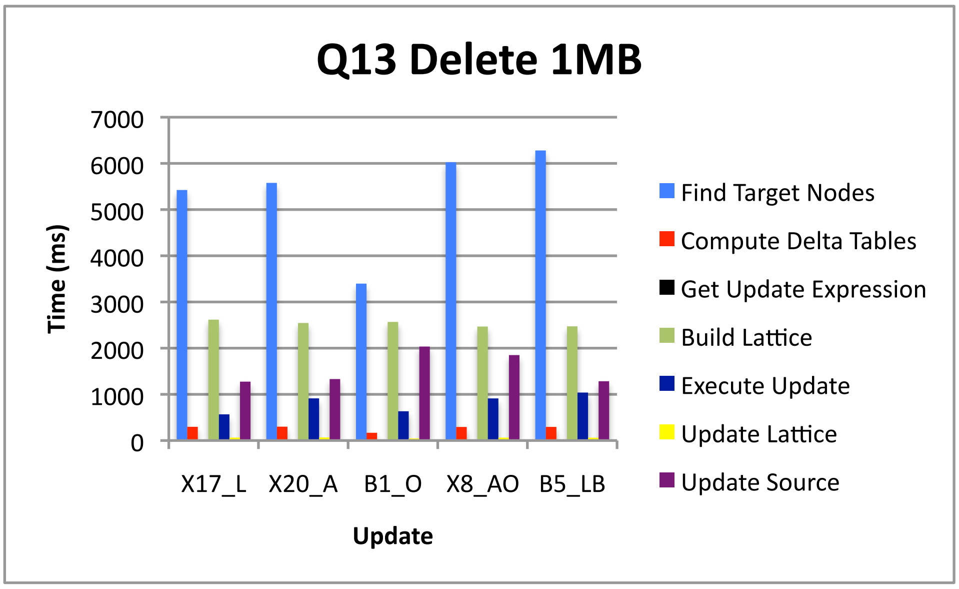


FIGURE C.50: Q13 Delete Graph 500KB

Query	TN	DT	UE	BL	EU	UL	US
X17_L	5424	297	14	2617	567	64	1275
X20_A	5578	300	13	2545	913	67	1331
B1_O	3397	168	14	2568	634	45	2035
X8_AO	6025	294	15	2468	912	63	1850
B5_LB	6280	295	14	2472	1039	62	1284

TABLE C.51: Q13 Delete 1Mb



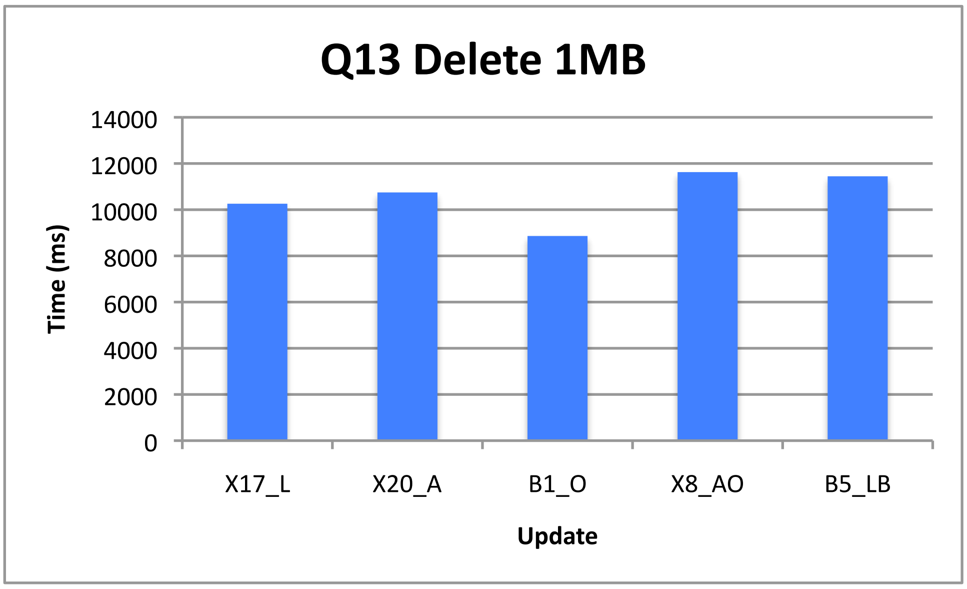
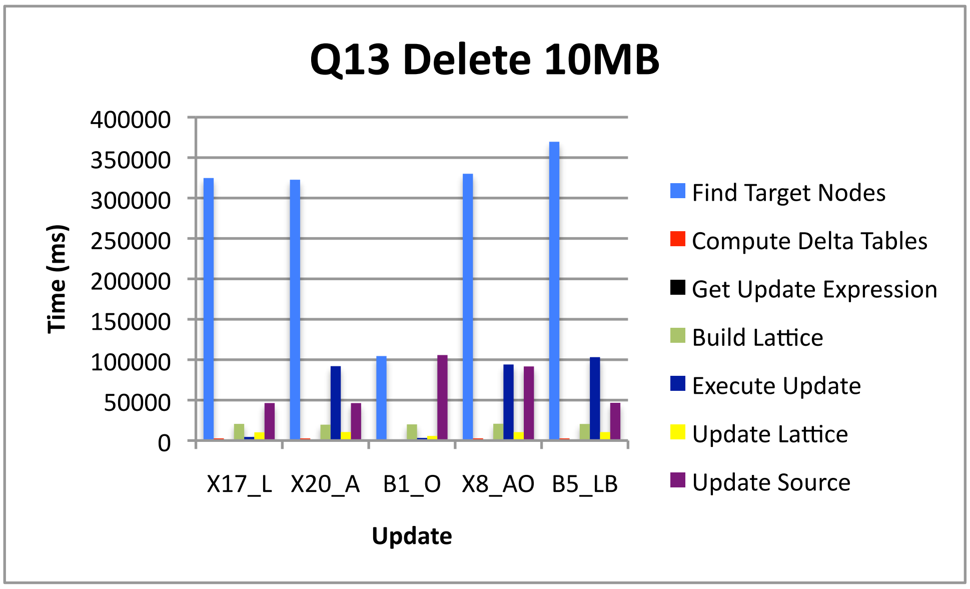


Figure C.51: Q13 Delete Graph 1MB

Query	TN	DT	UE	BL	EU	UL	US
X17_L	324750	2669	37	20588	4449	10168	46380
X20_A	322643	2645	38	19617	92023	10504	46337
B1_O	104507	1407	25	20006	3231	5650	105754
X8_AO	330026	2711	39	20766	94088	10398	91680
B5_LB	369605	2647	38	20496	103162	10425	46672

Table C.52: Q13 Delete 10Mb



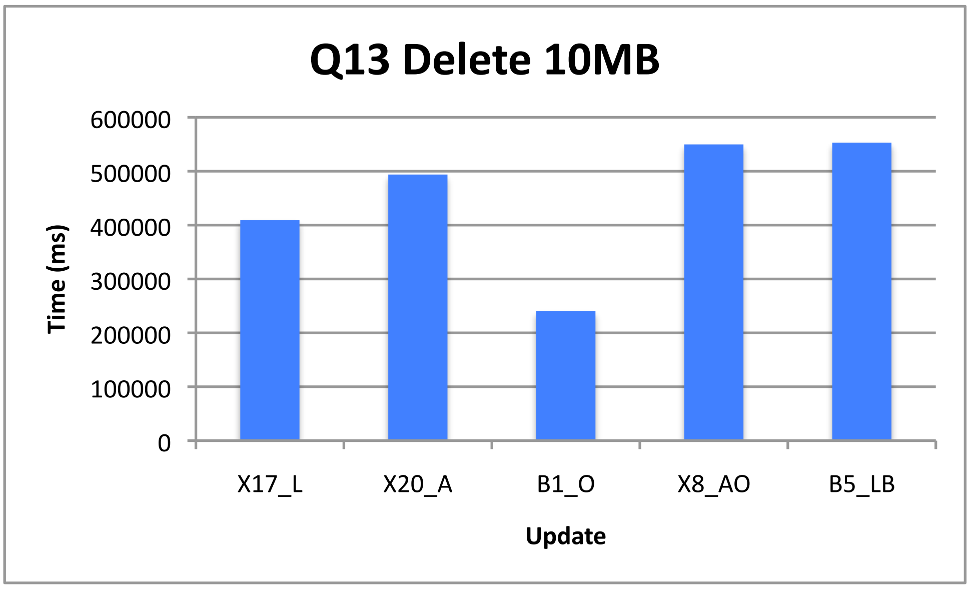
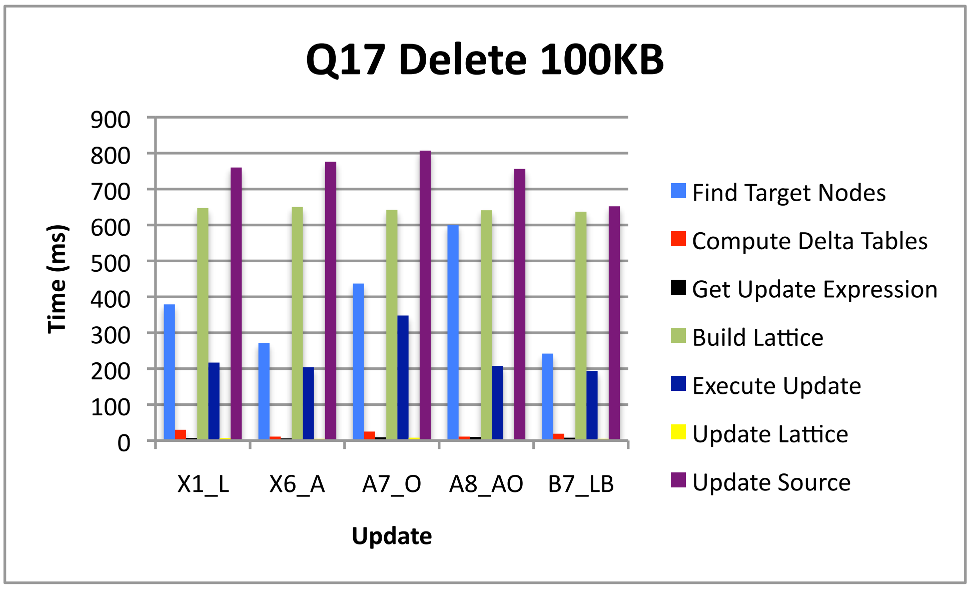


FIGURE C.52: Q13 Delete Graph 10MB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	379	30	7	647	217	7	760
X6_A	272	11	6	650	204	5	776
A7_O	437	25	9	642	348	8	807
A8_AO	600	11	10	641	208	3	756
B7_LB	242	19	8	637	194	5	652

TABLE C.53: Q17 Delete 100Kb



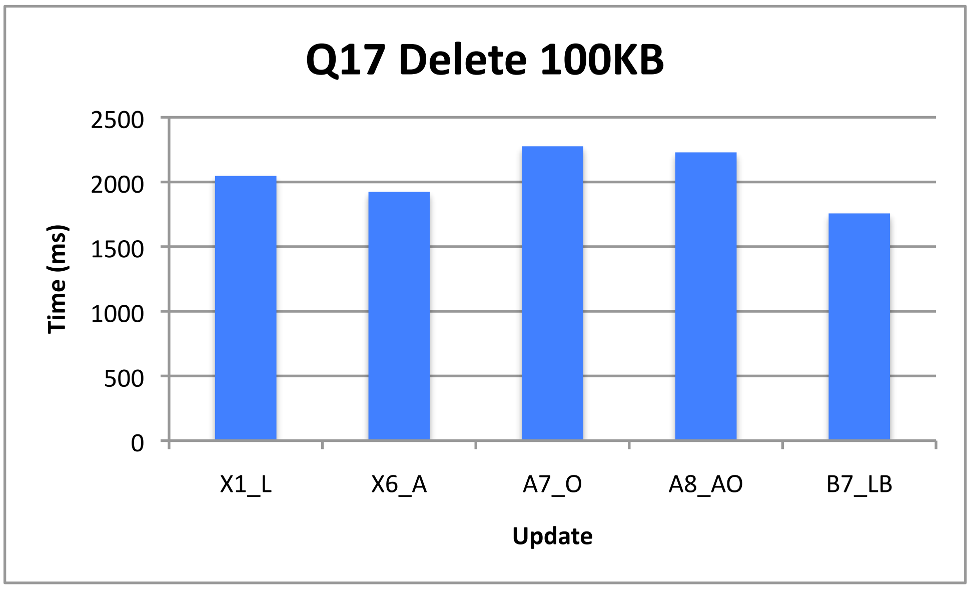
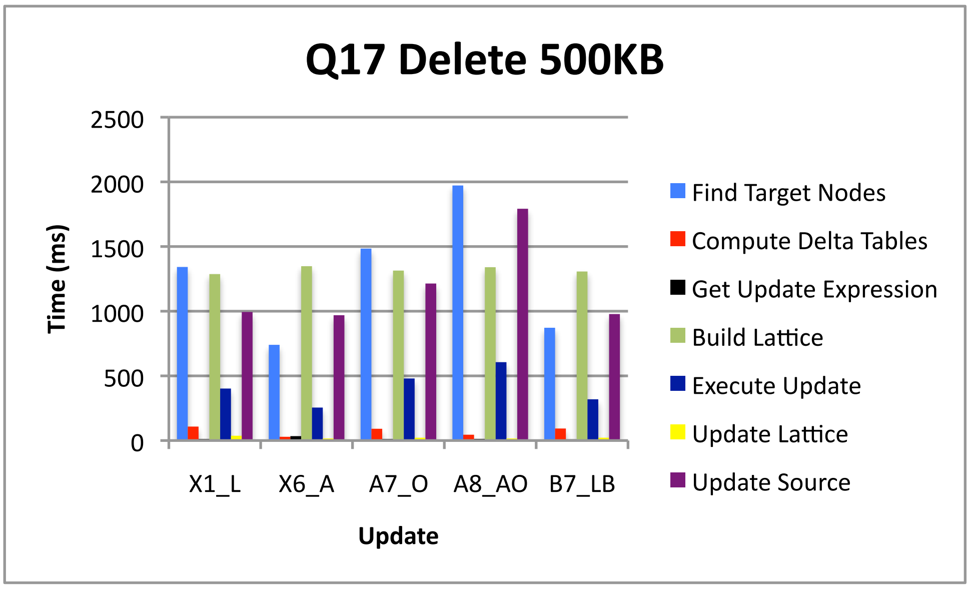


Figure C.53: Q17 Delete Graph 100KB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	1342	108	12	1287	402	37	994
X6_A	740	29	34	1348	255	16	969
A7_O	1484	91	12	1314	480	22	1214
A8_AO	1972	45	13	1340	606	16	1792
B7_LB	872	93	10	1307	319	22	978

Table C.54: Q17 Delete 500Kb



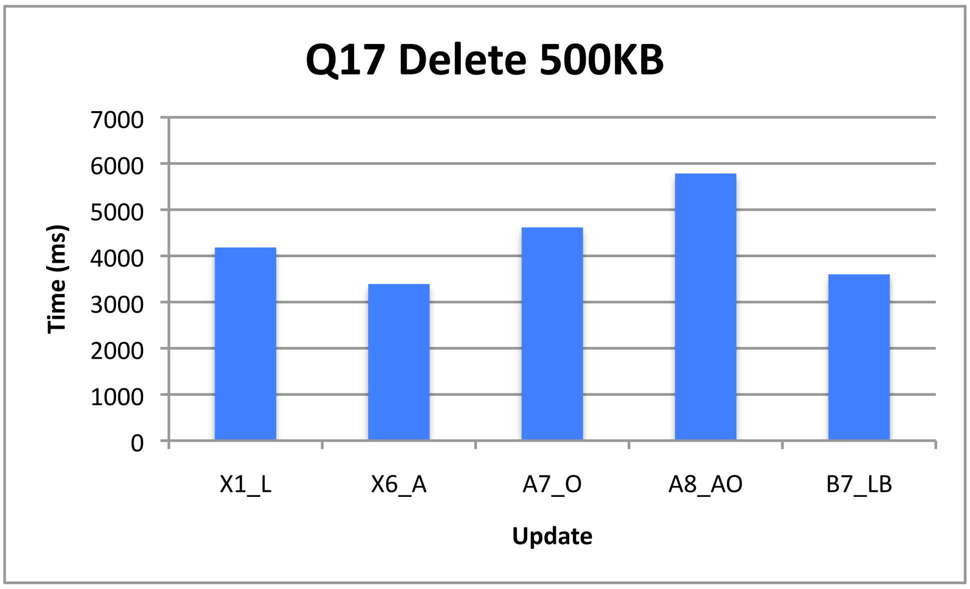
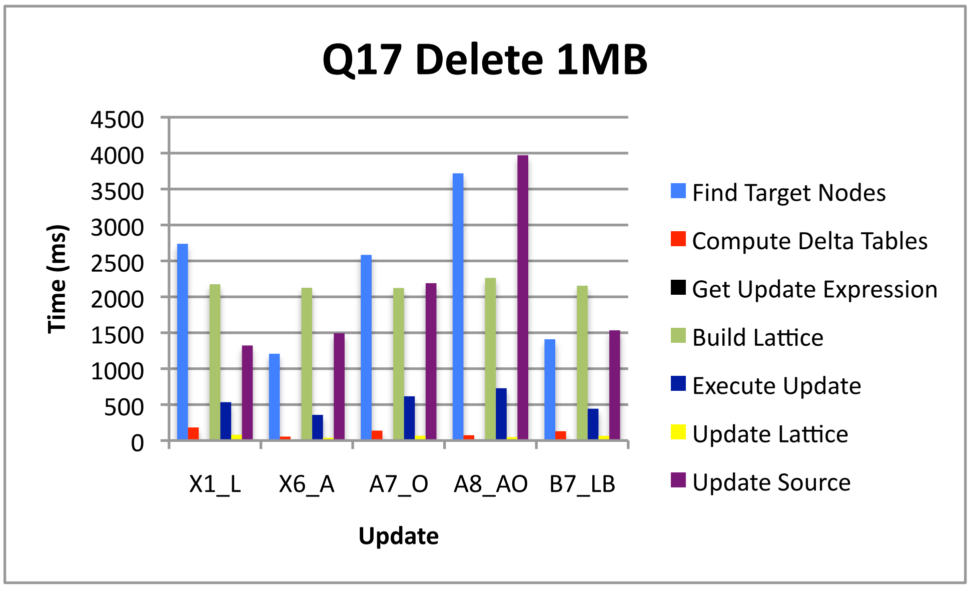


FIGURE C.54: Q17 Delete Graph 500KB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	2738	182	13	2175	534	80	1323
X6_A	1208	55	11	2126	357	40	1494
A7_O	2584	138	13	2123	616	67	2190
A8_AO	3719	74	17	2263	728	48	3971
B7_LB	1410	130	11	2155	443	65	1534

TABLE C.55: Q17 Delete 1Mb



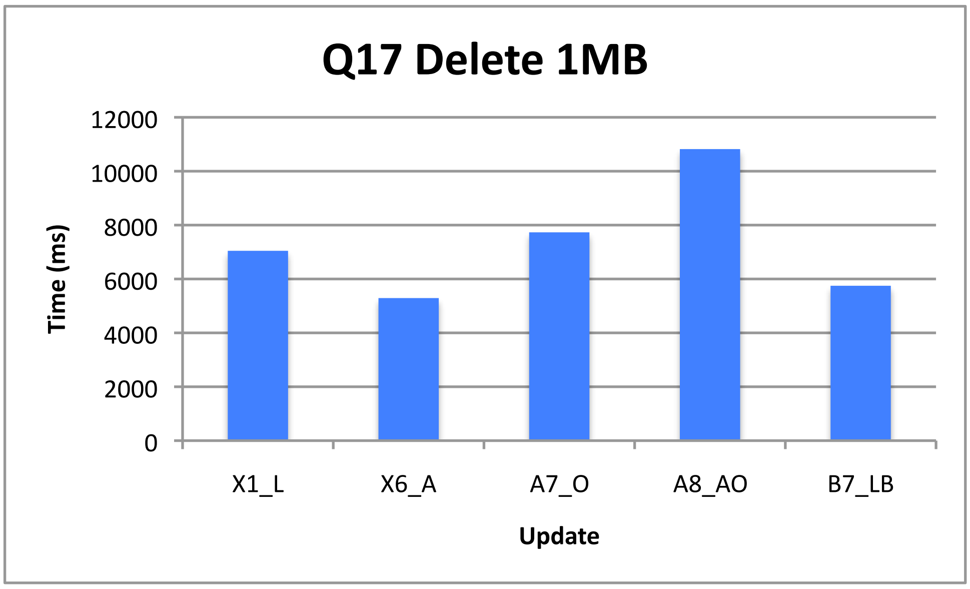
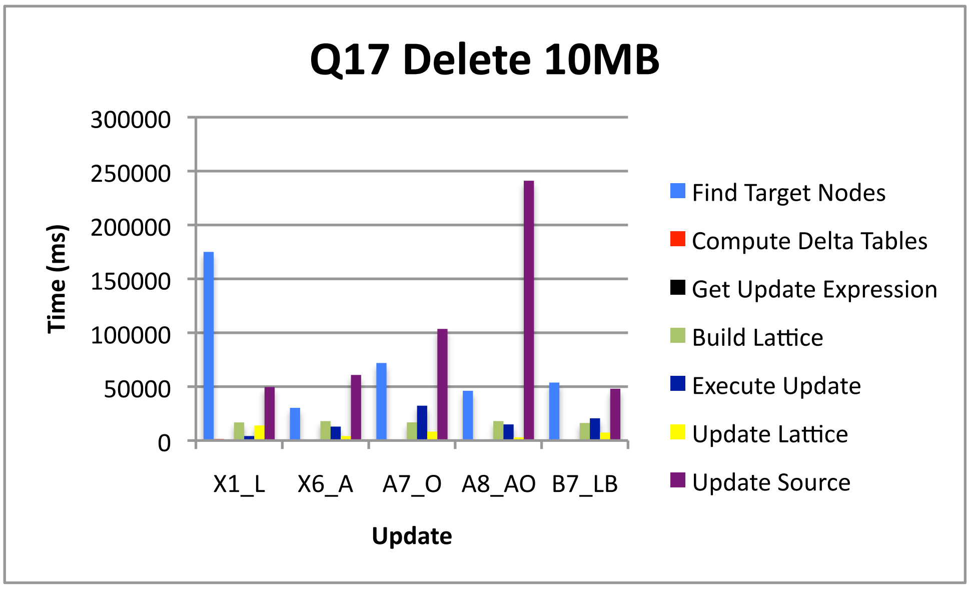


Figure C.55: Q17 Delete Graph 1MB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	175030	1402	36	16799	4189	13989	49674
X6_A	30348	412	17	18037	12948	4175	60856
A7_O	71940	1093	28	16862	32331	8237	103564
A8_AO	46167	516	21	18051	14986	2959	241084
B7_LB	53798	993	23	16269	20649	7397	48073

Table C.56: Q17 Delete 10Mb



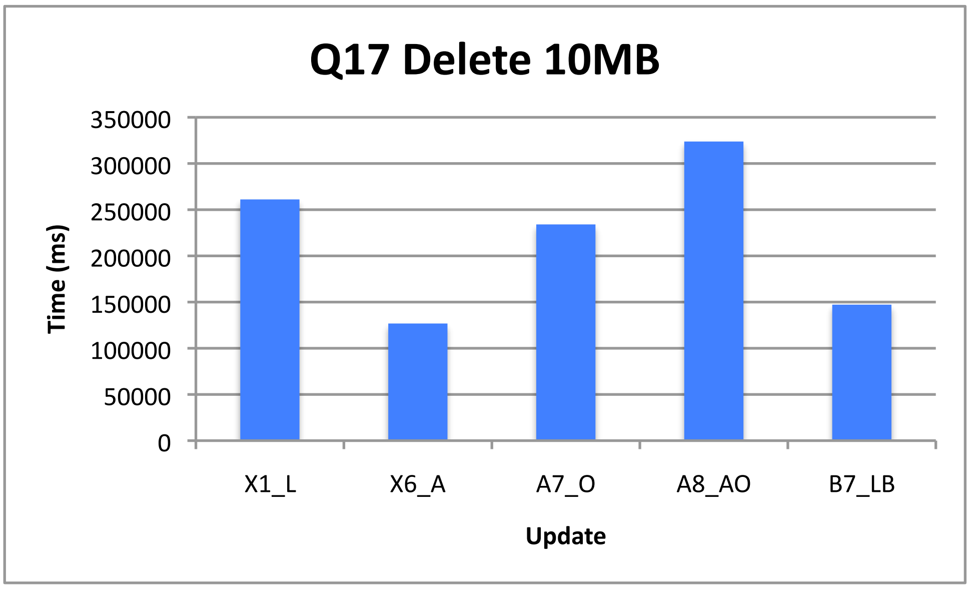


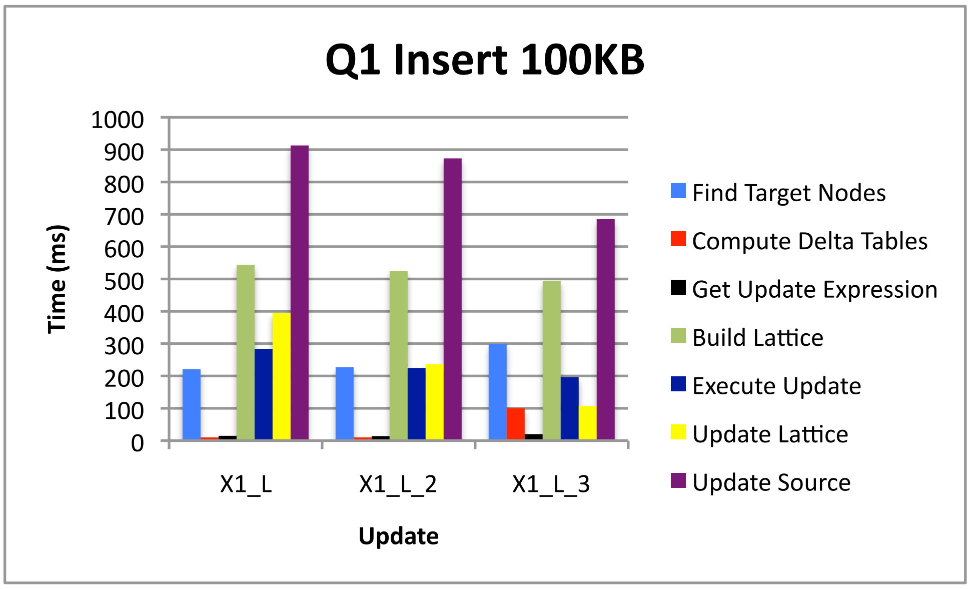
FIGURE C.56: Q17 Delete Graph 10MB

C.2 View Update

C.2.1 Insert

Query	TN	DT	UE	BL	EU	UL	US
X1_L	221	10	15	544	284	394	913
X1_L_2	227	10	14	524	225	236	873
X1_L_3	299	100	20	494	196	107	685

TABLE C.57: Q1 View Insert 100Kb



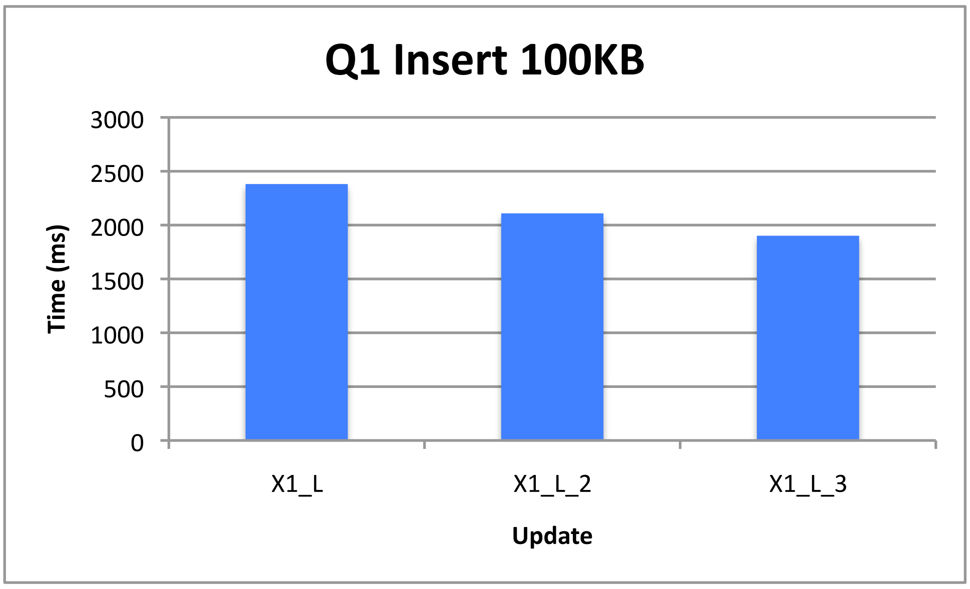
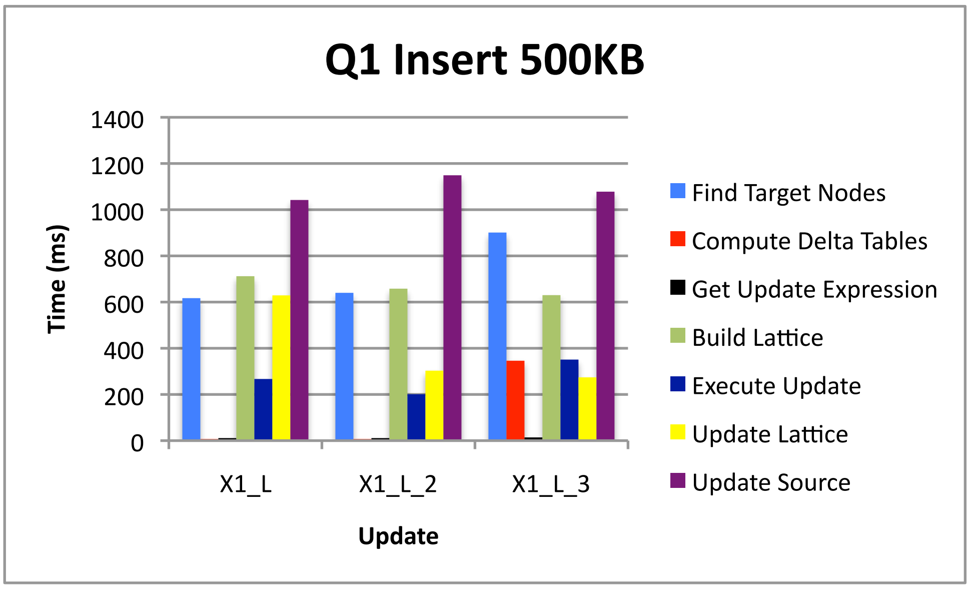


Figure C.57: Q1 Insert Graph 100KB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	617	7	11	712	267	629	1042
X1_L_2	640	7	11	658	201	303	1149
X1_L_3	901	346	14	630	351	274	1078

Table C.58: Q1 View Insert 500Kb



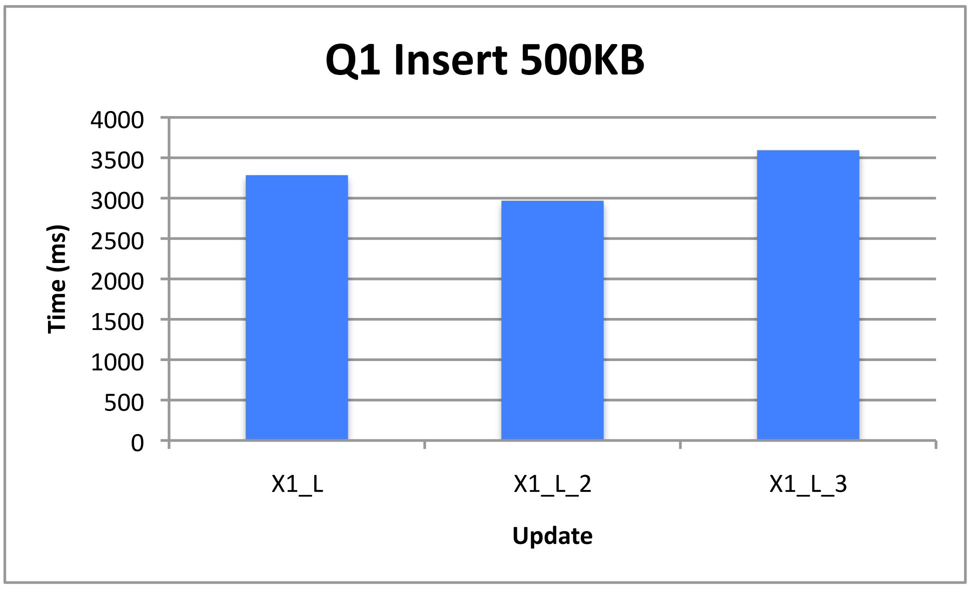
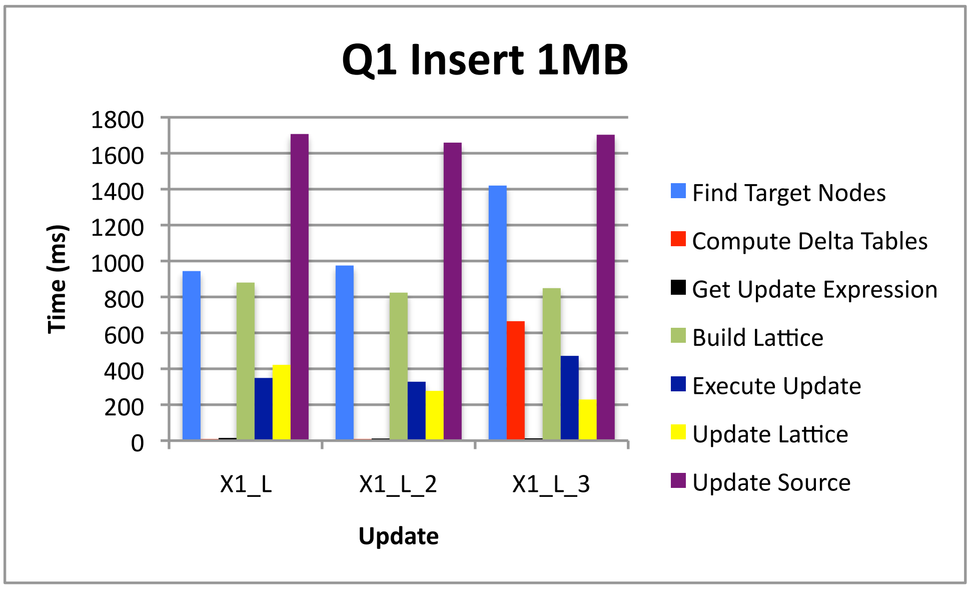


FIGURE C.58: Q1 Insert Graph 500KB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	944	9	15	880	349	422	1707
X1_L_2	975	9	12	824	328	277	1659
X1_L_3	1420	665	13	849	472	229	1703

TABLE C.59: Q1 View Insert 1Mb



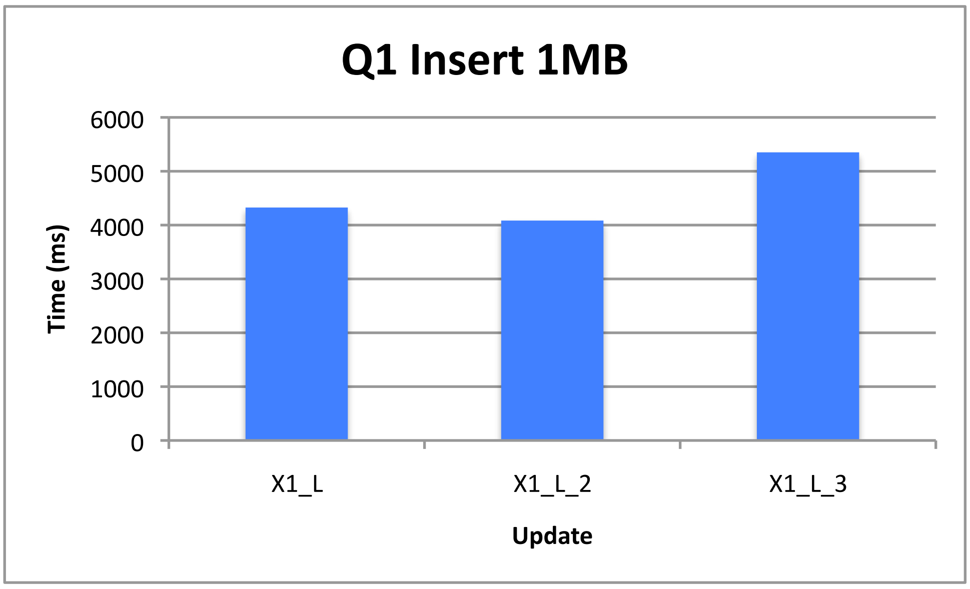
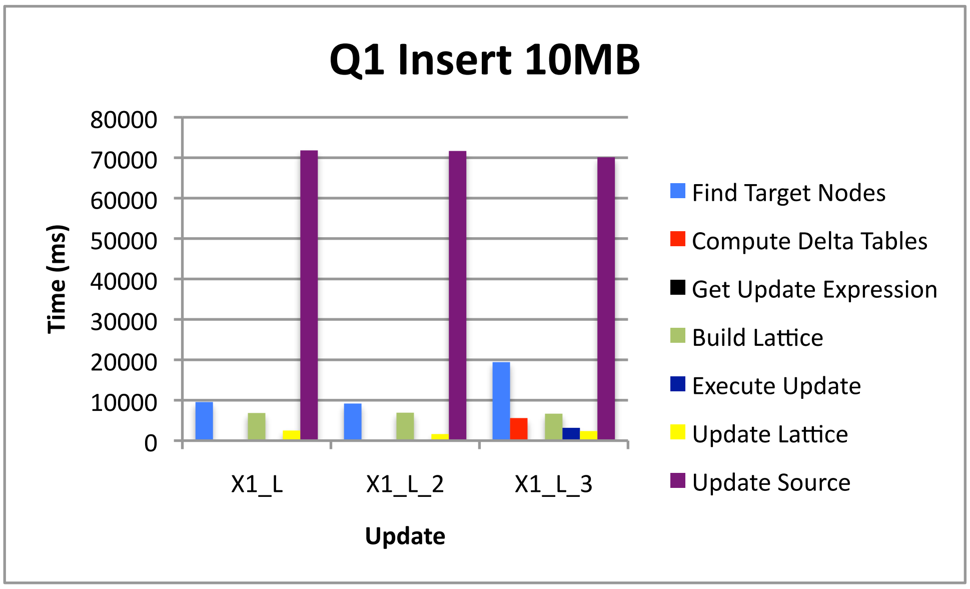


Figure C.59: Q1 Insert Graph 1MB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	9554	9	10	6838	225	2505	71817
X1_L_2	9181	9	10	6916	315	1626	71672
X1_L_3	19428	5588	15	6670	3172	2366	70111

Table C.60: Q1 View Insert 10Mb



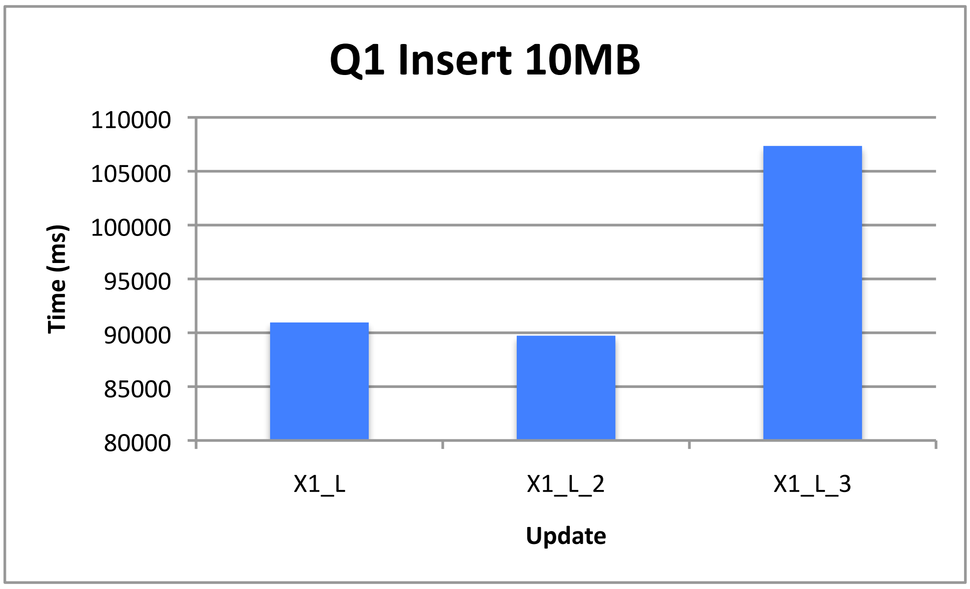
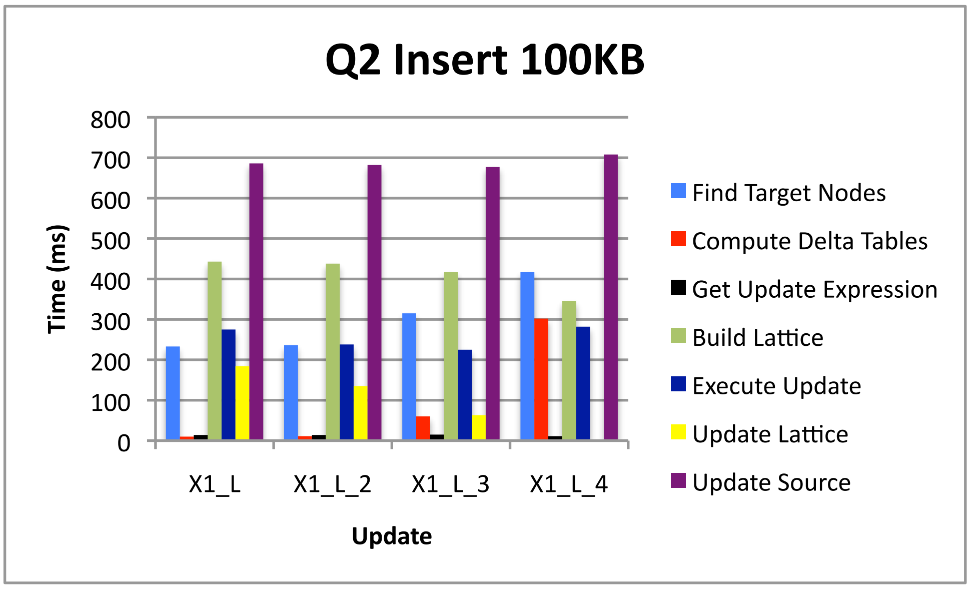


FIGURE C.60: Q1 Insert Graph 10MB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	233	10	14	443	275	184	686
X1_L_2	236	11	14	438	238	135	682
X1_L_3	315	60	15	417	225	63	677
X1_L_4	417	302	11	346	282	0	708

TABLE C.61: Q2 View Insert 100Kb



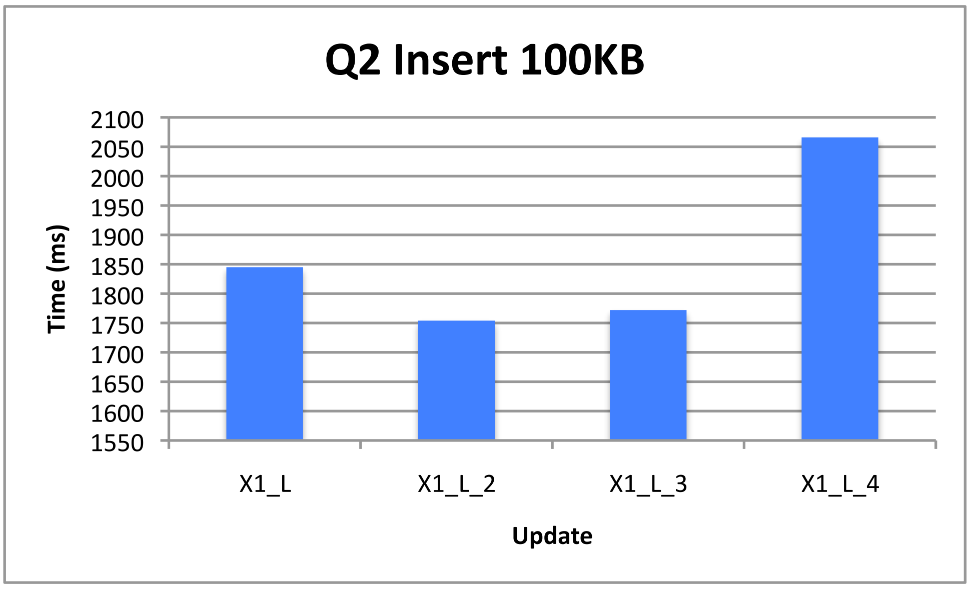
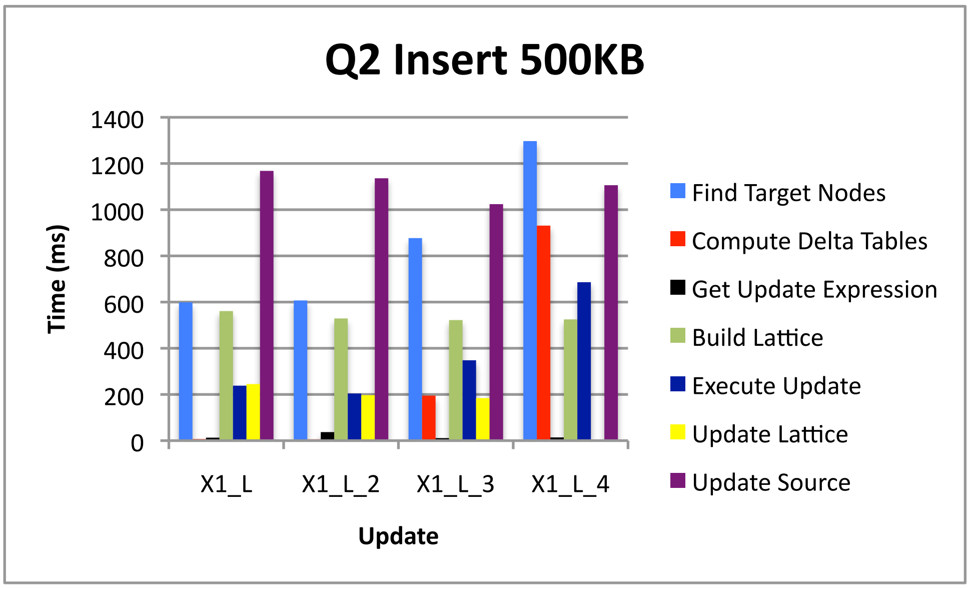


Figure C.61: Q2 Insert Graph 100KB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	599	7	13	561	238	245	1168
X1_L_2	607	6	37	529	204	197	1136
X1_L_3	877	195	11	522	348	184	1024
X1_L_4	1297	931	14	525	686	0	1106

Table C.62: Q2 View Insert 500Kb



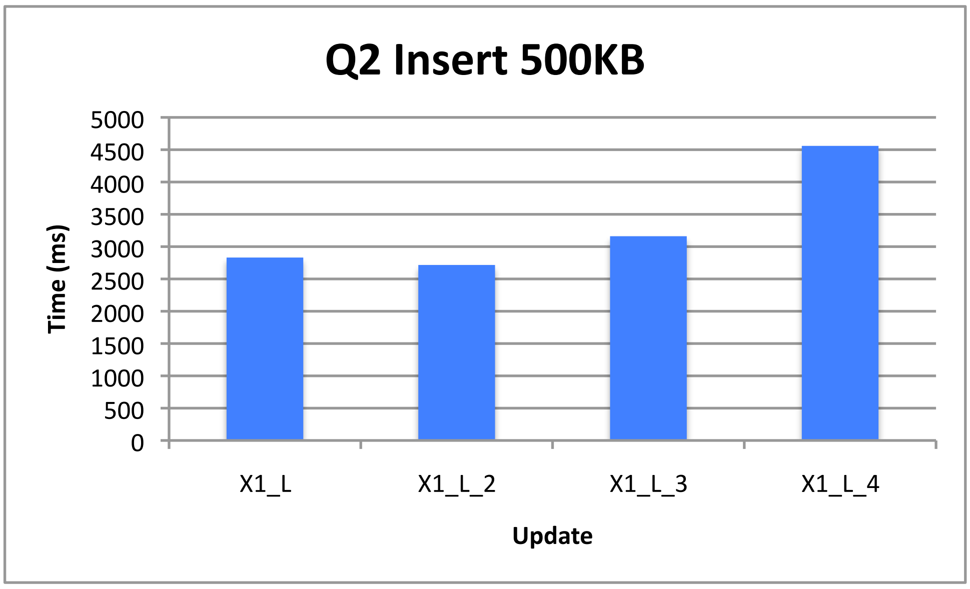
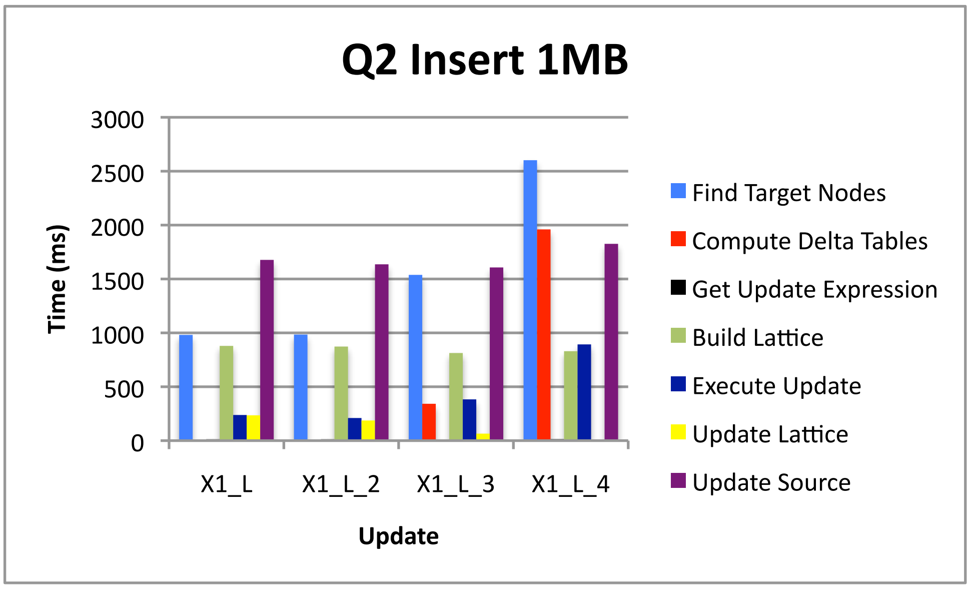


FIGURE C.62: Q2 Insert Graph 500KB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	981	9	13	879	238	235	1677
X1_L_2	984	9	13	873	210	187	1636
X1_L_3	1538	342	11	813	383	65	1607
X1_L_4	2602	1960	14	830	893	0	1826

TABLE C.63: Q2 View Insert 1Mb



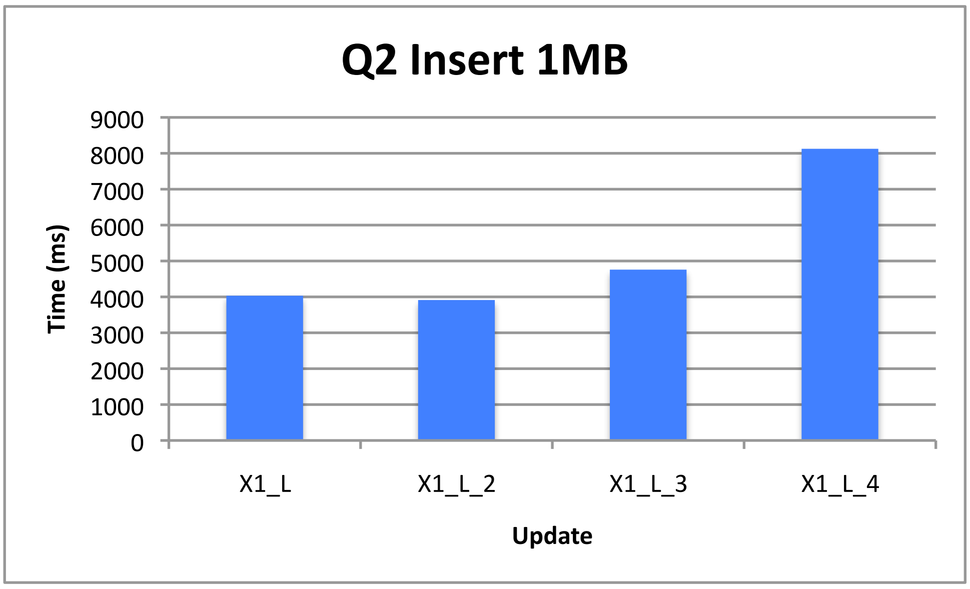
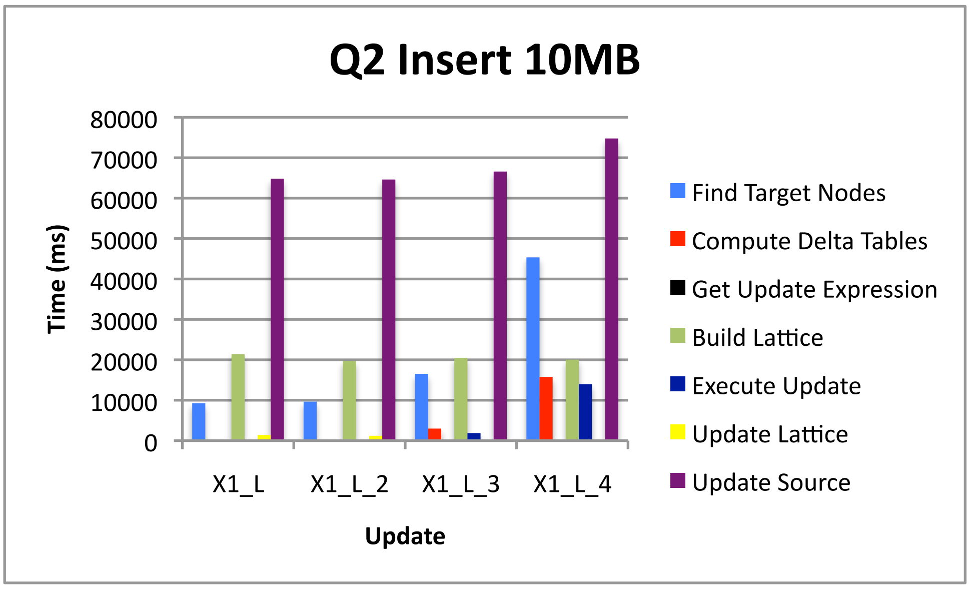


Figure C.63: Q2 Insert Graph 1MB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	9244	9	10	21399	227	1406	64817
X1_L_2	9645	8	11	19696	192	1201	64609
X1_L_3	16538	2985	16	20454	1879	311	66583
X1_L_4	45347	15770	16	20035	13958	0	74767

Table C.64: Q2 View Insert 10Mb



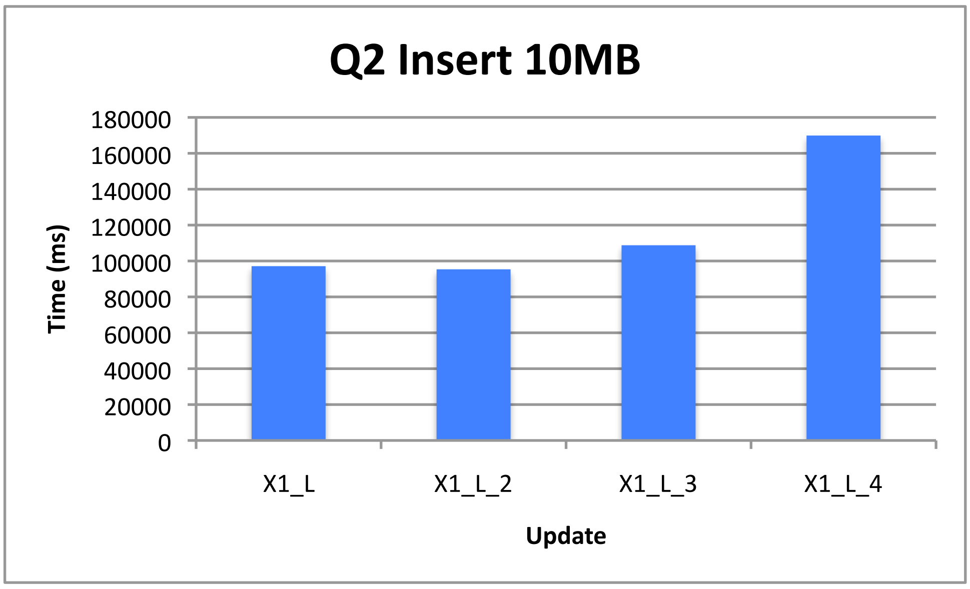
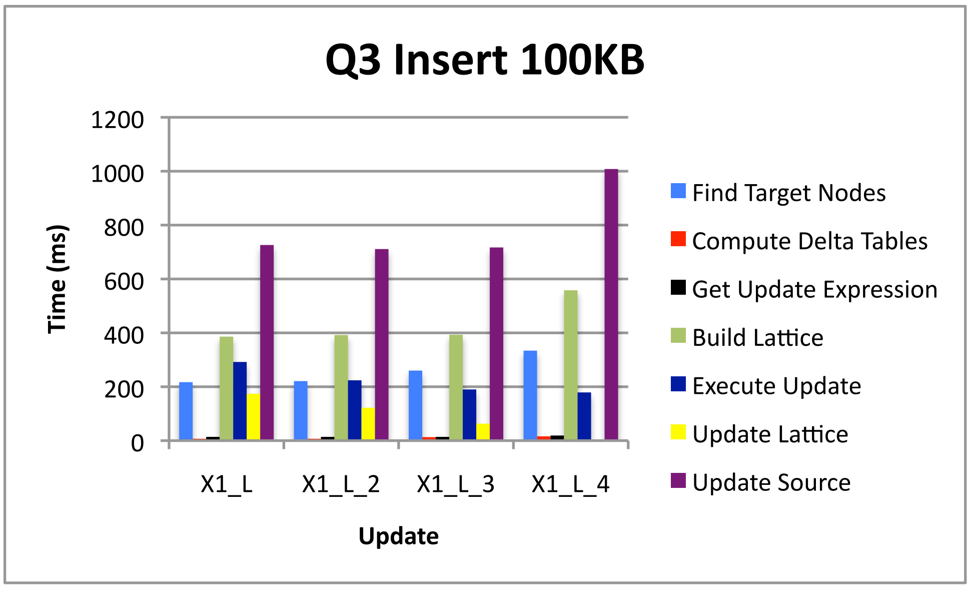


FIGURE C.64: Q2 Insert Graph 10MB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	217	7	14	386	292	174	726
X1_L_2	221	7	14	392	224	122	711
X1_L_3	260	13	14	393	190	63	717
X1_L_4	334	16	19	558	179	0	1008

TABLE C.65: Q3 View Insert 100Kb



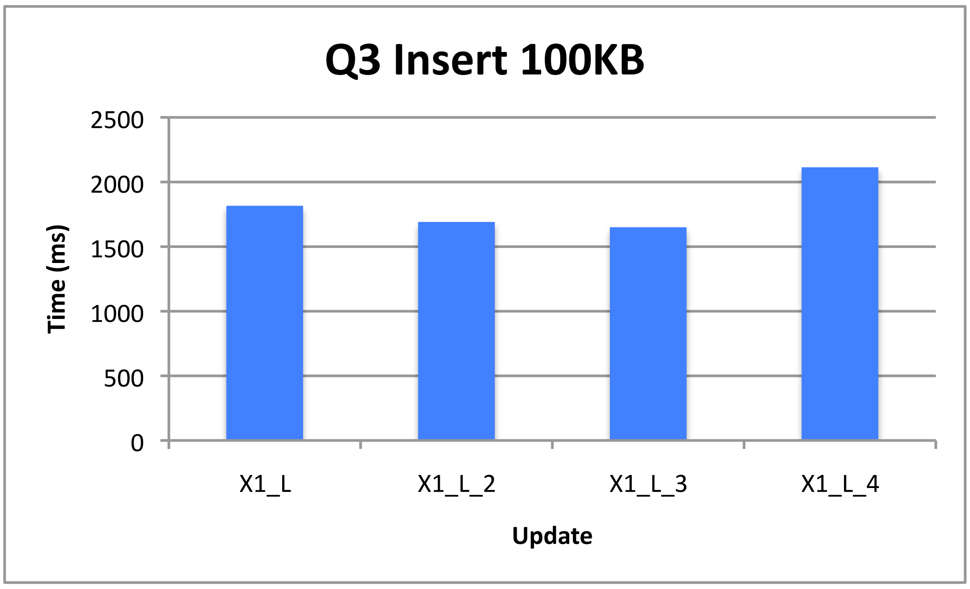
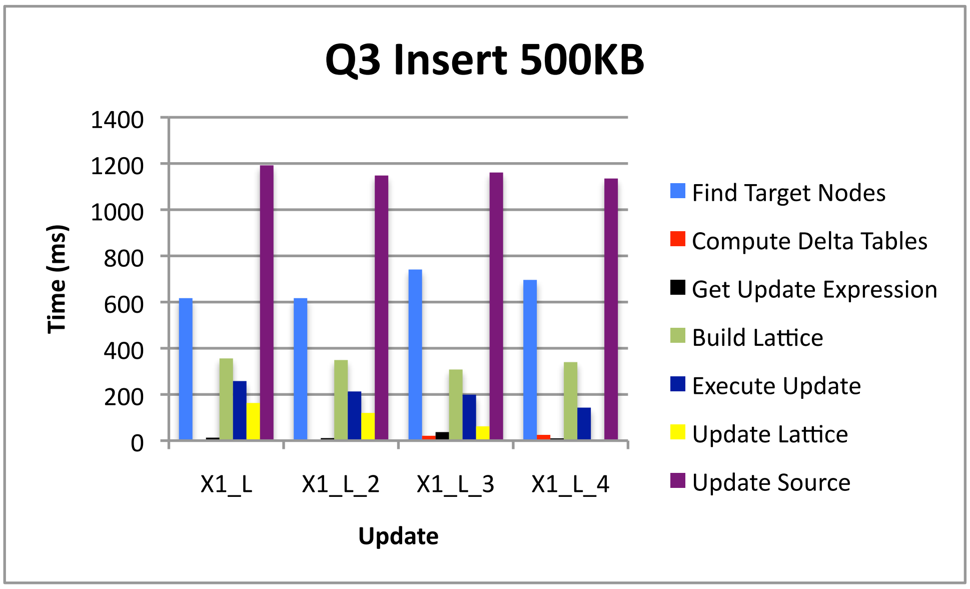


Figure C.65: Q3 Insert Graph 100KB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	617	4	13	356	258	163	1192
X1_L_2	617	4	11	349	213	120	1148
X1_L_3	741	21	37	308	199	62	1161
X1_L_4	696	25	10	340	143	0	1135

Table C.66: Q3 View Insert 500Kb



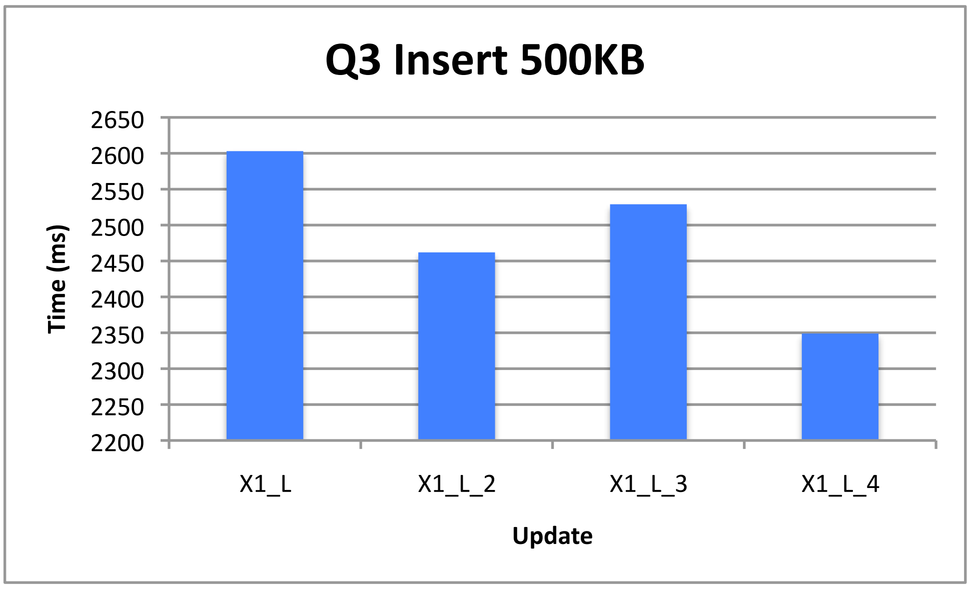
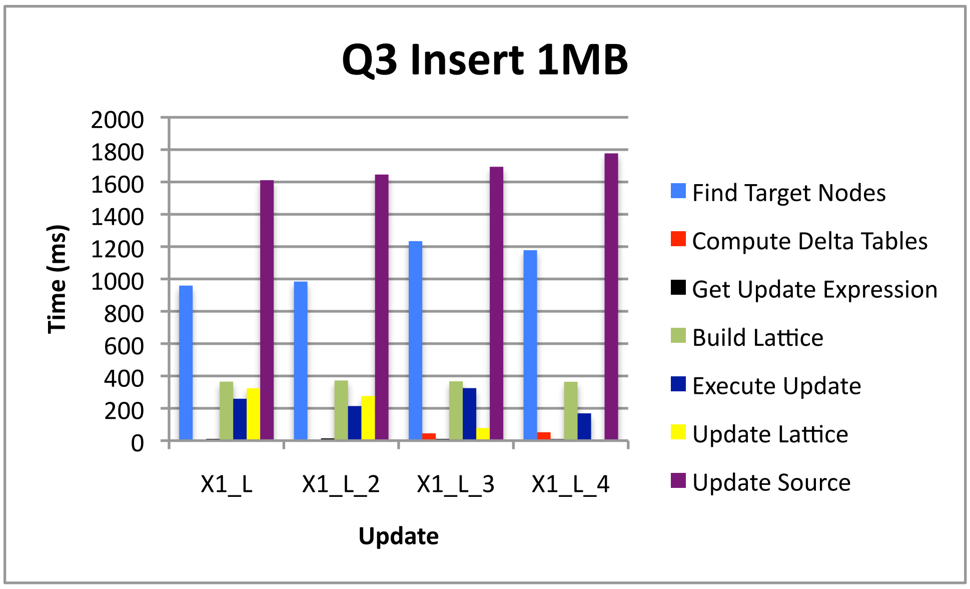


FIGURE C.66: Q3 Insert Graph 500KB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	959	6	11	365	259	324	1611
X1_L_2	984	7	14	372	214	276	1646
X1_L_3	1234	45	11	367	325	78	1694
X1_L_4	1178	52	10	364	169	0	1777

TABLE C.67: Q3 View Insert 1Mb



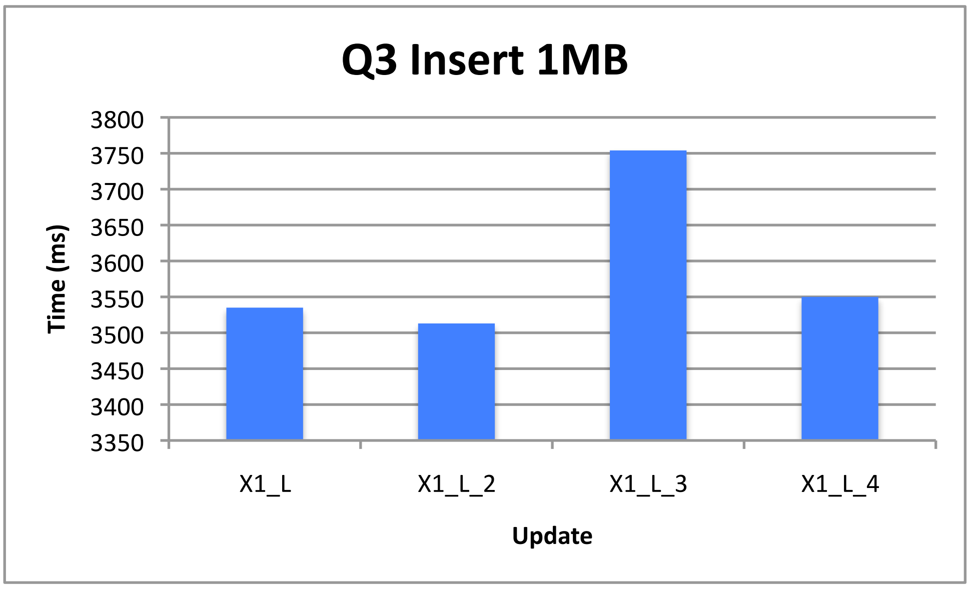
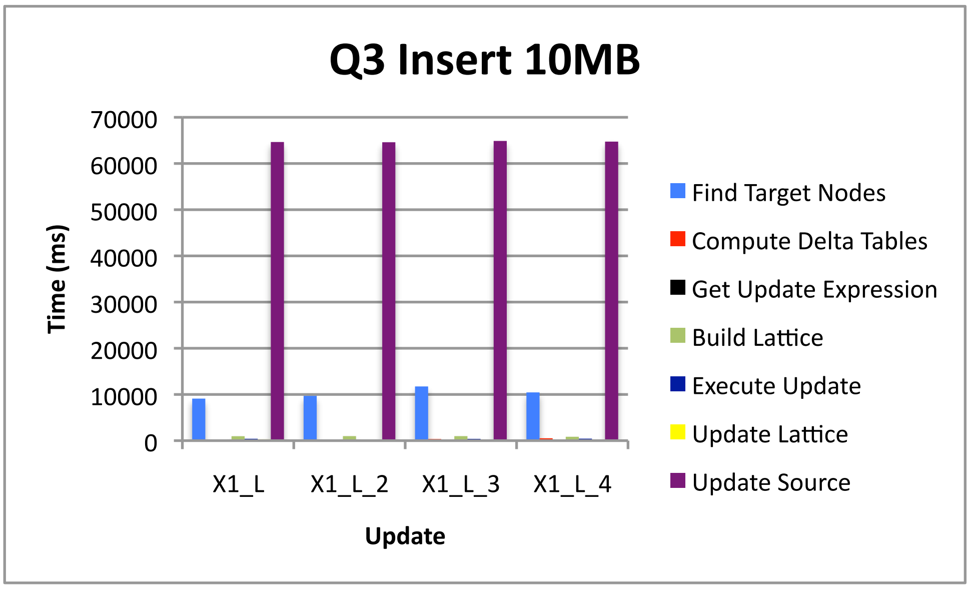


Figure C.67: Q3 Insert Graph 1MB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	9100	7	11	959	406	287	64651
X1_L_2	9694	6	10	979	202	197	64602
X1_L_3	11734	350	10	976	386	123	64883
X1_L_4	10445	513	13	840	429	0	64739

Table C.68: Q3 View Insert 10Mb



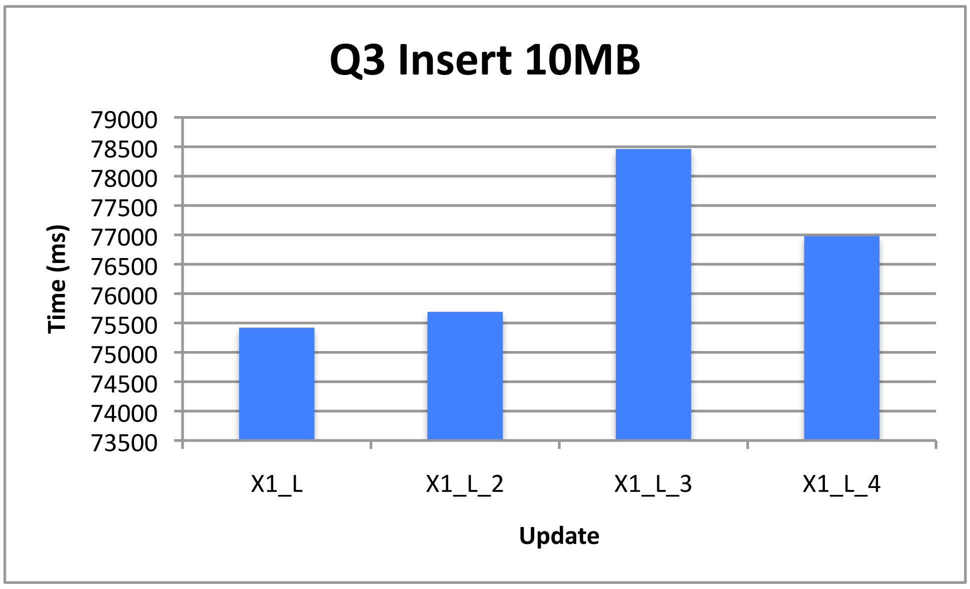
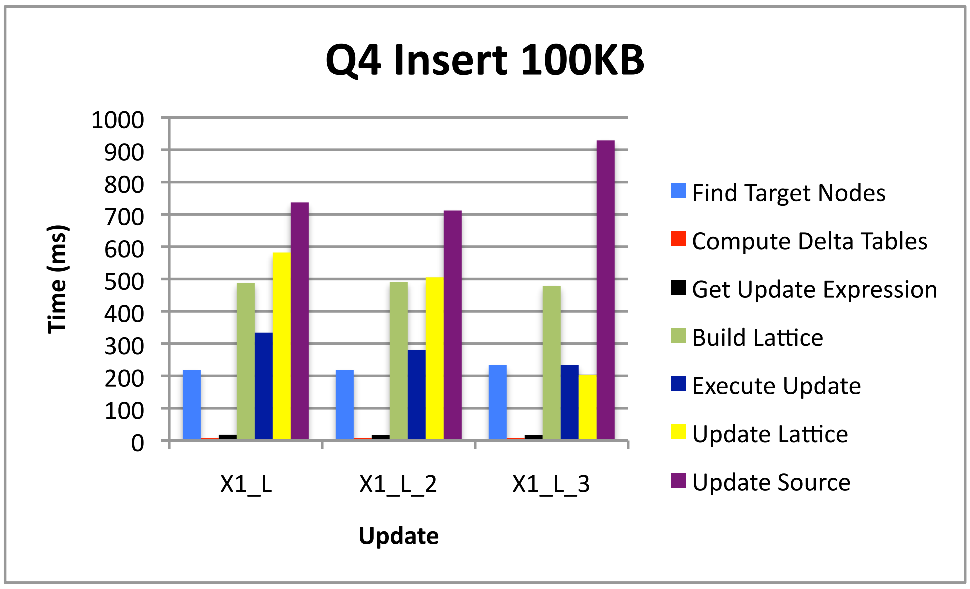


FIGURE C.68: Q3 Insert Graph 10MB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	218	7	18	488	334	582	737
X1_L_2	218	8	17	491	281	505	712
X1_L_3	233	8	17	479	234	202	929

TABLE C.69: Q4 View Insert 100Kb



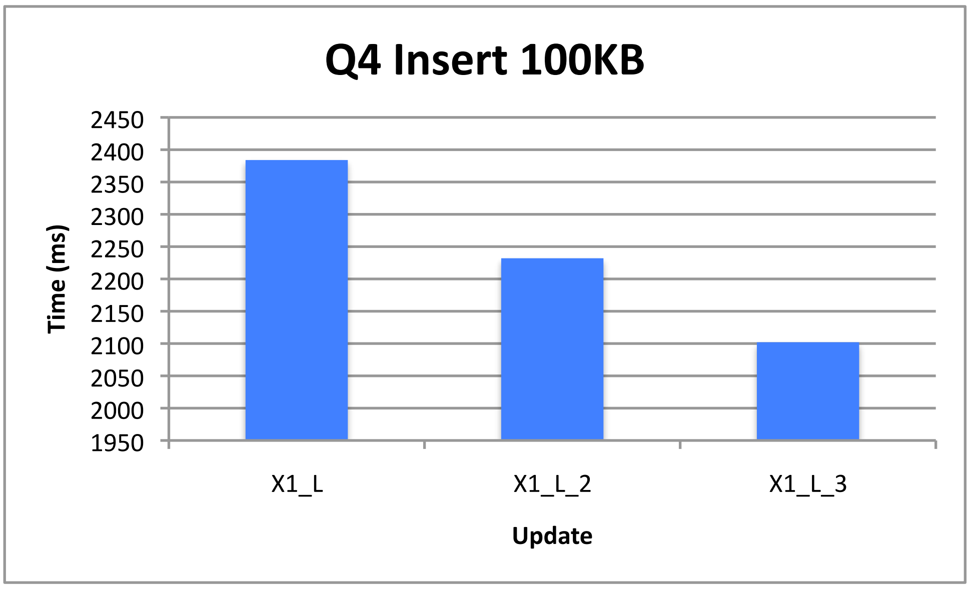
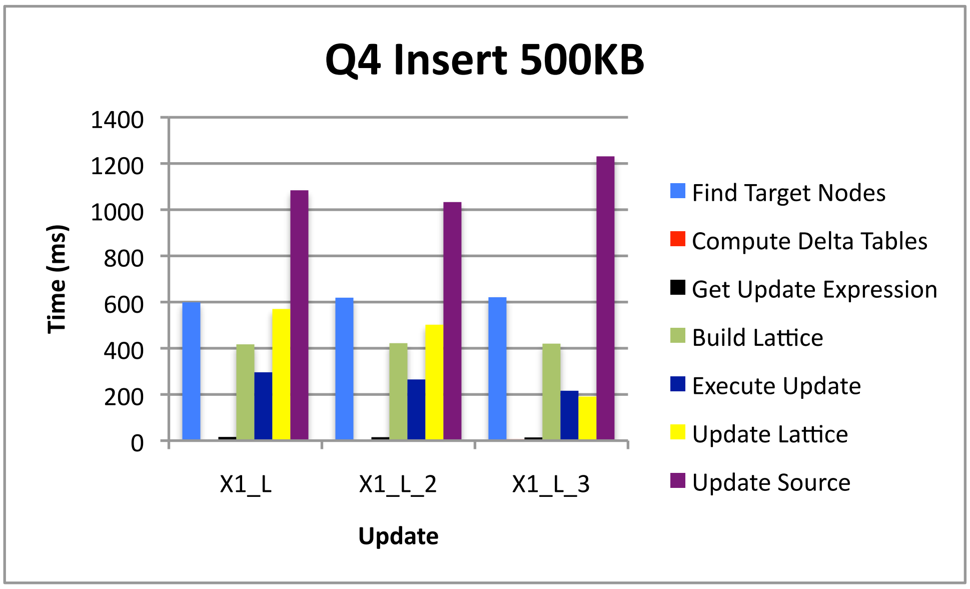


Figure C.69: Q4 Insert Graph 100KB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	599	4	16	417	296	570	1084
X1_L_2	619	4	15	422	265	502	1033
X1_L_3	621	6	14	420	216	191	1231

Table C.70: Q4 View Insert 500Kb



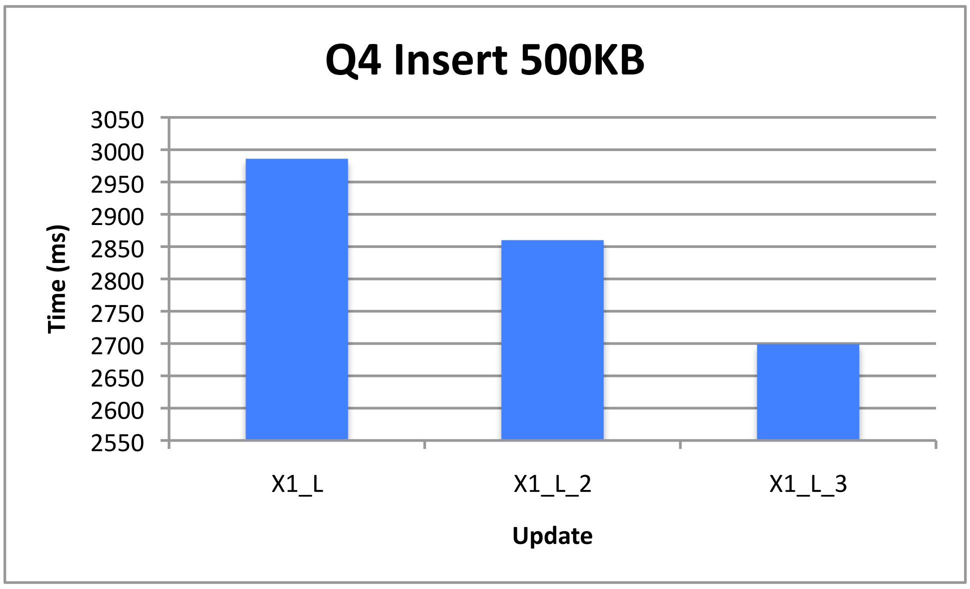
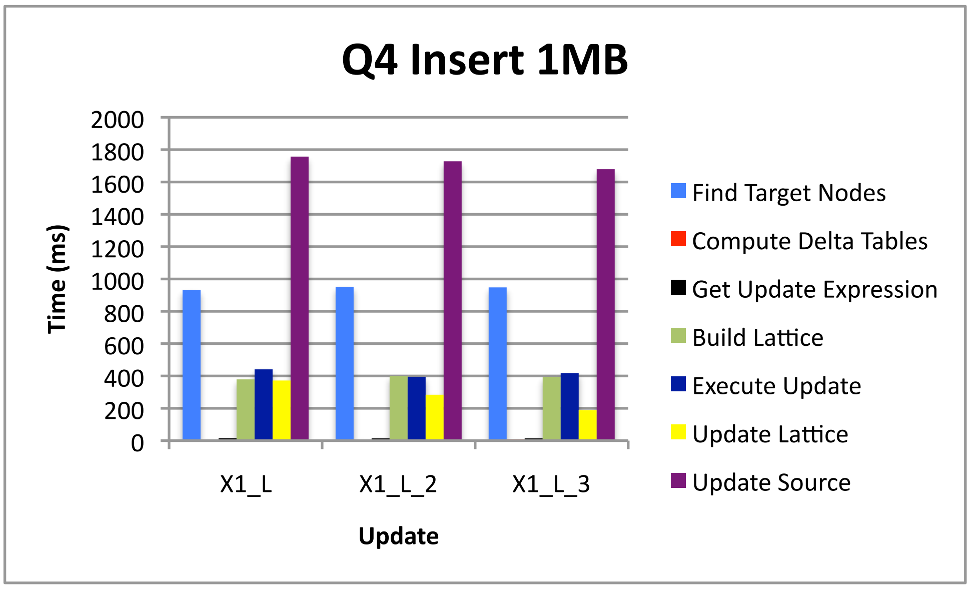


FIGURE C.70: Q4 Insert Graph 500KB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	932	6	15	379	441	372	1757
X1_L_2	952	7	14	400	395	284	1728
X1_L_3	948	9	14	395	418	189	1679

TABLE C.71: Q4 View Insert 1Mb



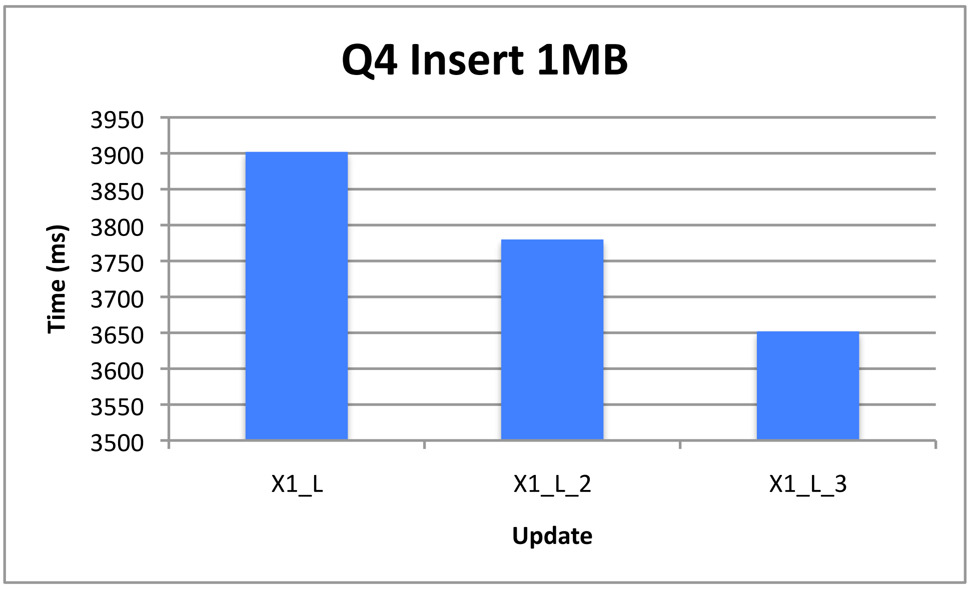
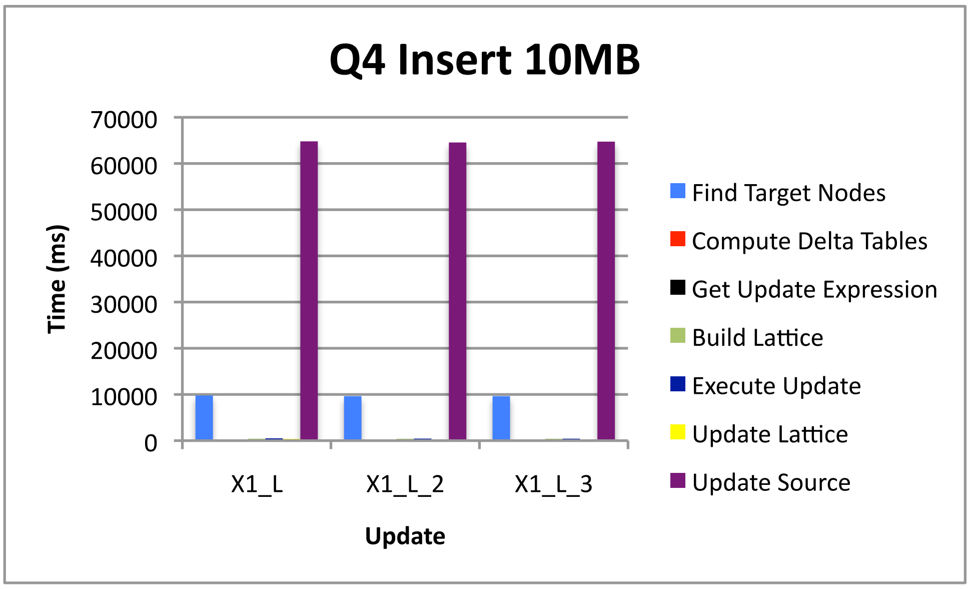


Figure C.71: Q4 Insert Graph 1MB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	9769	7	14	410	501	363	64797
X1_L_2	9635	6	14	401	439	279	64549
X1_L_3	9632	10	12	405	420	194	64732

Table C.72: Q4 View Insert 10Mb



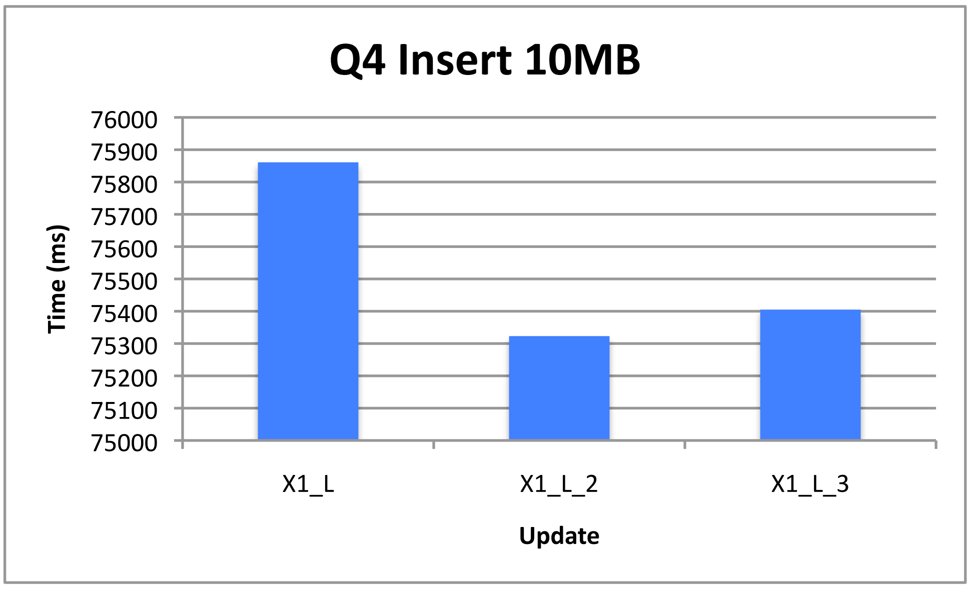
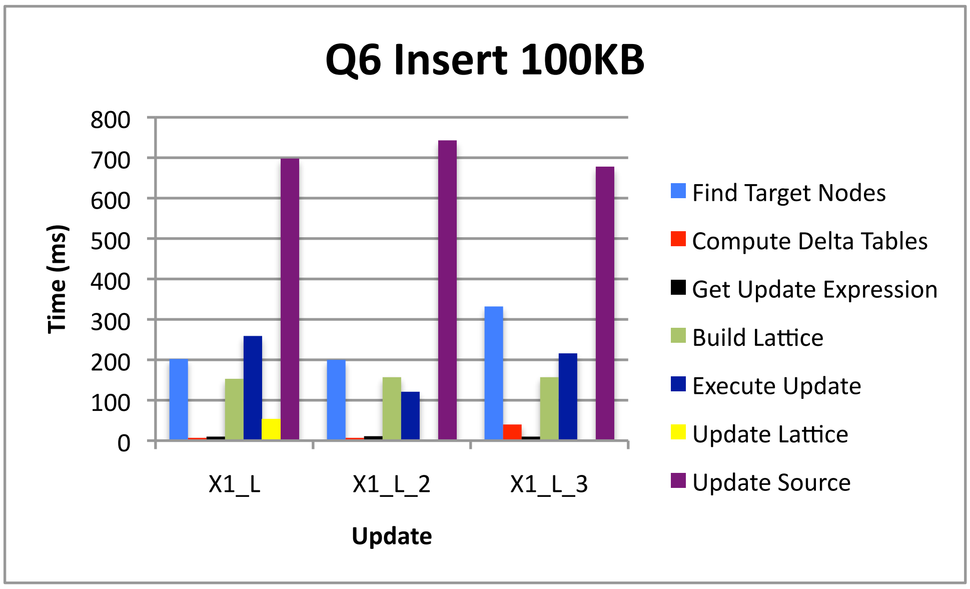


FIGURE C.72: Q4 Insert Graph 10MB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	202	7	10	153	259	54	698
X1_L_2	200	7	11	157	121	0	743
X1_L_3	332	40	10	157	216	0	678

TABLE C.73: Q6 View Insert 100Kb



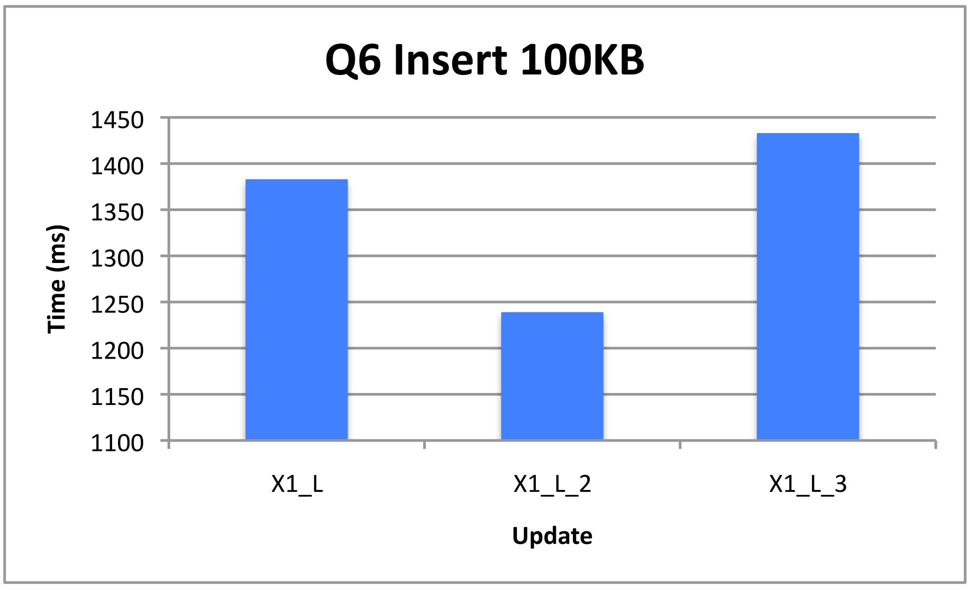
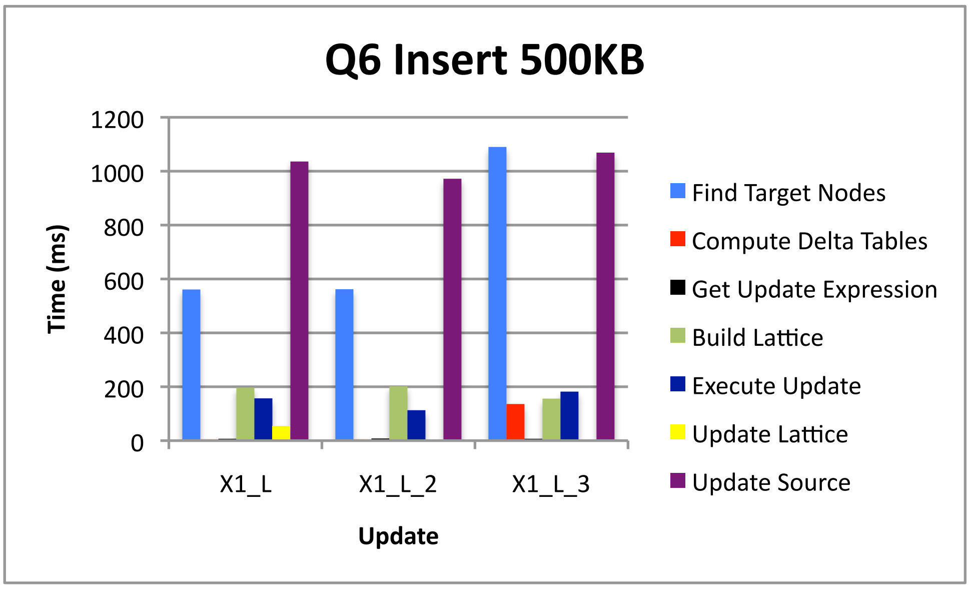


Figure C.73: Q6 Insert Graph 100KB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	561	5	7	197	157	54	1036
X1_L_2	562	5	8	202	113	0	972
X1_L_3	1090	136	7	156	182	0	1069

Table C.74: Q6 View Insert 500Kb



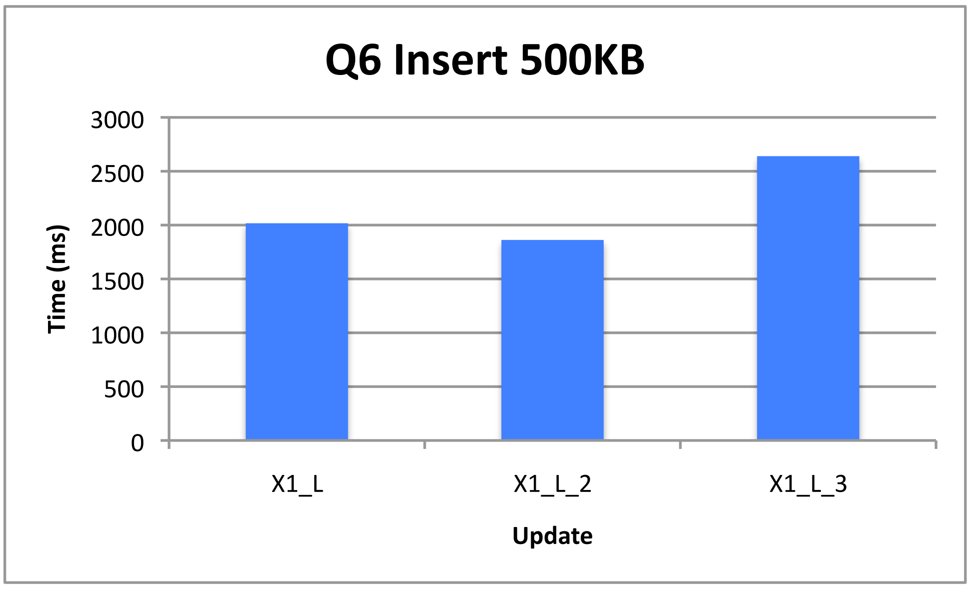
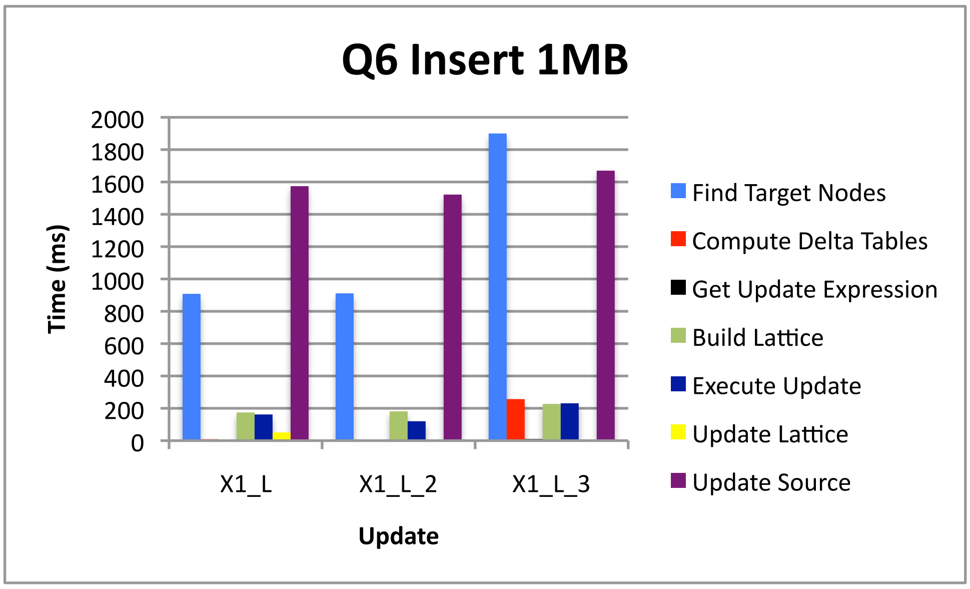


FIGURE C.74: Q6 Insert Graph 500KB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	908	9	6	174	162	51	1574
X1_L_2	911	8	7	181	120	0	1522
X1_L_3	1900	257	10	227	231	0	1670

TABLE C.75: Q6 View Insert 1Mb



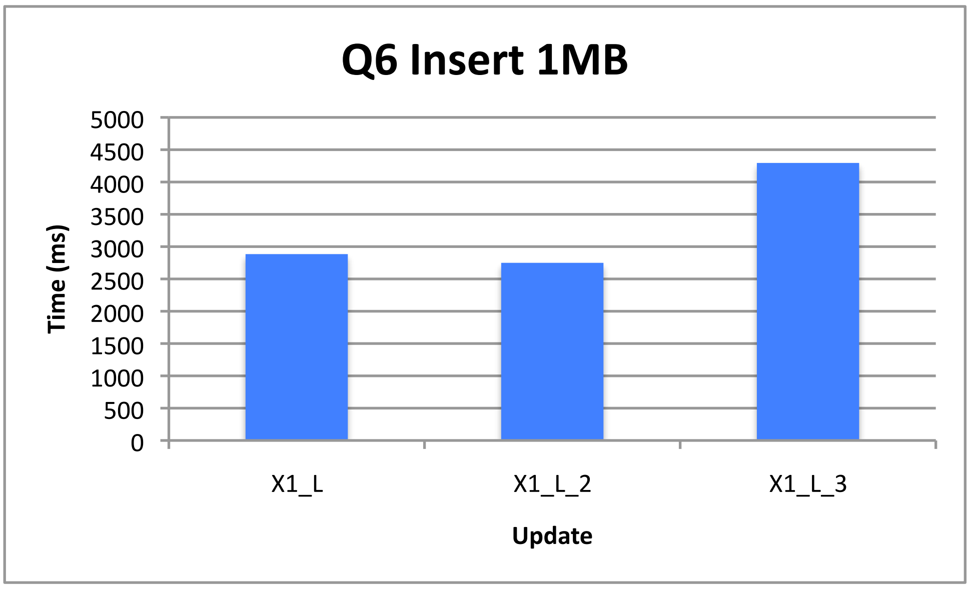
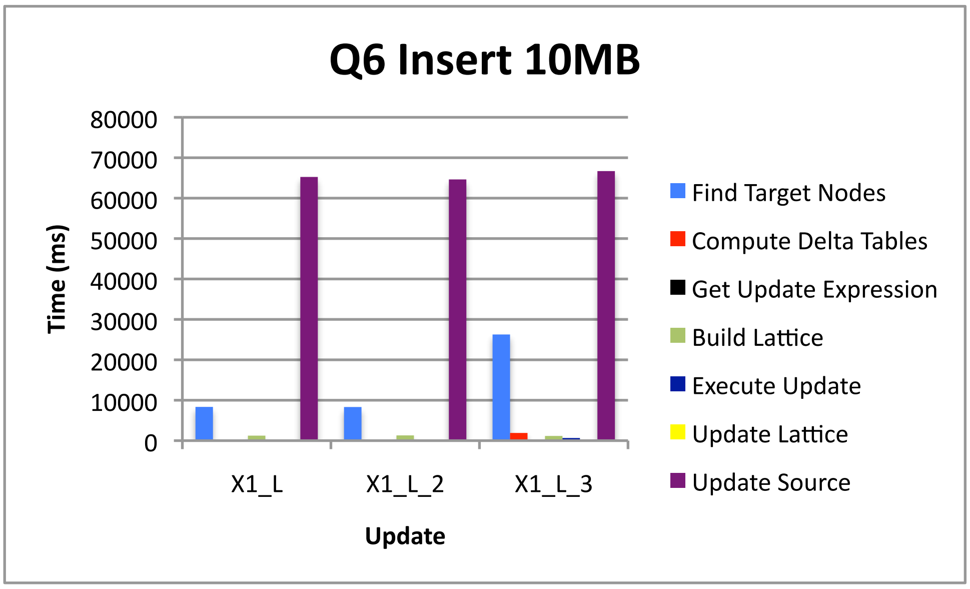


Figure C.75: Q6 Insert Graph 1MB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	8346	7	7	1235	144	38	65242
X1_L_2	8319	7	7	1297	111	0	64630
X1_L_3	26266	1901	11	1156	650	0	66692

Table C.76: Q6 View Insert 10Mb



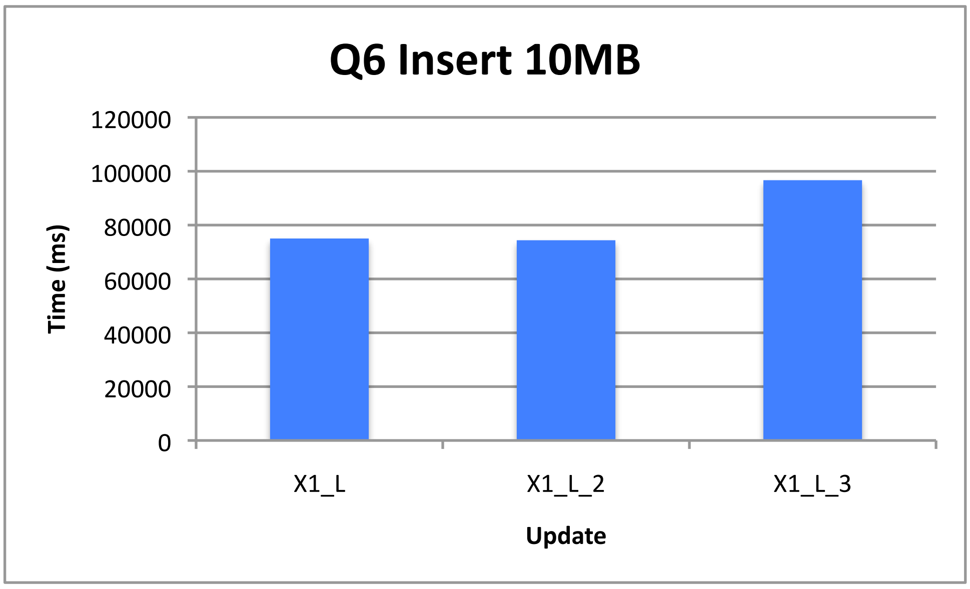
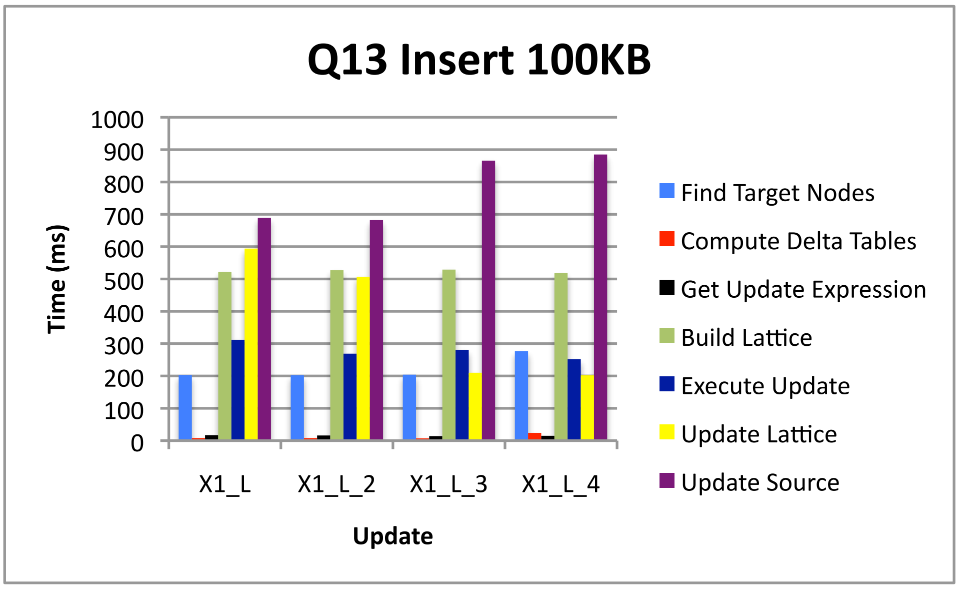


FIGURE C.76: Q6 Insert Graph 10MB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	203	8	17	522	312	594	689
X1_L_2	202	8	16	527	269	507	682
X1_L_3	204	7	14	529	281	210	866
X1_L_4	277	24	15	518	252	202	885

TABLE C.77: Q13 View Insert 100Kb



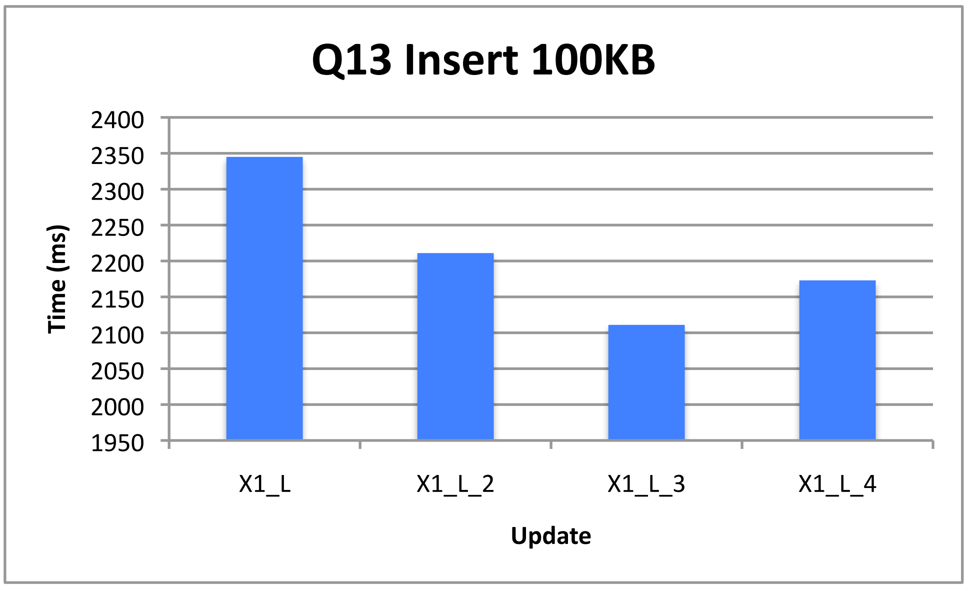
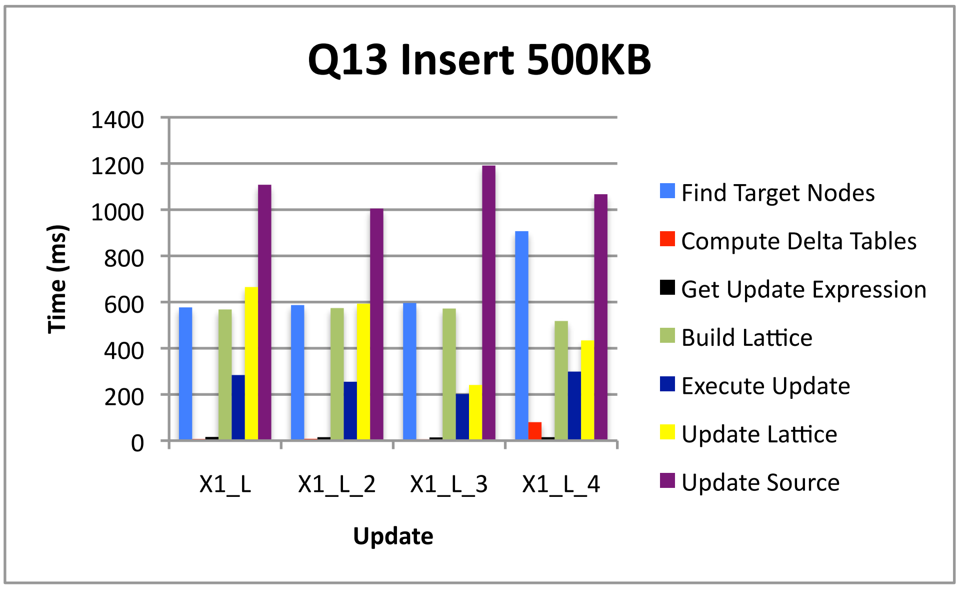


Figure C.77: Q13 Insert Graph 100KB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	577	7	16	568	284	665	1108
X1_L_2	587	8	15	574	255	593	1005
X1_L_3	596	6	14	572	202	241	1191
X1_L_4	907	80	15	518	299	434	1067

Table C.78: Q13 View Insert 500Kb



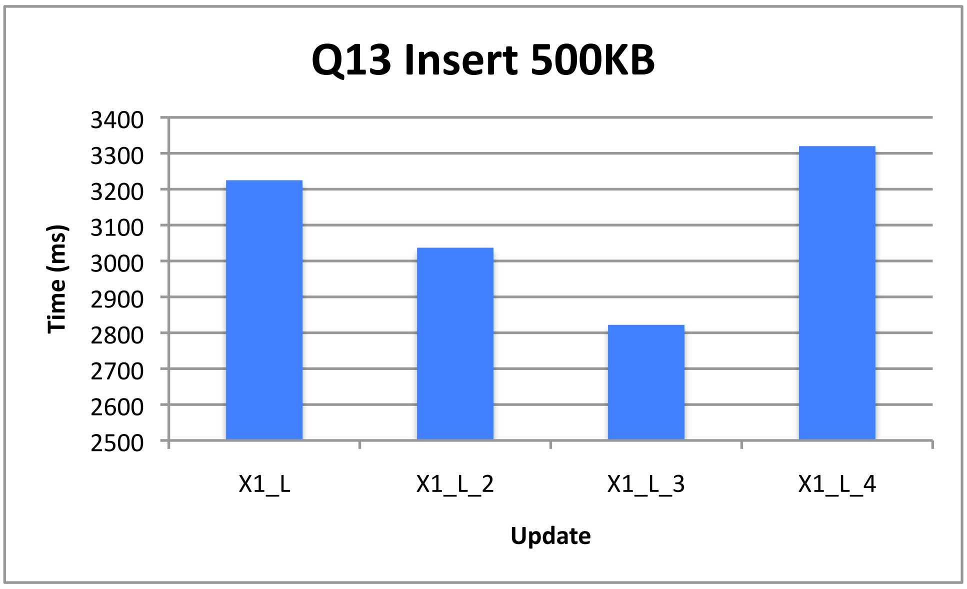
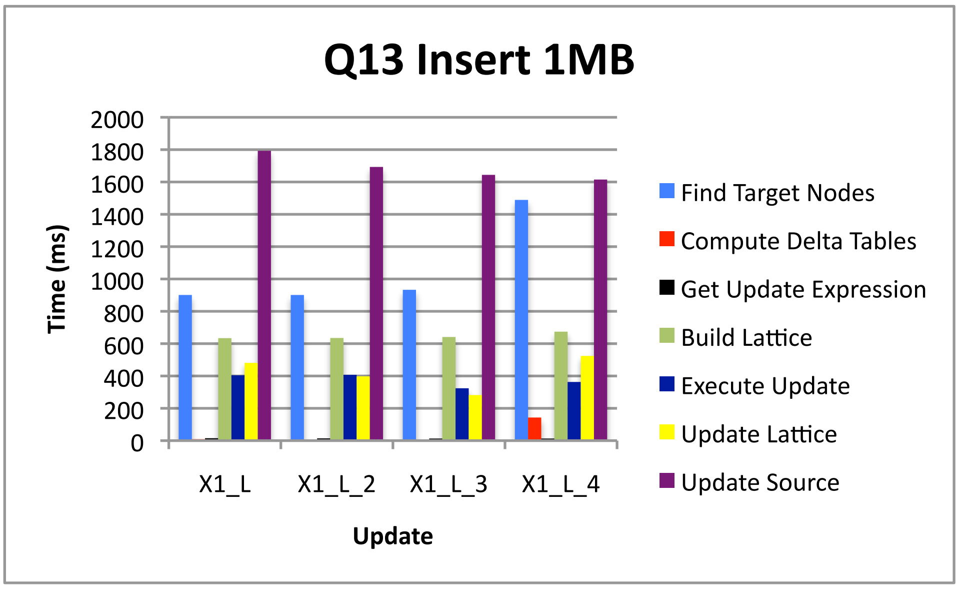


FIGURE C.78: Q13 Insert Graph 500KB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	901	9	15	634	404	481	1793
X1_L_2	901	7	14	635	407	398	1693
X1_L_3	933	7	13	641	324	282	1644
X1_L_4	1489	143	13	674	363	524	1615

TABLE C.79: Q13 View Insert 1Mb



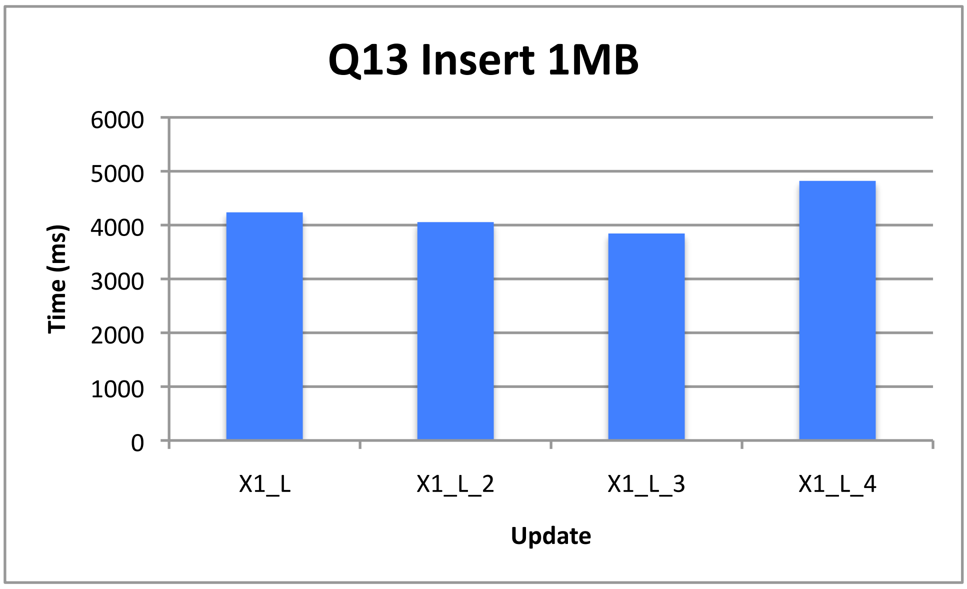
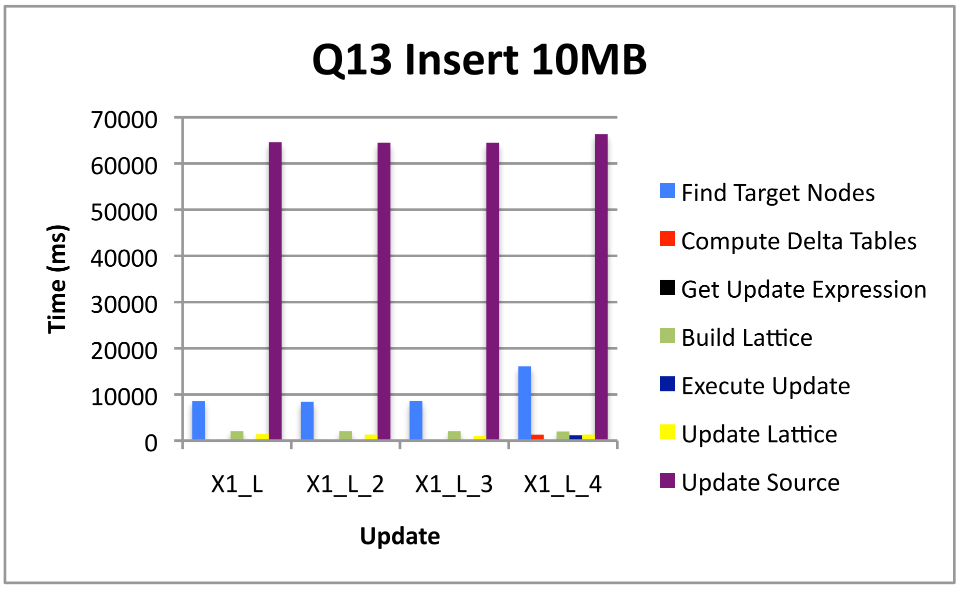


Figure C.79: Q13 Insert Graph 1MB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	8576	7	13	2064	261	1427	64602
X1_L_2	8423	6	13	2075	226	1305	64513
X1_L_3	8606	7	12	2052	185	1002	64502
X1_L_4	16072	1293	17	1974	1149	1312	66344

Table C.80: Q13 View Insert 10Mb



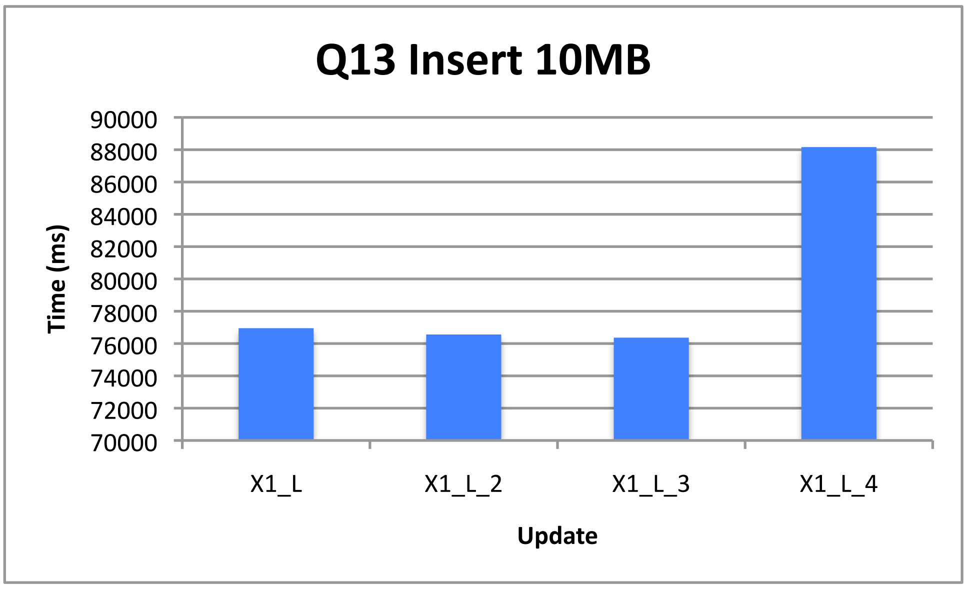
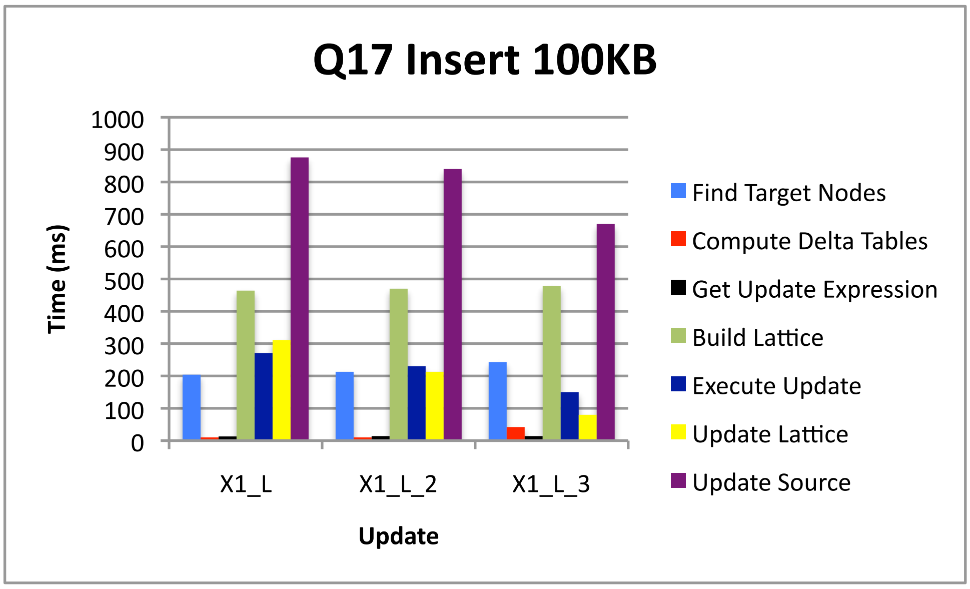


FIGURE C.80: Q13 Insert Graph 10MB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	204	10	13	464	271	311	876
X1_L_2	213	10	14	470	230	213	840
X1_L_3	243	42	14	478	150	80	670

TABLE C.81: Q17 View Insert 100Kb



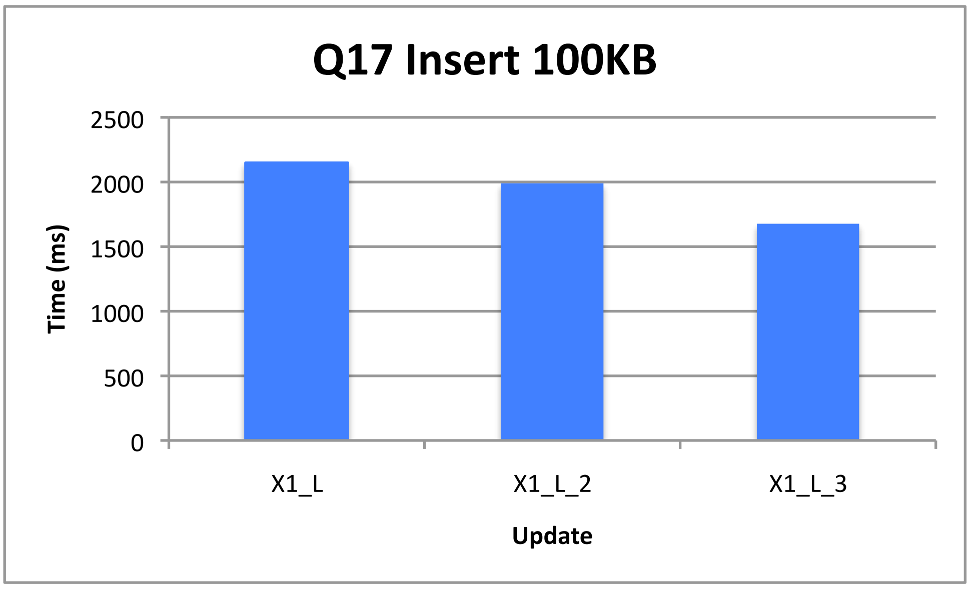
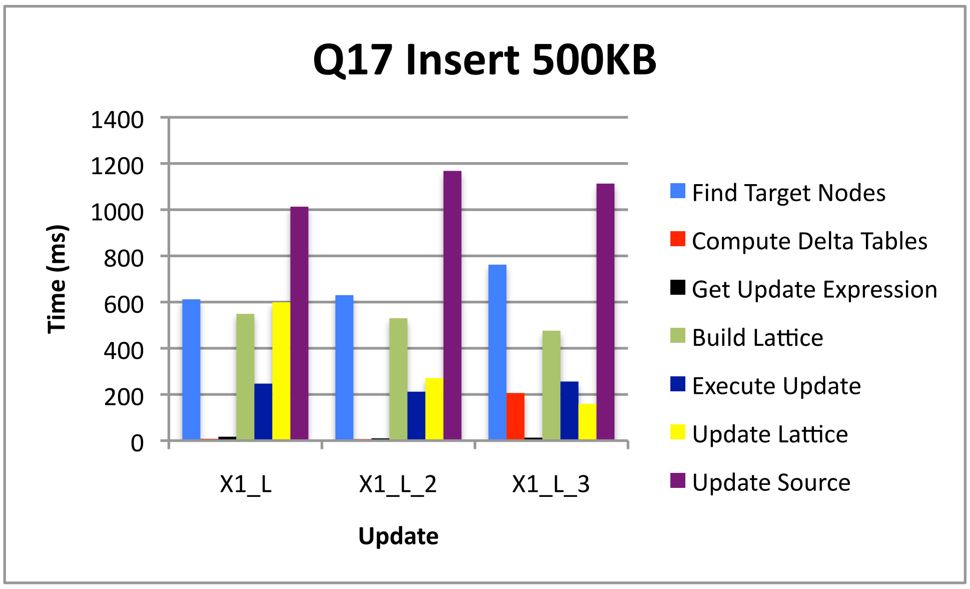


Figure C.81: Q17 Insert Graph 100KB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	612	8	17	549	247	599	1013
X1_L_2	630	7	10	530	212	271	1168
X1_L_3	762	206	13	476	256	159	1113

Table C.82: Q17 View Insert 500Kb



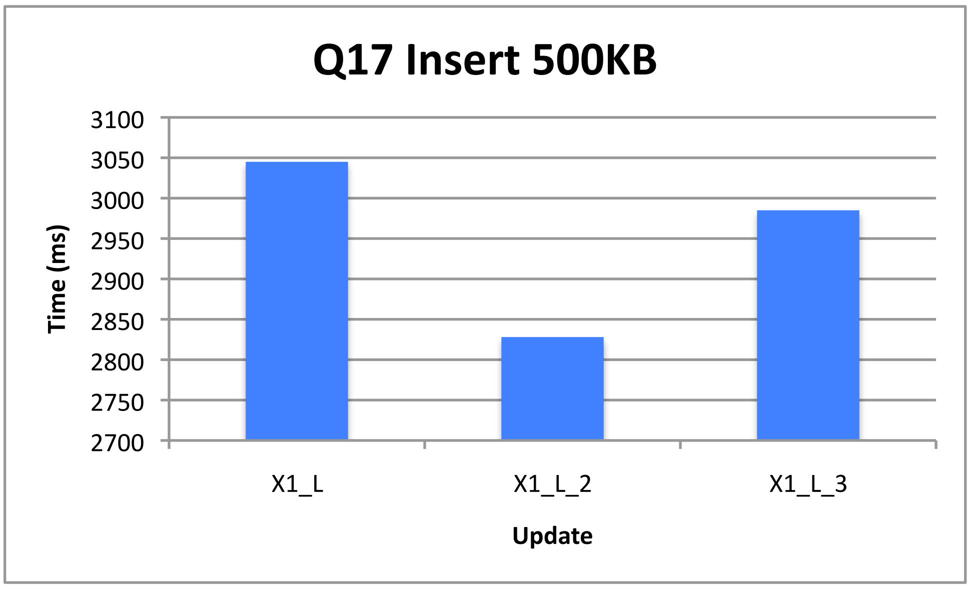
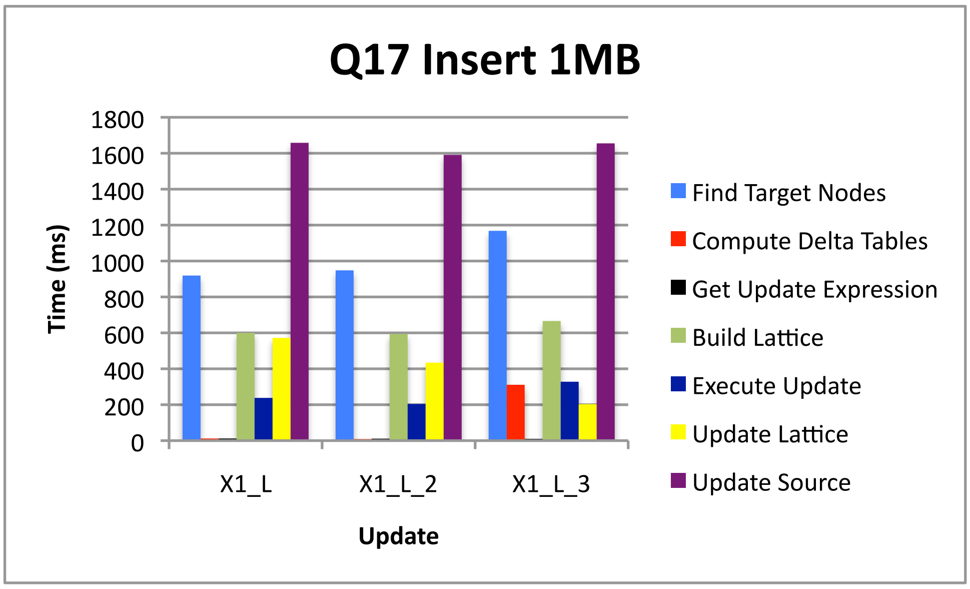


FIGURE C.82: Q17 Insert Graph 500KB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	919	12	12	600	238	572	1658
X1_L_2	948	9	11	594	204	434	1591
X1_L_3	1168	311	10	666	328	203	1655

TABLE C.83: Q17 View Insert 1Mb



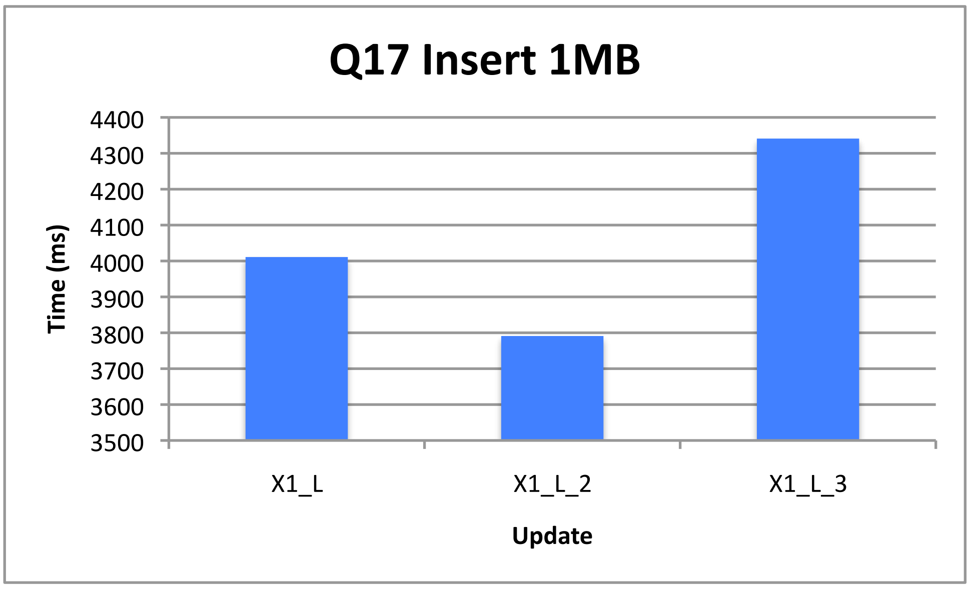
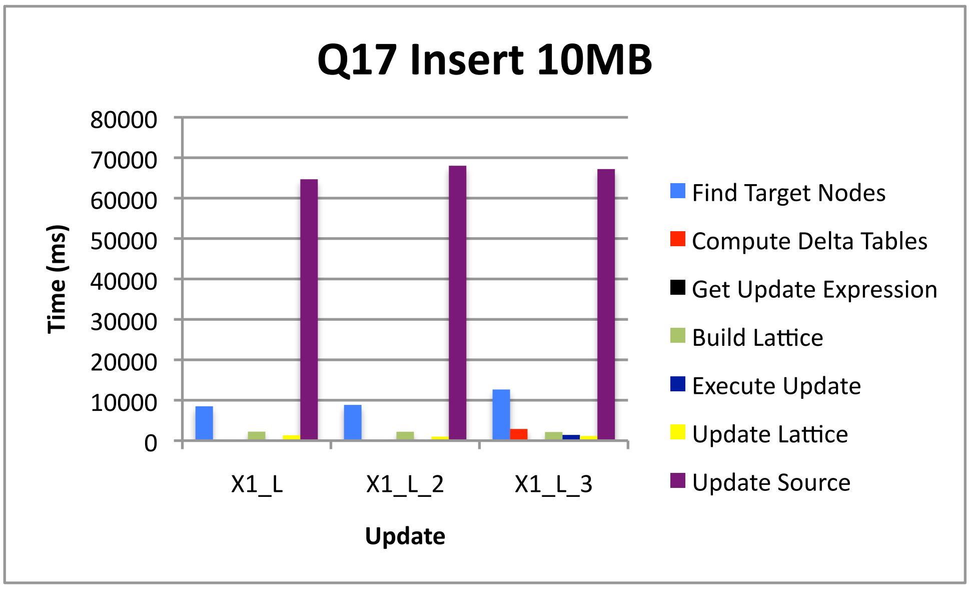


Figure C.83: Q17 Insert Graph 1MB

Query	TN	DT	UE	BL	EU	UL	US
X1_L	8503	9	11	2228	212	1339	64672
X1_L_2	8845	9	9	2203	181	992	68017
X1_L_3	12655	2888	16	2137	1417	1153	67193

Table C.84: Q17 View Insert 10Mb



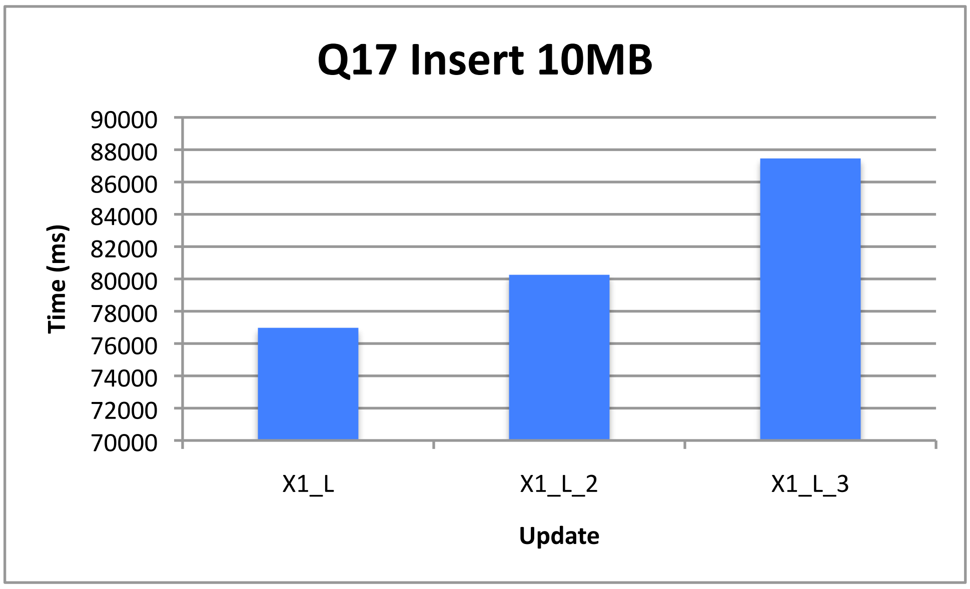
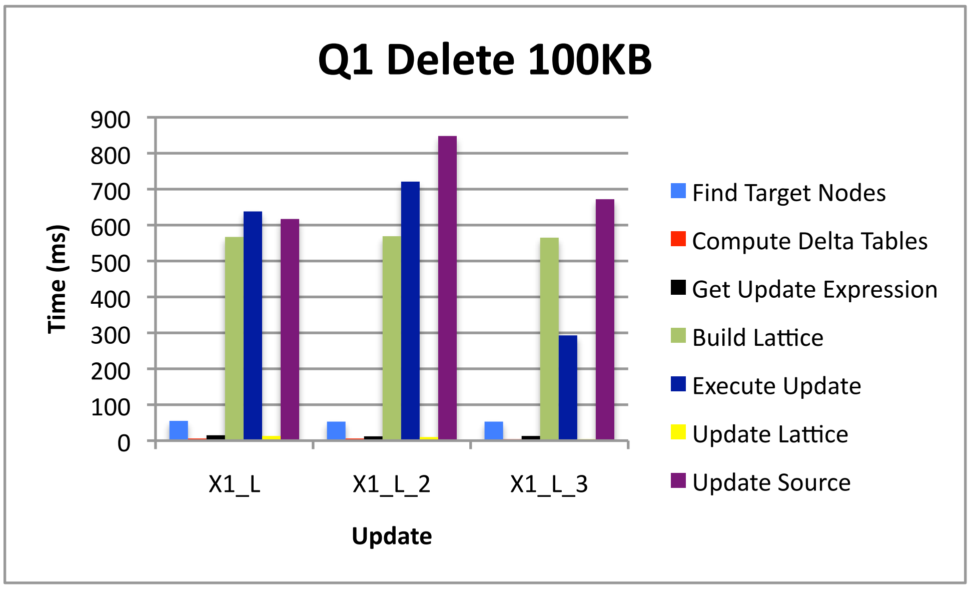


FIGURE C.84: Q17 Insert Graph 10MB

C.2.2 Delete

Query	TN	DT	UE	BL	EU	UL	US
DeleteX1_L	55	6	15	567	638	13	617
DeleteX1_L_2	53	6	12	569	721	10	848
DeleteX1_L_3	53	4	13	565	293	4	672

TABLE C.85: Q1 View Delete 100Kb



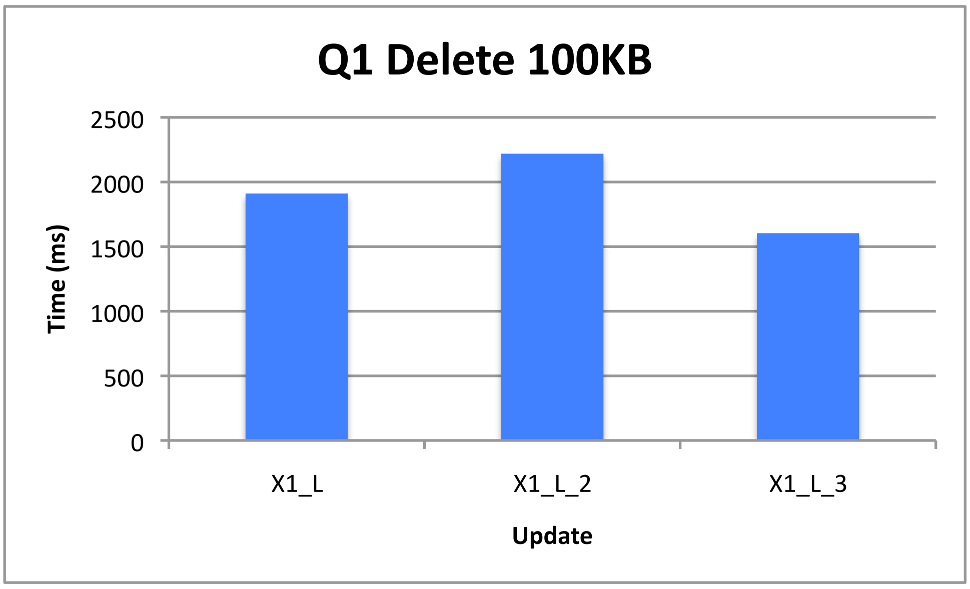
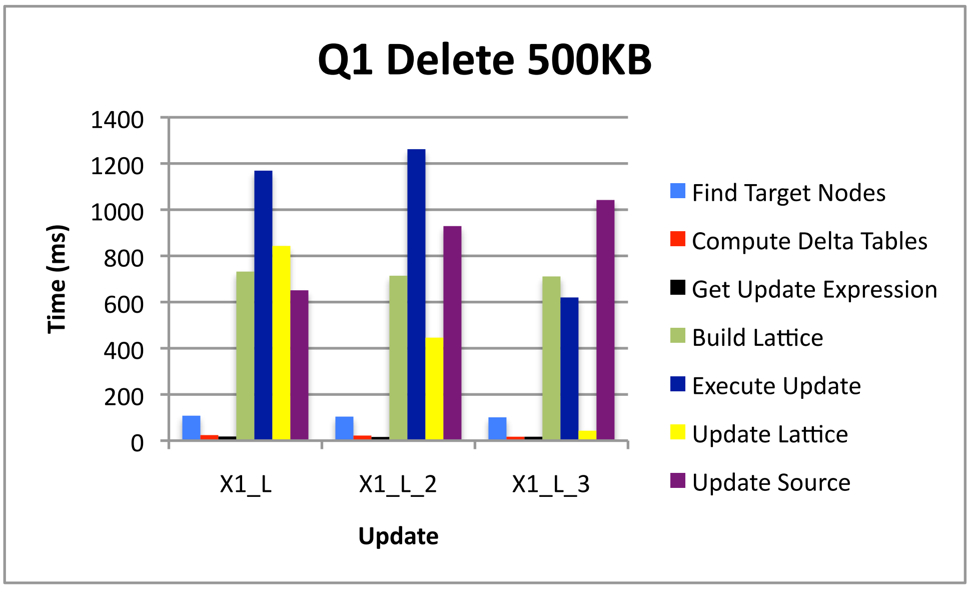


Figure C.85: Q1 Delete Graph 100KB

Query	TN	DT	UE	BL	EU	UL	US
DeleteX1_L	108	24	18	732	1169	843	651
DeleteX1_L_2	104	22	16	714	1262	446	929
DeleteX1_L_3	101	17	17	711	620	43	1042

Table C.86: Q1 View Delete 500Kb



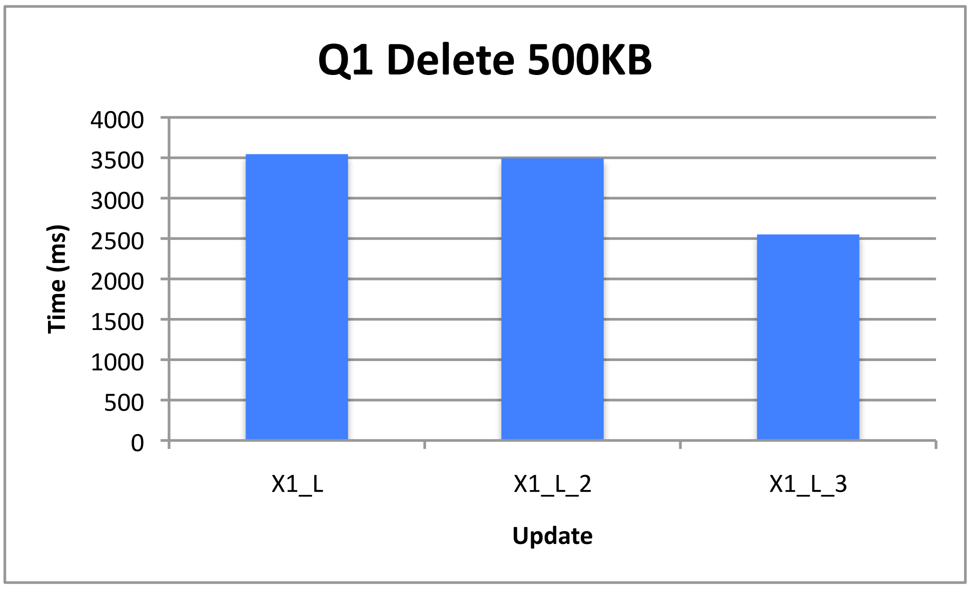
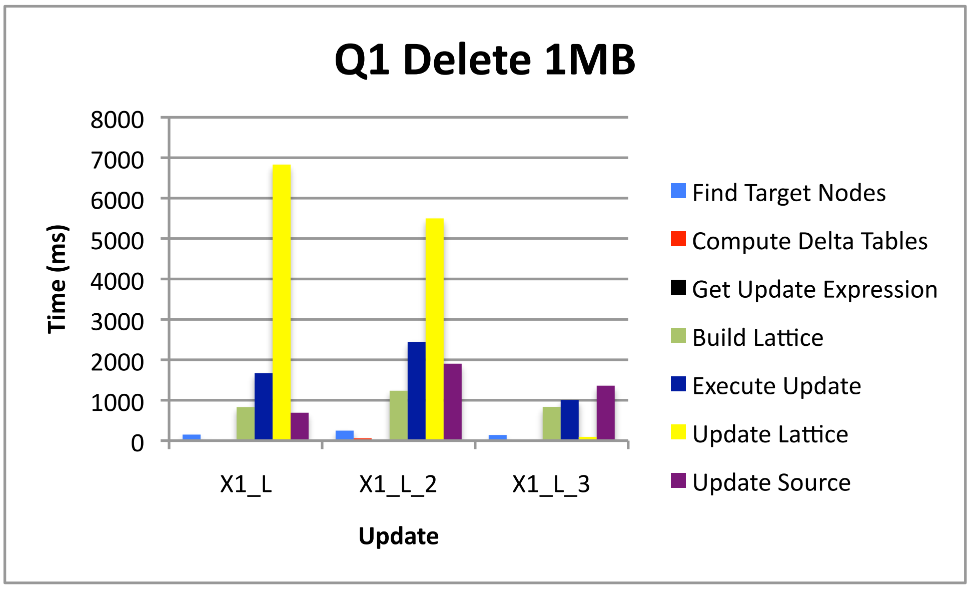


FIGURE C.86: Q1 Delete Graph 500KB

Query	TN	DT	UE	BL	EU	UL	US
DeleteX1_L	150	31	14	831	1671	6831	691
DeleteX1_L_2	248	56	24	1235	2445	5499	1904
DeleteX1_L_3	140	24	15	836	1008	94	1360

TABLE C.87: Q1 View Delete 1Mb



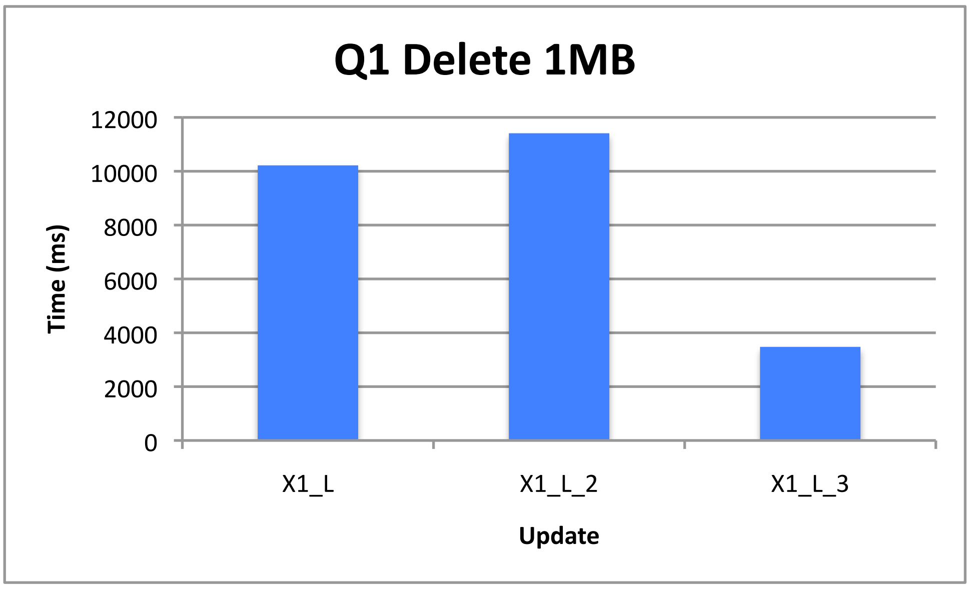
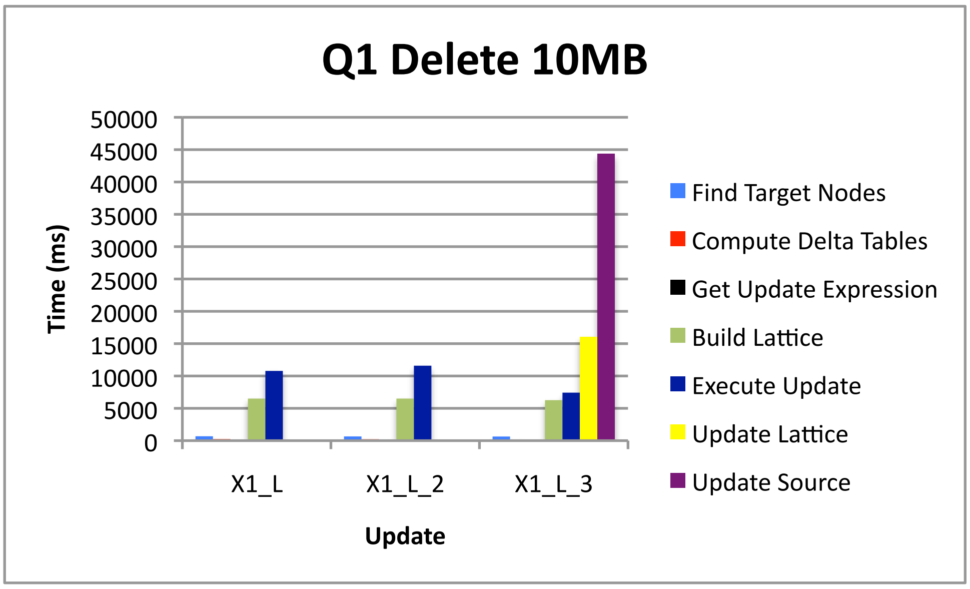


Figure C.87: Q1 Delete Graph 1MB

Query	TN	DT	UE	BL	EU	UL	US
DeleteX1_L	677	240	23	6504	10784	0	0
DeleteX1_L_2	652	220	25	6506	11591	0	0
DeleteX1_L_3	639	186	45	6264	7423	16031	44349

Table C.88: Q1 View Delete 10Mb



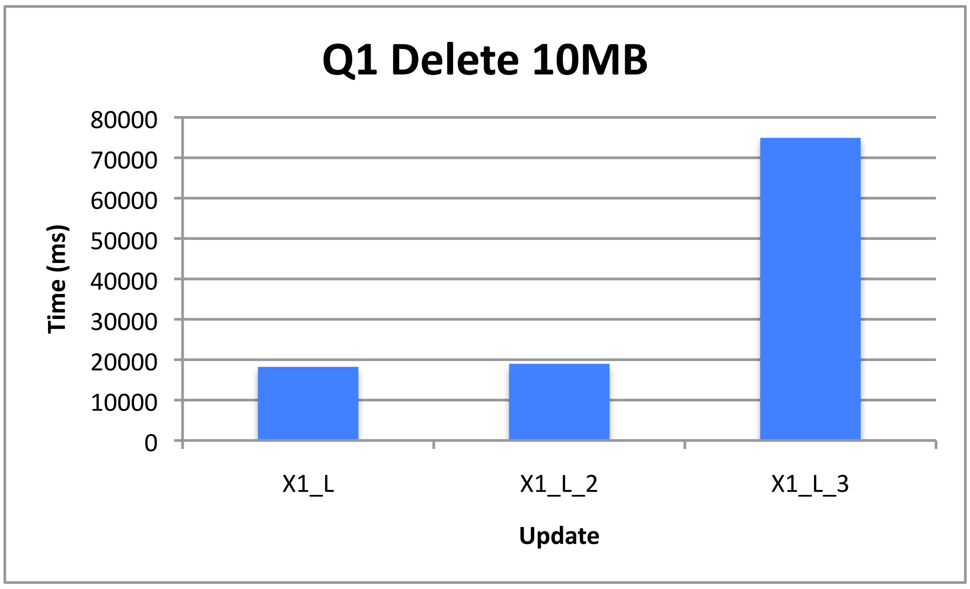
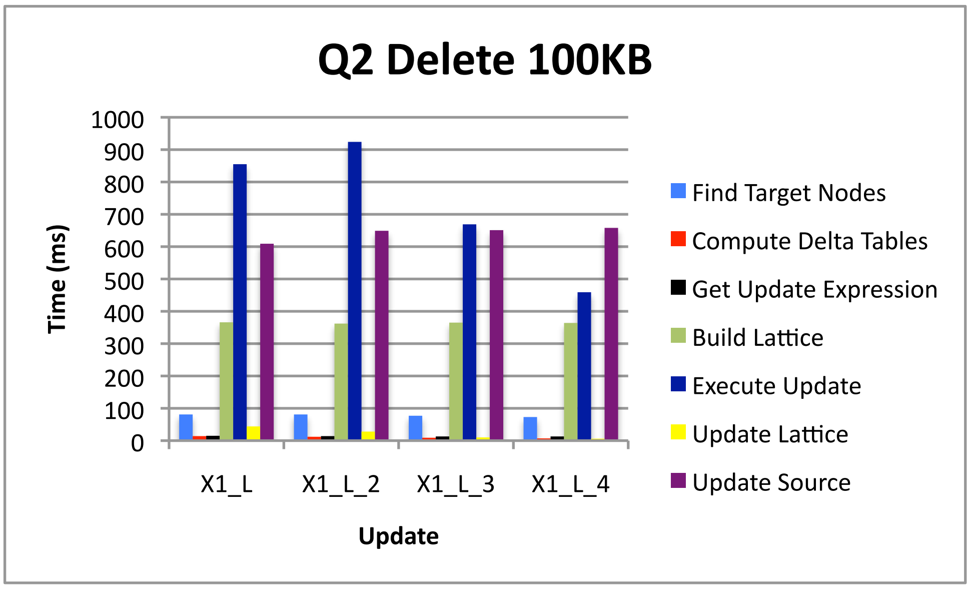


FIGURE C.88: Q1 Delete Graph 10MB

Query	TN	DT	UE	BL	EU	UL	US
DeleteX1_L	81	14	15	366	855	44	609
DeleteX1_L_2	81	12	14	362	924	28	649
DeleteX1_L_3	77	9	13	365	669	10	651
DeleteX1_L_4	73	7	13	364	459	6	658

TABLE C.89: Q2 View Delete 100Kb



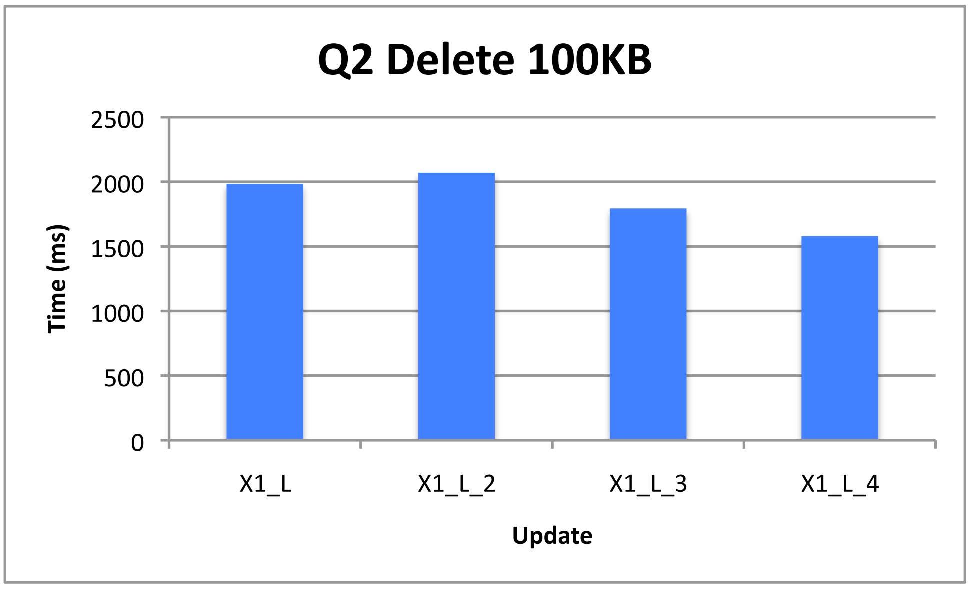
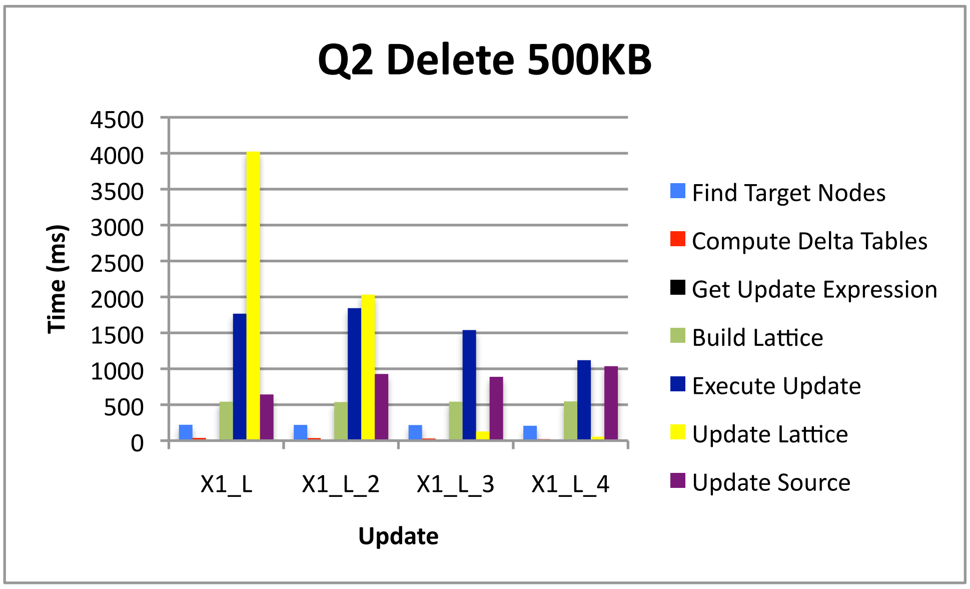


Figure C.89: Q2 Delete Graph 100KB

Query	TN	DT	UE	BL	EU	UL	US
DeleteX1_L	221	38	12	542	1767	4024	643
DeleteX1_L_2	219	36	13	537	1844	2032	928
DeleteX1_L_3	217	30	12	542	1539	128	888
DeleteX1_L_4	207	21	12	546	1120	55	1036

Table C.90: Q2 View Delete 500Kb



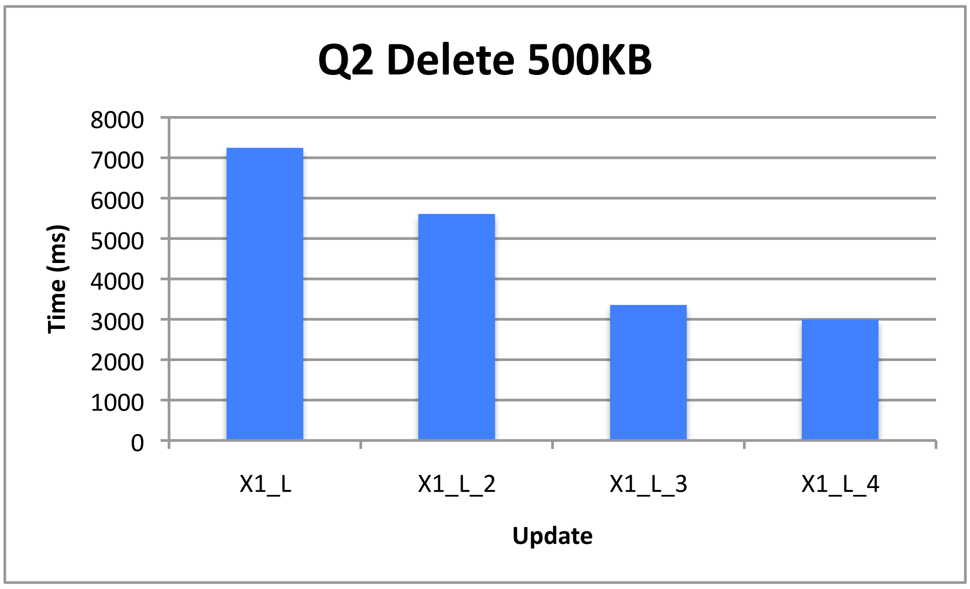
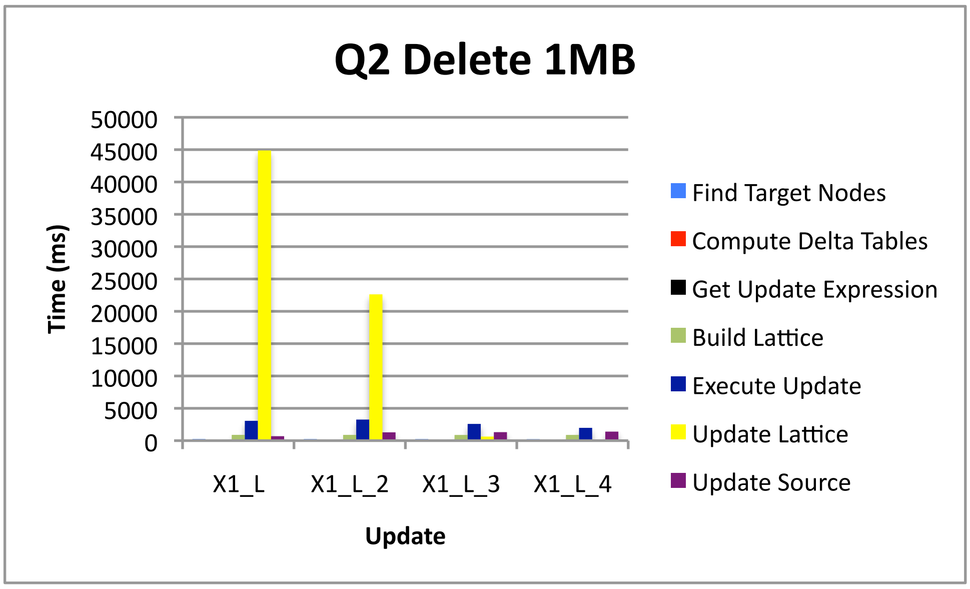


FIGURE C.90: Q2 Delete Graph 500KB

Query	TN	DT	UE	BL	EU	UL	US
DeleteX1_L	246	74	14	893	3065	44873	688
DeleteX1_L_2	241	72	15	890	3259	22621	1286
DeleteX1_L_3	239	61	15	881	2587	629	1313
DeleteX1_L_4	228	43	16	885	1979	149	1397

TABLE C.91: Q2 View Delete 1Mb



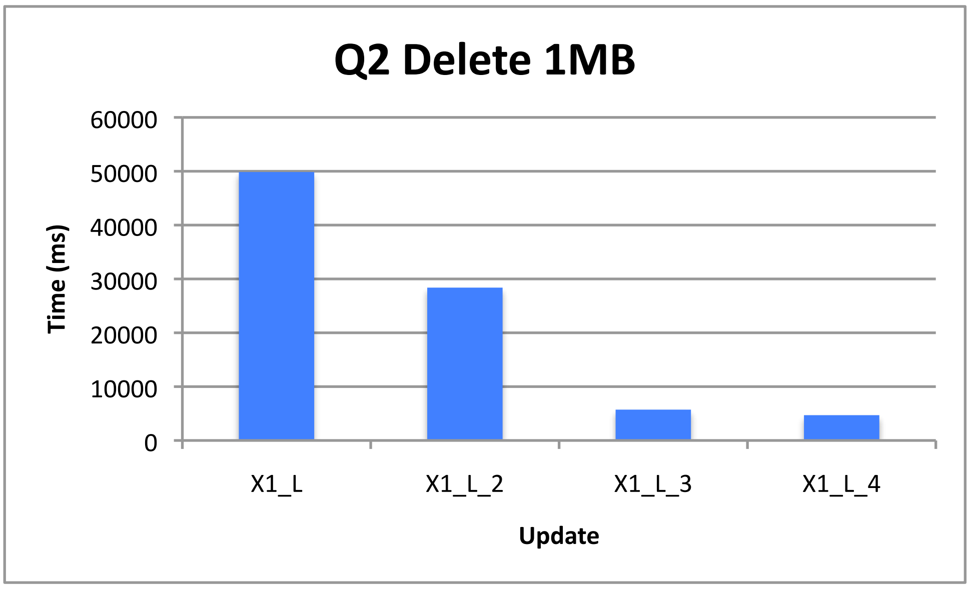
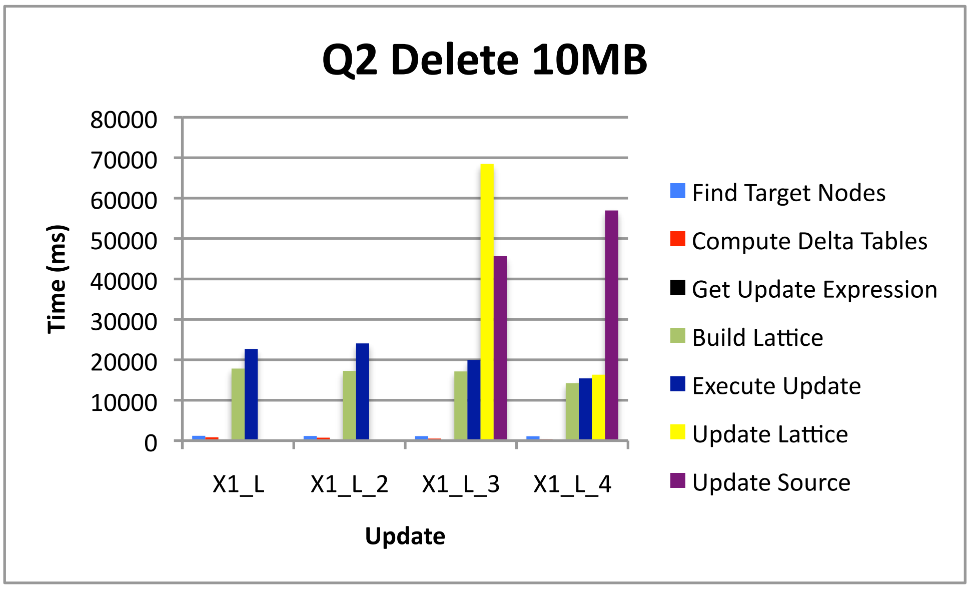


Figure C.91: Q2 Delete Graph 1MB

Query	TN	DT	UE	BL	EU	UL	US
DeleteX1_L	1203	809	29	17836	22687	0	0
DeleteX1_L_2	1151	744	31	17263	24061	0	0
DeleteX1_L_3	1110	517	173	17129	19973	68457	45637
DeleteX1_L_4	1068	364	34	14208	15406	16320	56952

Table C.92: Q2 View Delete 10Mb



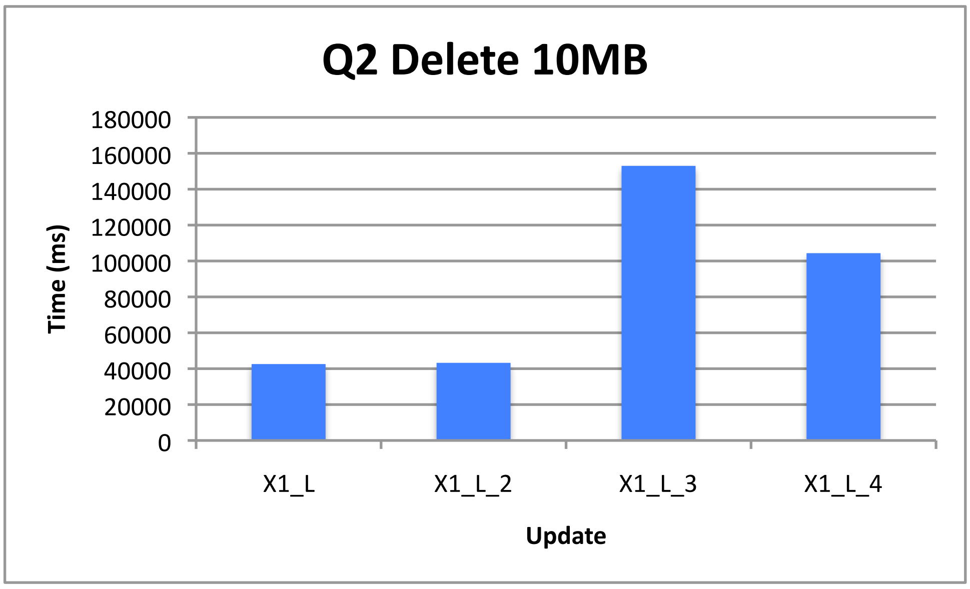
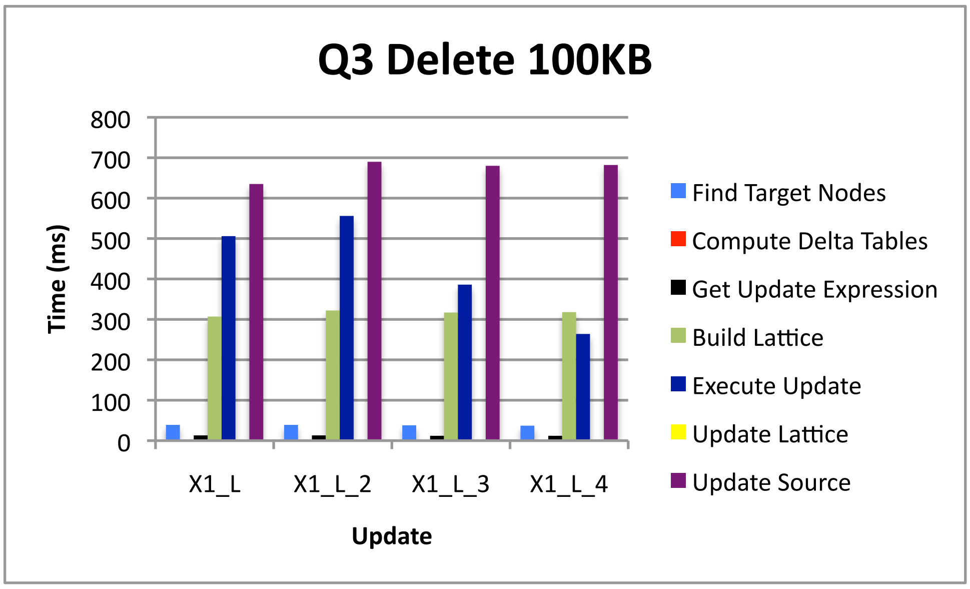


FIGURE C.92: Q2 Delete Graph 10MB

Query	TN	DT	UE	BL	EU	UL	US
DeleteX1_L	39	1	13	307	506	3	635
DeleteX1_L_2	39	1	13	322	556	3	690
DeleteX1_L_3	38	2	12	317	386	1	680
DeleteX1_L_4	37	0	12	318	264	0	682

TABLE C.93: Q3 View Delete 100Kb



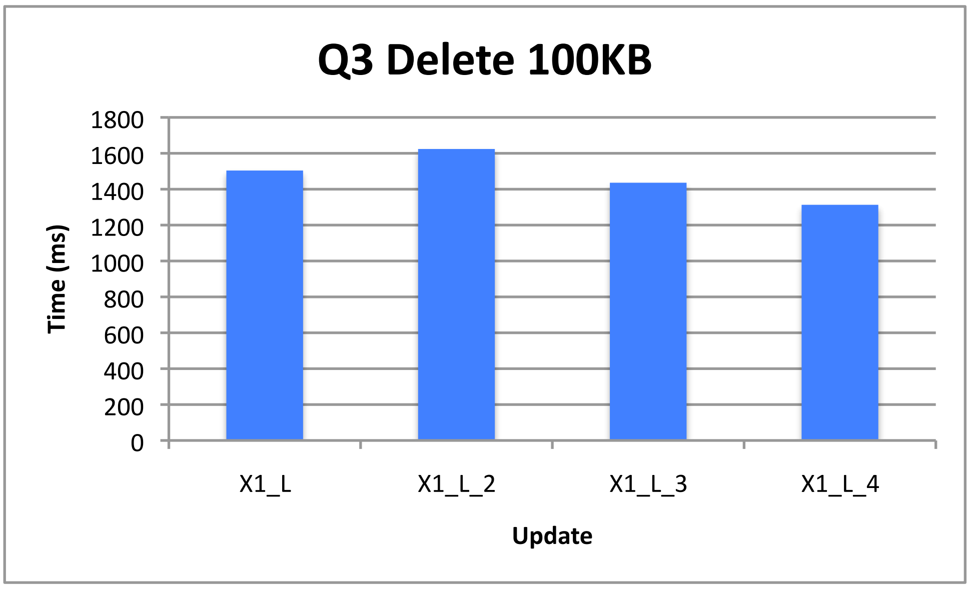
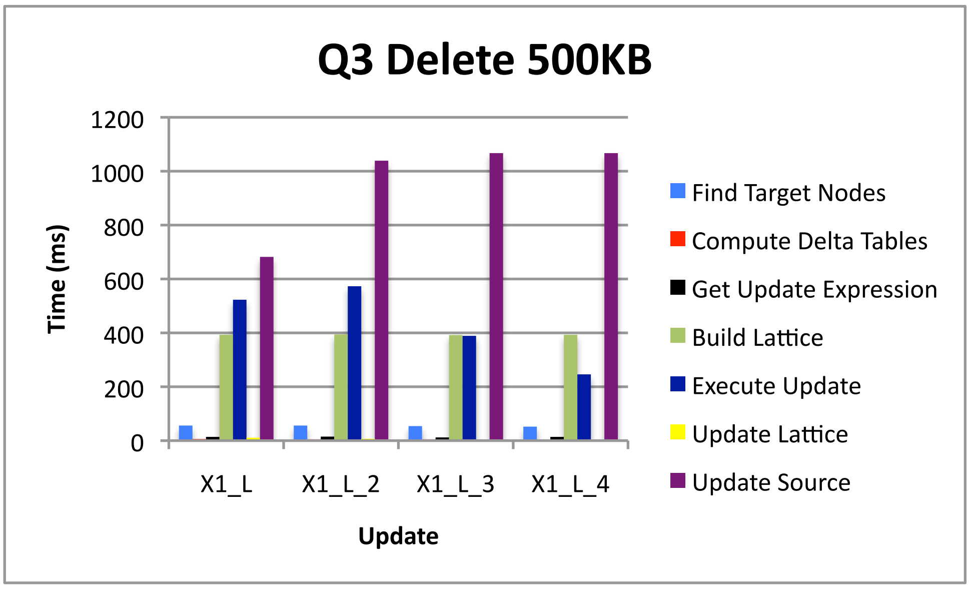


Figure C.93: Q3 Delete Graph 100KB

Query	TN	DT	UE	BL	EU	UL	US
DeleteX1_L	56	6	14	393	523	11	682
DeleteX1_L_2	56	5	15	394	573	7	1039
DeleteX1_L_3	54	5	12	392	389	3	1067
DeleteX1_L_4	52	4	14	393	246	2	1067

Table C.94: Q3 View Delete 500Kb



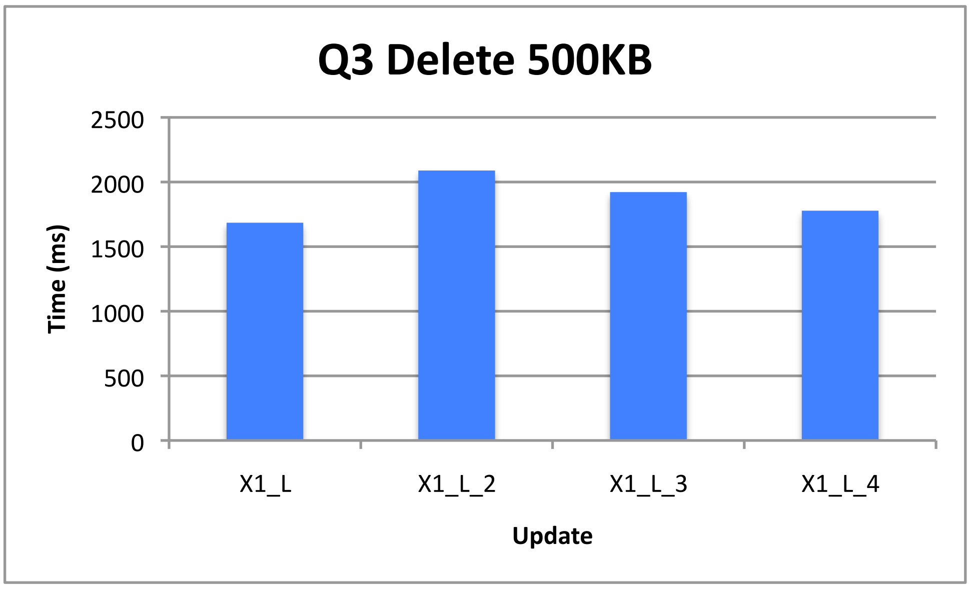
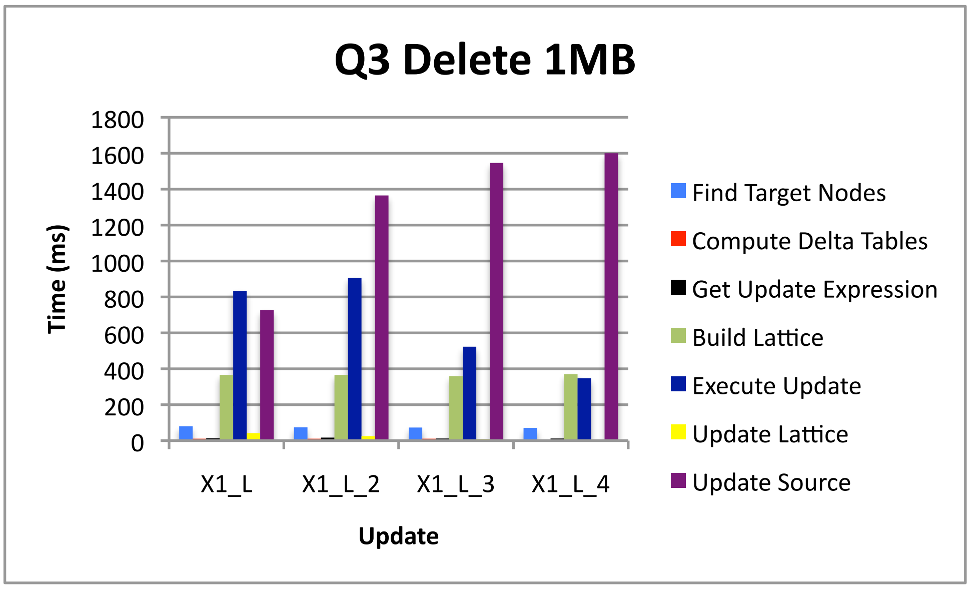


FIGURE C.94: Q3 Delete Graph 500KB

Query	TN	DT	UE	BL	EU	UL	US
DeleteX1_L	80	11	13	366	834	42	726
DeleteX1_L_2	74	11	17	366	906	25	1365
DeleteX1_L_3	73	11	12	359	523	9	1546
DeleteX1_L_4	71	7	12	370	347	6	1600

TABLE C.95: Q3 View Delete 1Mb



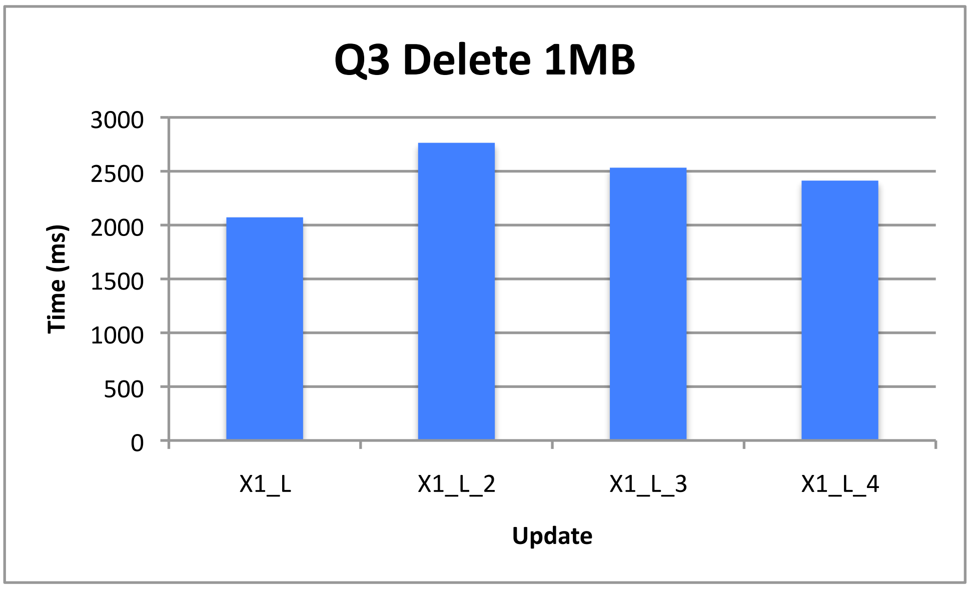
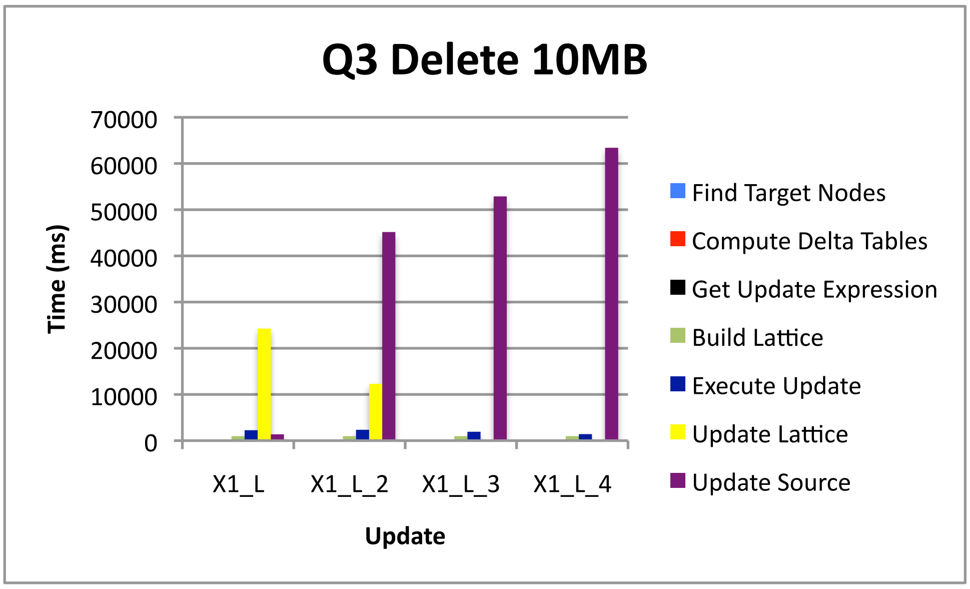


Figure C.95: Q3 Delete Graph 1MB

Query	TN	DT	UE	BL	EU	UL	US
DeleteX1_L	205	57	14	963	2259	24254	1372
DeleteX1_L_2	202	52	14	972	2352	12302	45164
DeleteX1_L_3	194	44	18	969	1923	189	52878
DeleteX1_L_4	191	33	15	977	1413	87	63405

Table C.96: Q3 View Delete 10Mb



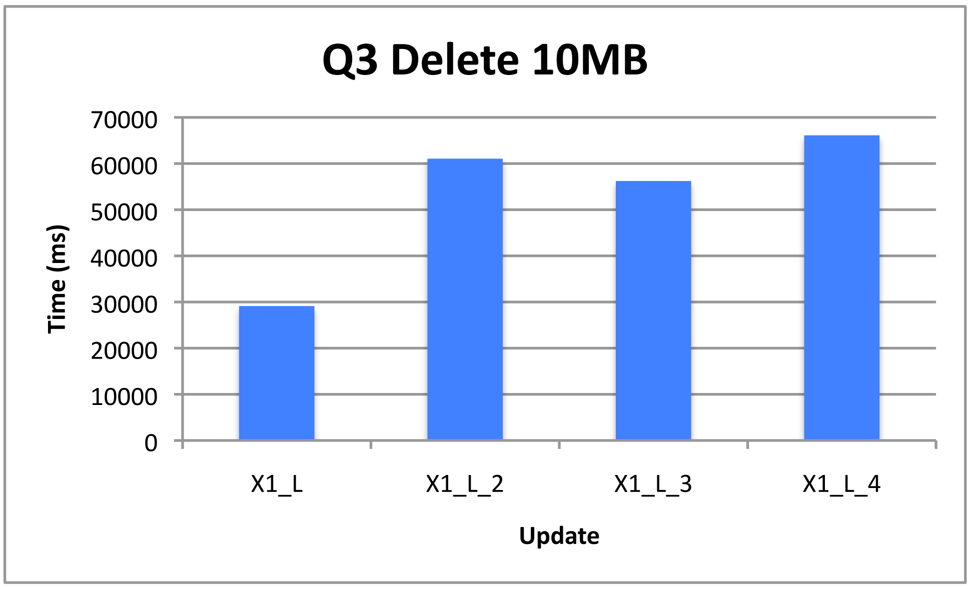
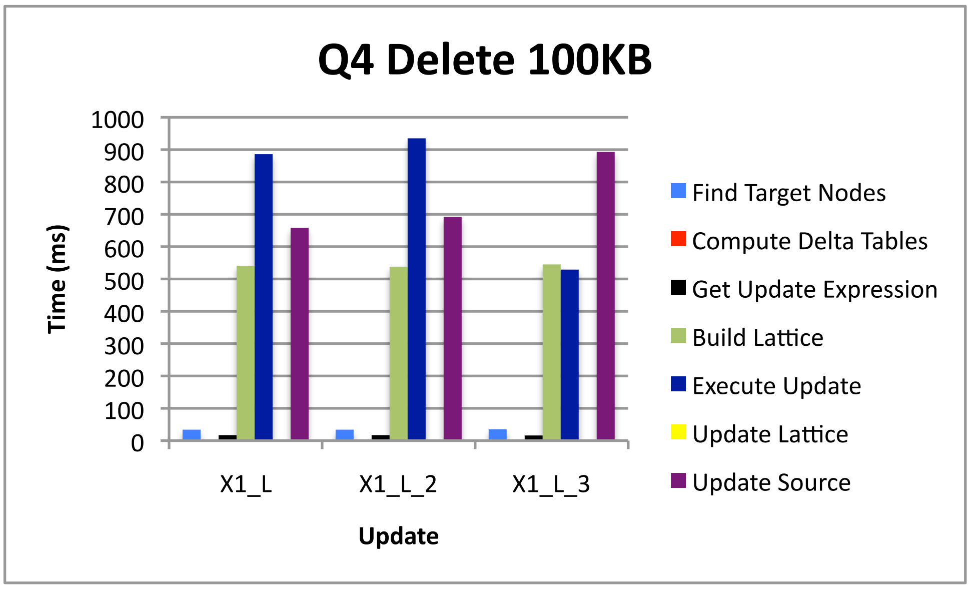


FIGURE C.96: Q3 Delete Graph 10MB

Query	TN	DT	UE	BL	EU	UL	US
DeleteX1_L	34	1	17	541	886	0	658
DeleteX1_L_2	34	1	17	538	935	0	692
DeleteX1_L_3	35	1	16	545	529	0	893

TABLE C.97: Q4 View Delete 100Kb



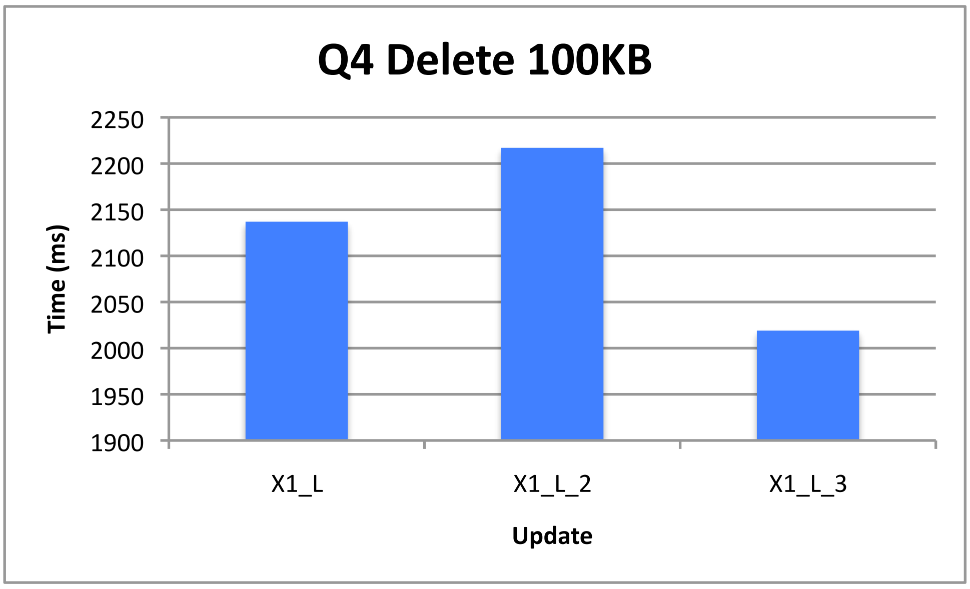
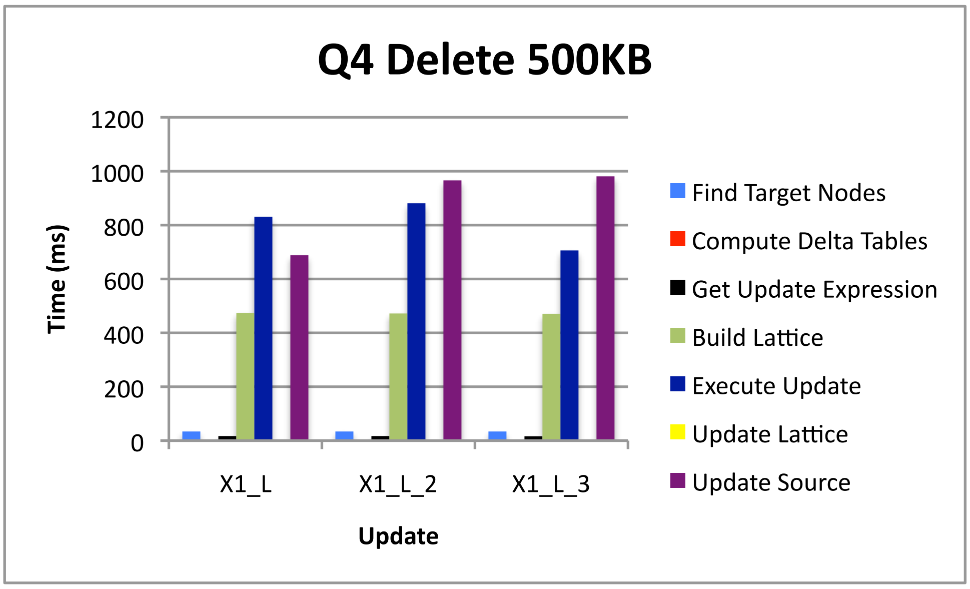


Figure C.97: Q4 Delete Graph 100KB

Query	TN	DT	UE	BL	EU	UL	US
DeleteX1_L	34	1	17	474	831	1	688
DeleteX1_L_2	34	1	17	472	881	0	966
DeleteX1_L_3	34	0	16	471	706	1	981

Table C.98: Q4 View Delete 500Kb



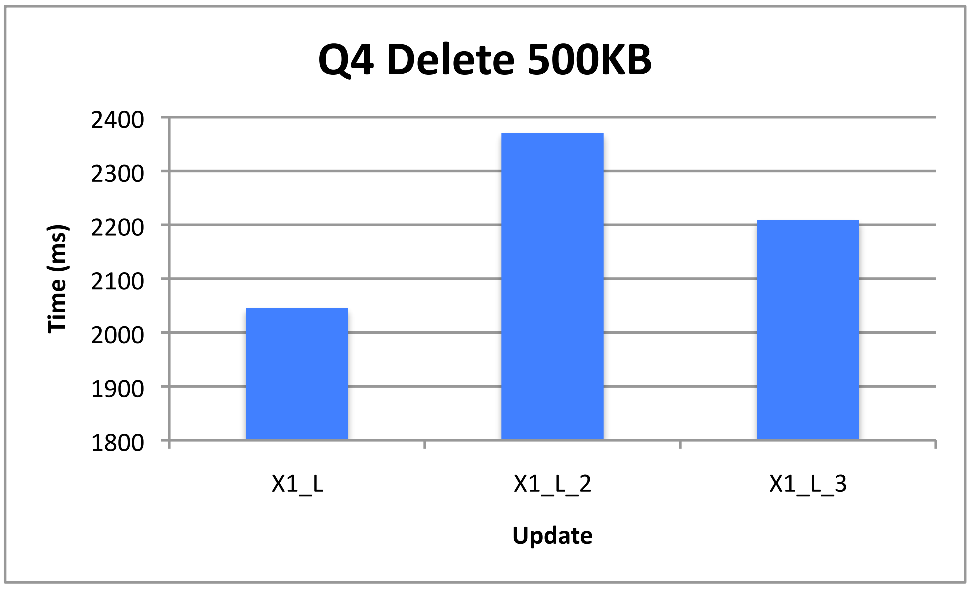
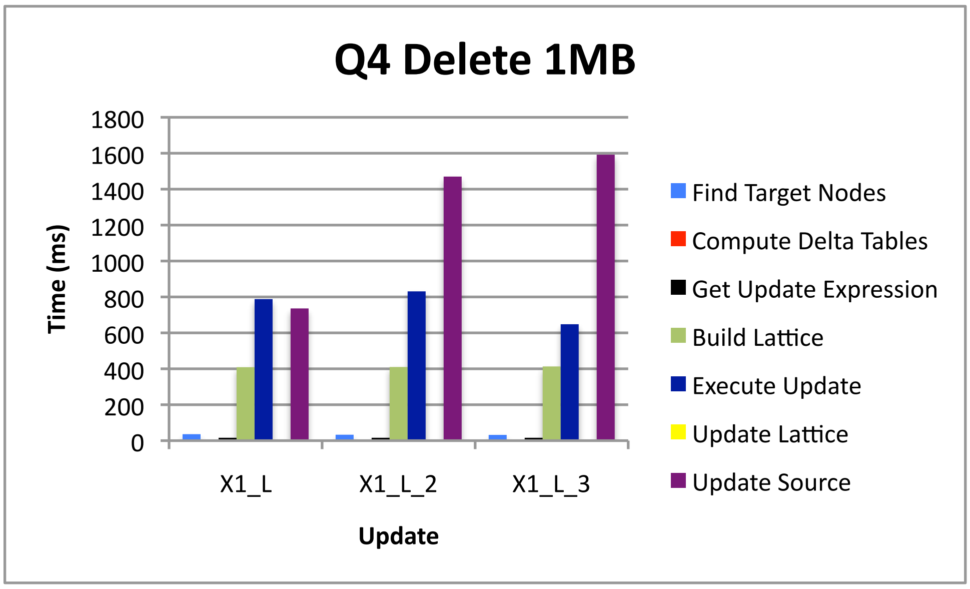


FIGURE C.98: Q4 Delete Graph 500KB

Query	TN	DT	UE	BL	EU	UL	US
DeleteX1_L	36	1	16	409	788	1	736
DeleteX1_L_2	33	1	16	410	831	1	1470
DeleteX1_L_3	32	1	16	413	648	0	1592

TABLE C.99: Q4 View Delete 1Mb



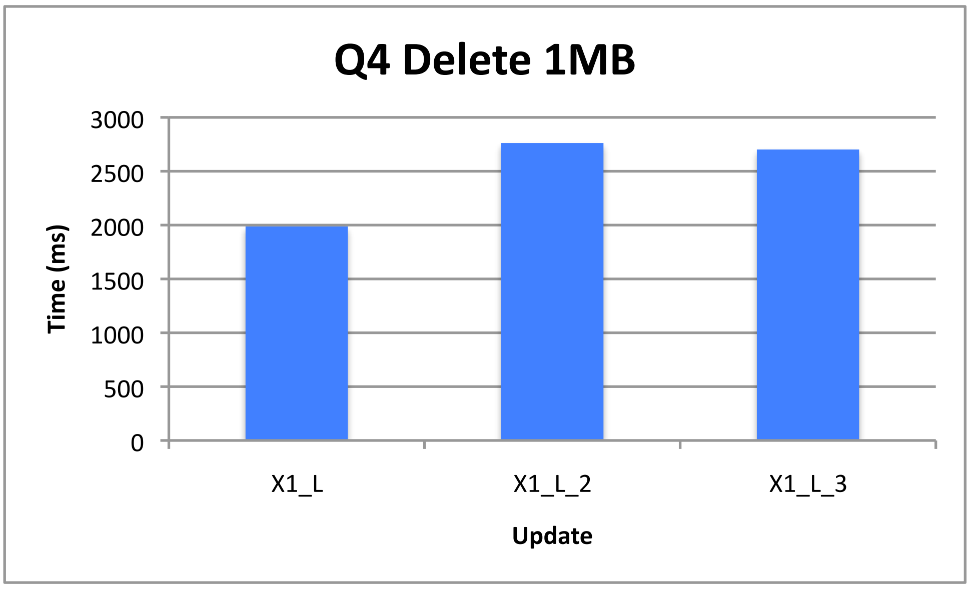
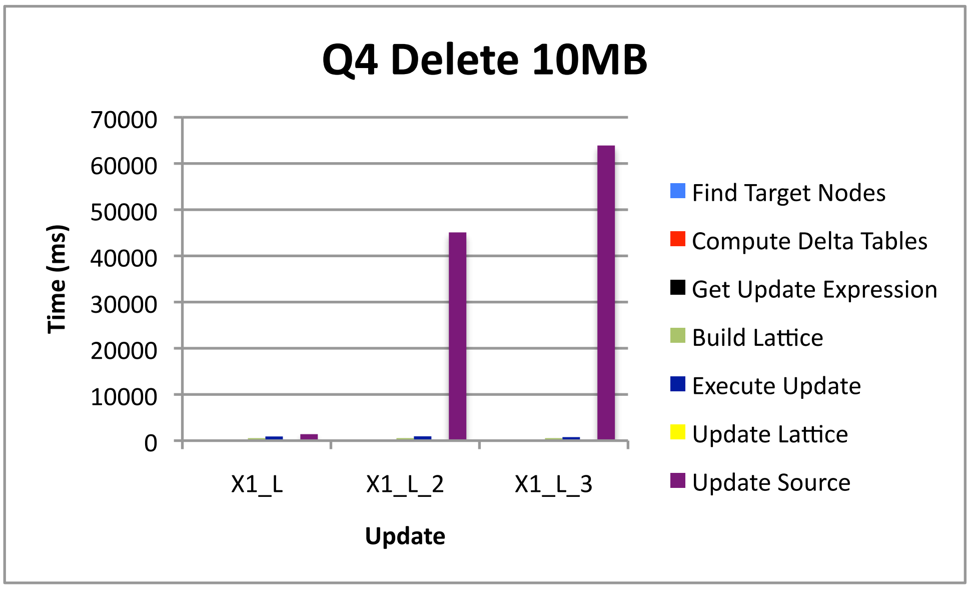


Figure C.99: Q4 Delete Graph 1MB

Query	TN	DT	UE	BL	EU	UL	US
DeleteX1_L	33	2	15	539	906	3	1403
DeleteX1_L_2	35	1	15	548	942	2	45083
DeleteX1_L_3	33	1	15	545	762	1	63897

Table C.100: Q4 View Delete 10Mb



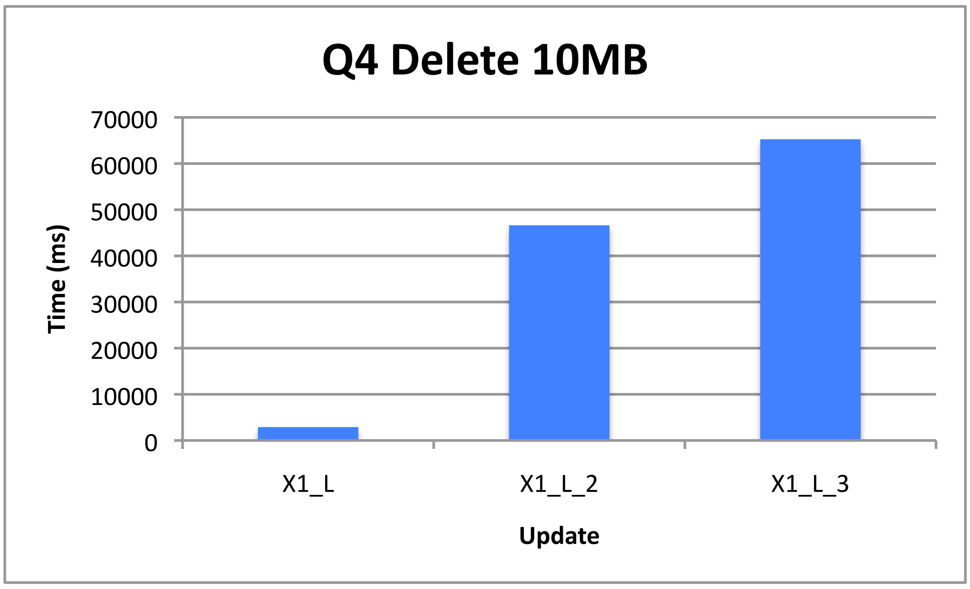
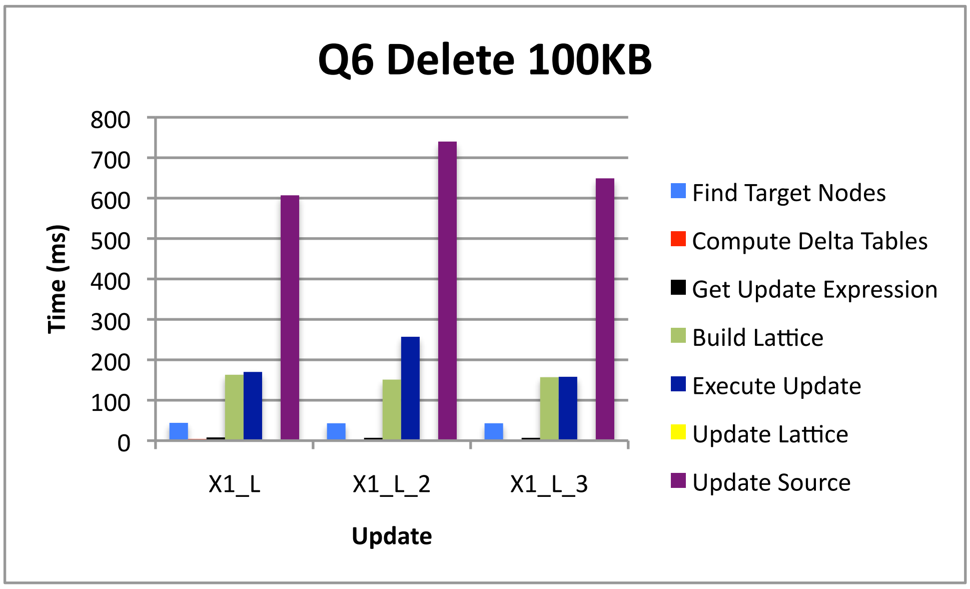


FIGURE C.100: Q4 Delete Graph 10MB

Query	TN	DT	UE	BL	EU	UL	US
DeleteX1_L	44	4	8	163	170	2	607
DeleteX1_L_2	43	2	7	151	257	1	740
DeleteX1_L_3	43	1	7	157	158	1	649

TABLE C.101: Q6 View Delete 100Kb



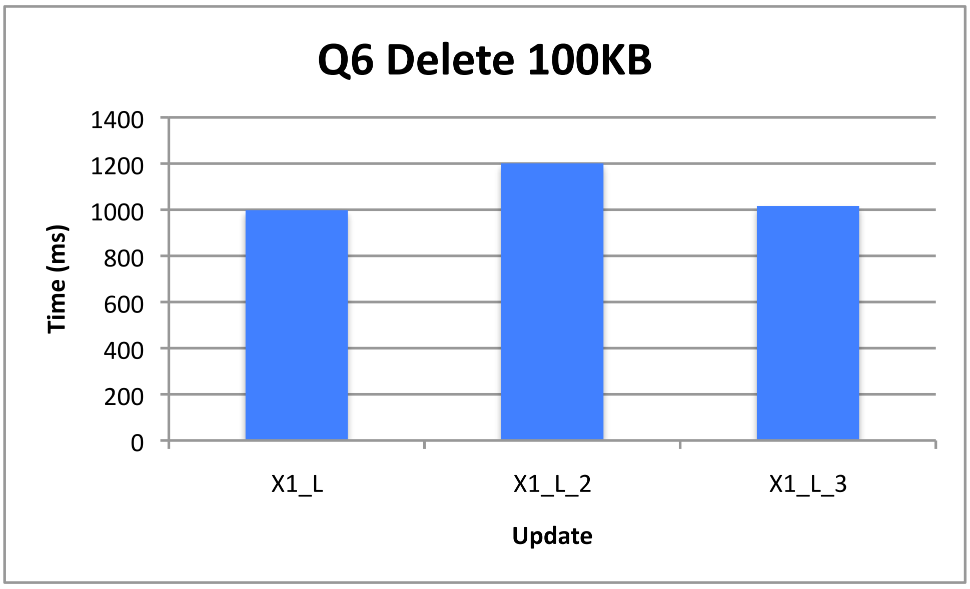
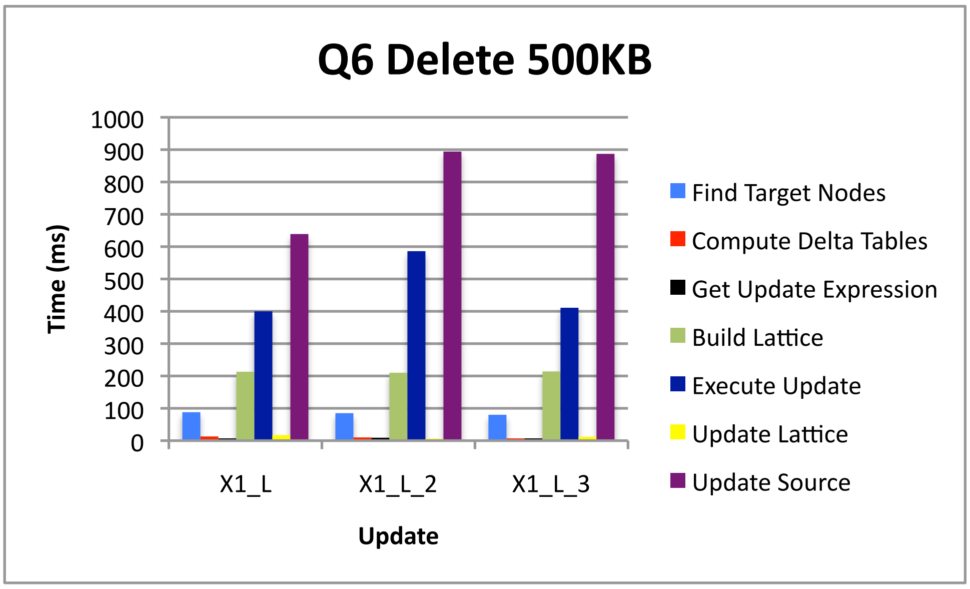


Figure C.101: Q6 Delete Graph 100KB

Query	TN	DT	UE	BL	EU	UL	US
DeleteX1_L	88	13	7	213	400	17	639
DeleteX1_L_2	85	10	9	210	586	6	894
DeleteX1_L_3	80	7	7	214	411	12	887

Table C.102: Q6 View Delete 500Kb



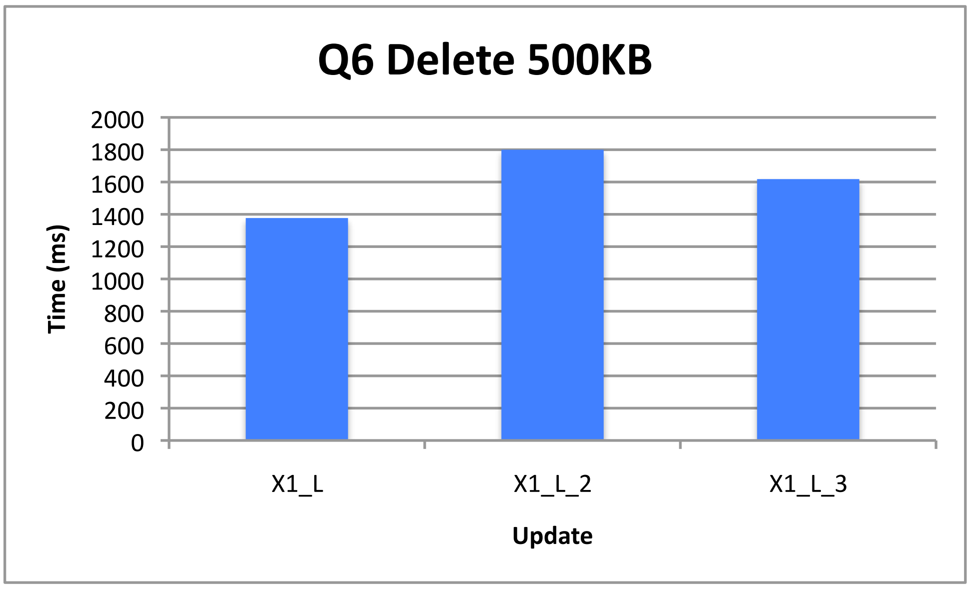
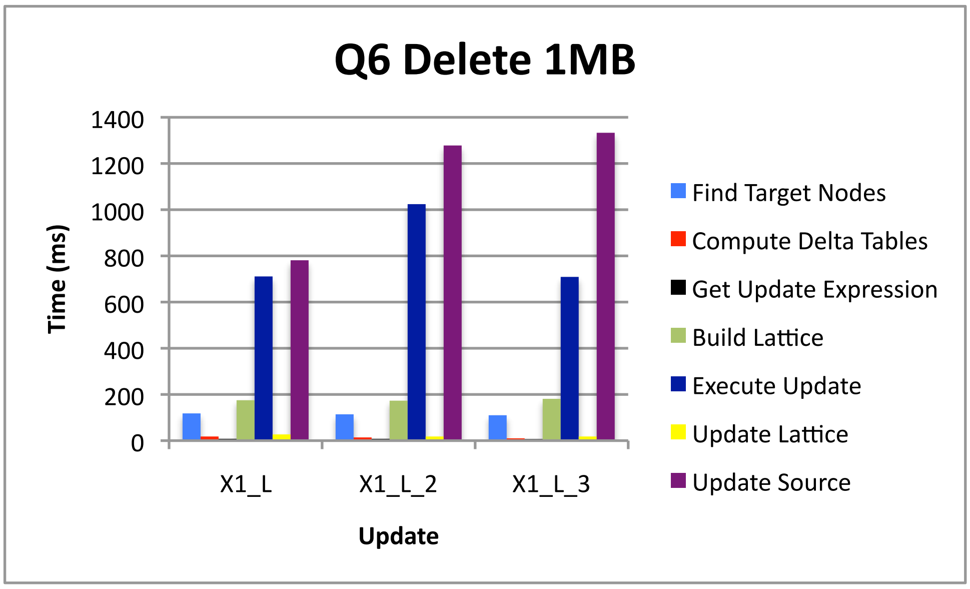


FIGURE C.102: Q6 Delete Graph 500KB

Query	TN	DT	UE	BL	EU	UL	US
DeleteX1_L	118	18	8	175	711	27	781
DeleteX1_L_2	114	14	8	173	1024	18	1278
DeleteX1_L_3	110	10	7	181	709	18	1333

TABLE C.103: Q6 View Delete 1Mb



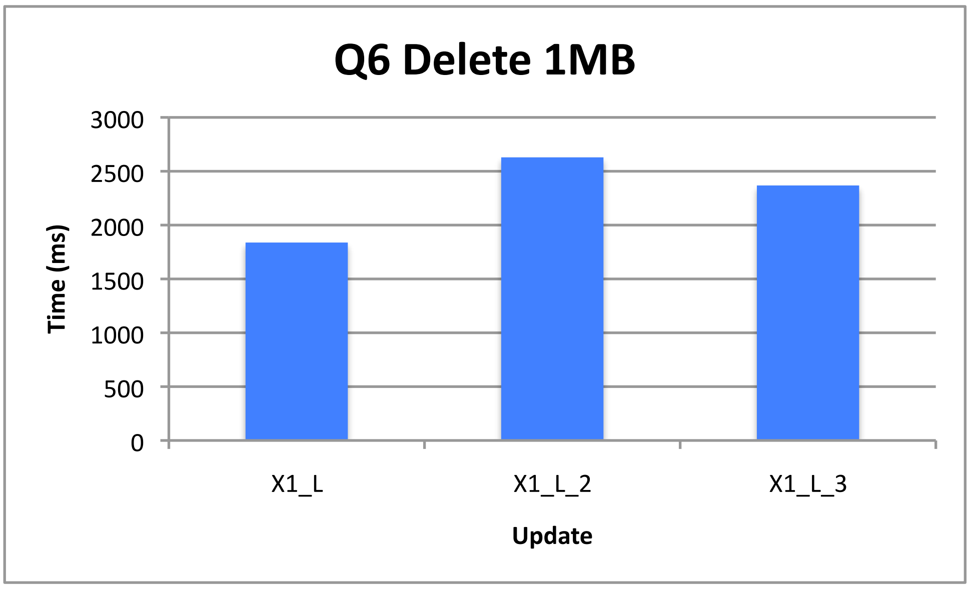
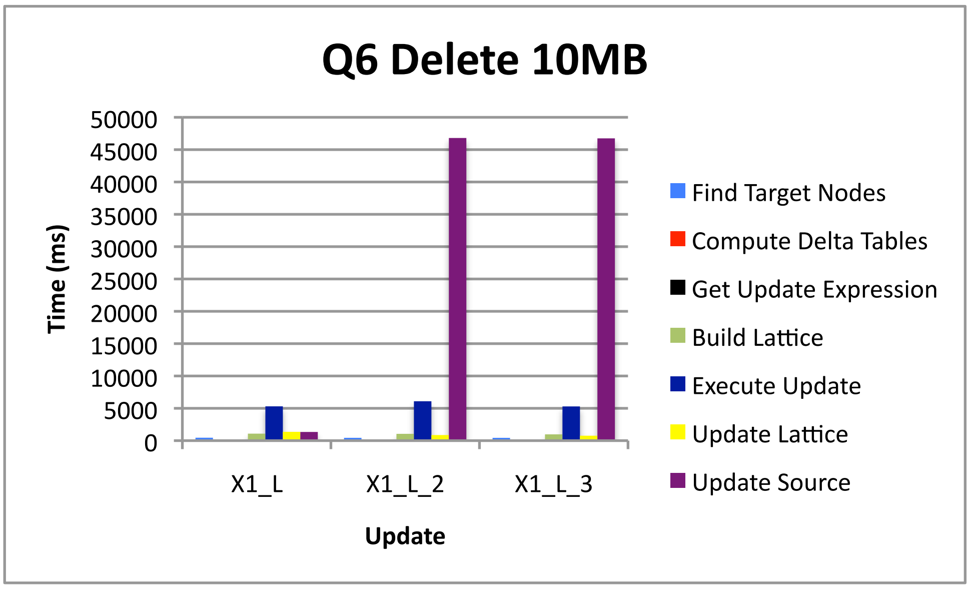


Figure C.103: Q6 Delete Graph 1MB

Query	TN	DT	UE	BL	EU	UL	US
DeleteX1_L	452	114	16	1080	5305	1362	1343
DeleteX1_L_2	435	99	17	1051	6092	863	46800
DeleteX1_L_3	432	62	6	973	5291	762	46745

Table C.104: Q6 View Delete 10Mb



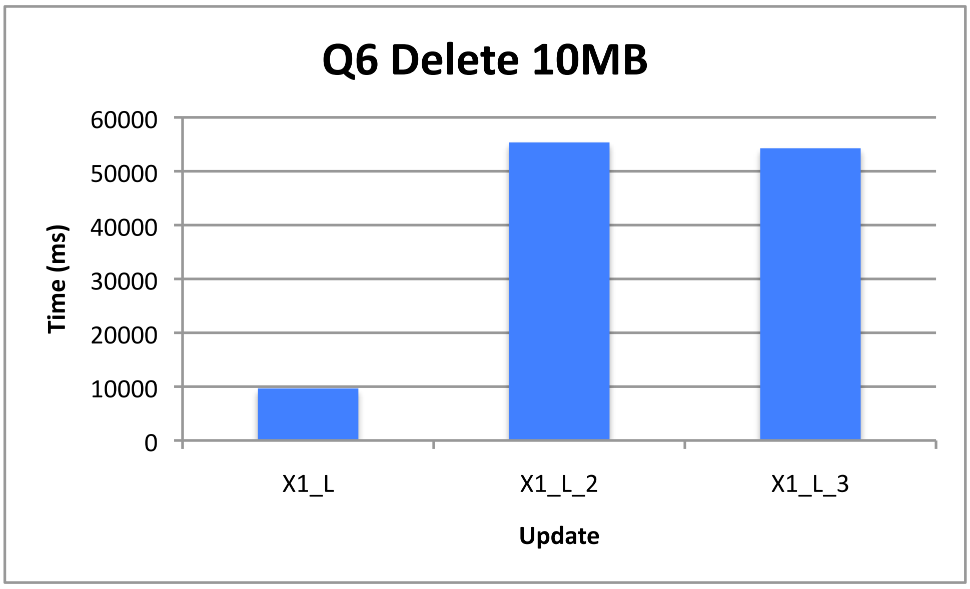
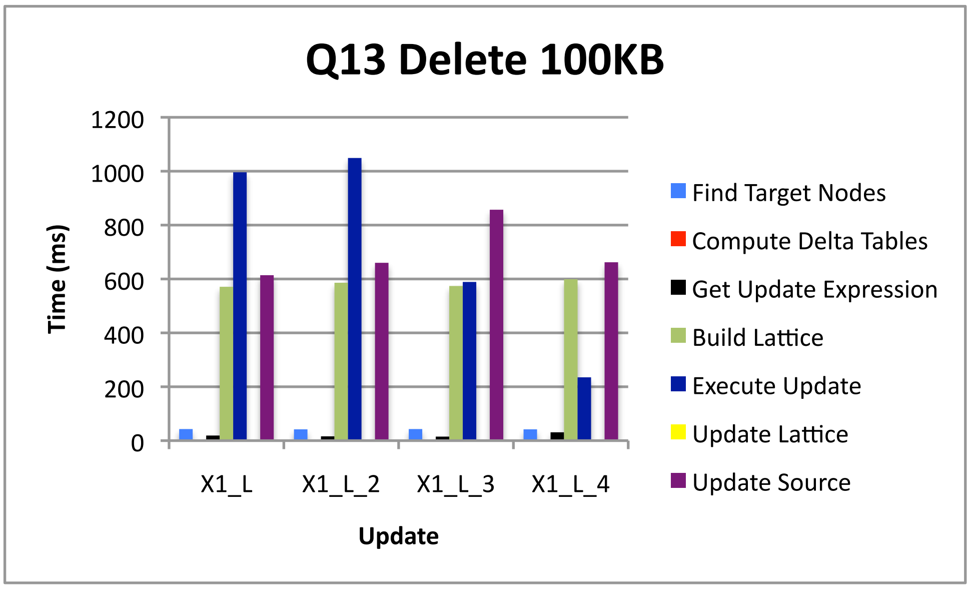


FIGURE C.104: Q6 Delete Graph 10MB

Query	TN	DT	UE	BL	EU	UL	US
DeleteX1_L	43	3	19	571	996	3	614
DeleteX1_L_2	42	4	16	586	1049	3	660
DeleteX1_L_3	43	3	15	574	589	2	857
DeleteX1_L_4	42	3	31	599	235	3	662

TABLE C.105: Q13 View Delete 100Kb



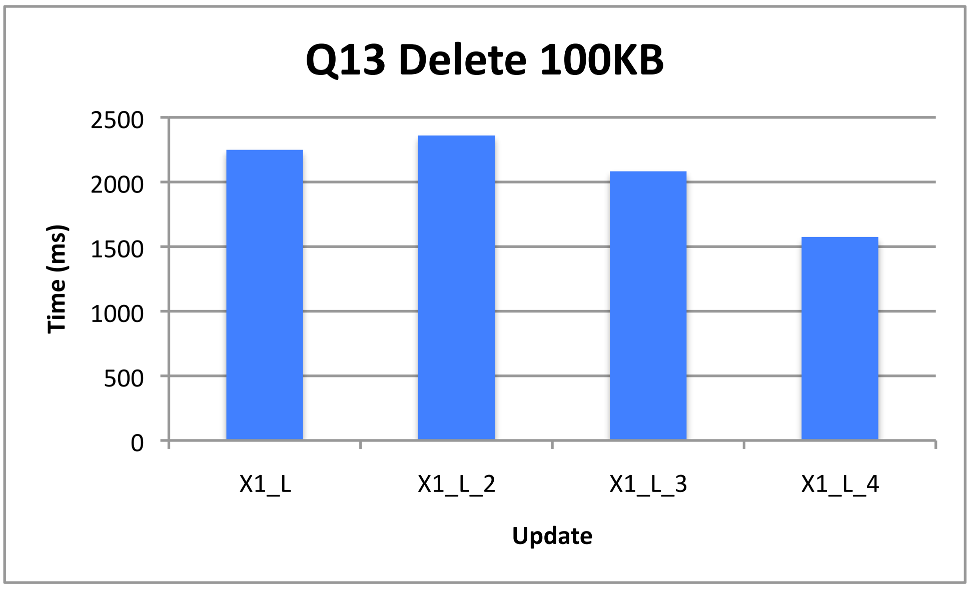
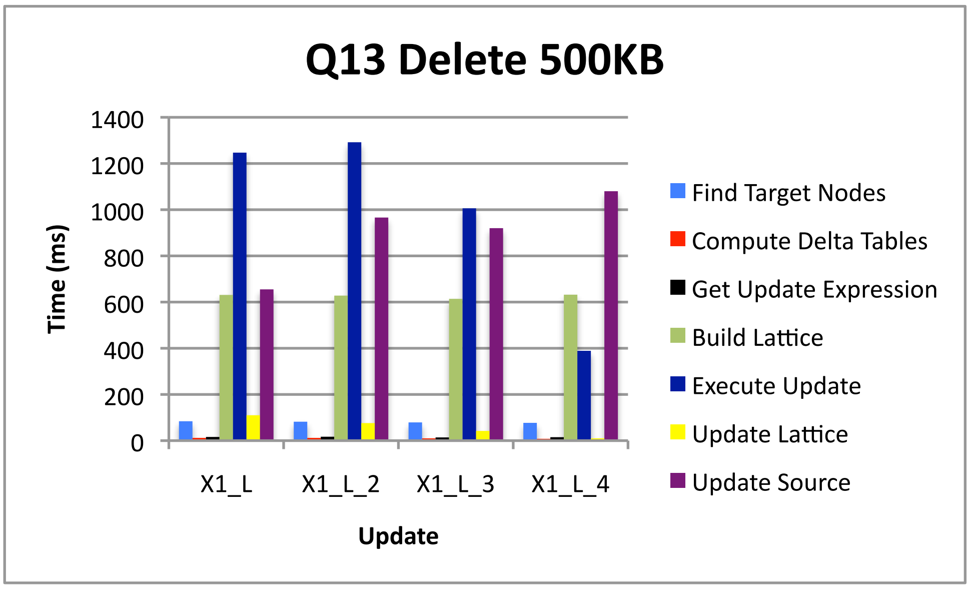


Figure C.105: Q13 Delete Graph 100KB

Query	TN	DT	UE	BL	EU	UL	US
DeleteX1_L	84	12	16	631	1247	110	655
DeleteX1_L_2	82	12	17	628	1292	76	966
DeleteX1_L_3	79	10	14	614	1006	42	920
DeleteX1_L_4	77	8	15	632	389	10	1080

Table C.106: Q13 View Delete 500Kb



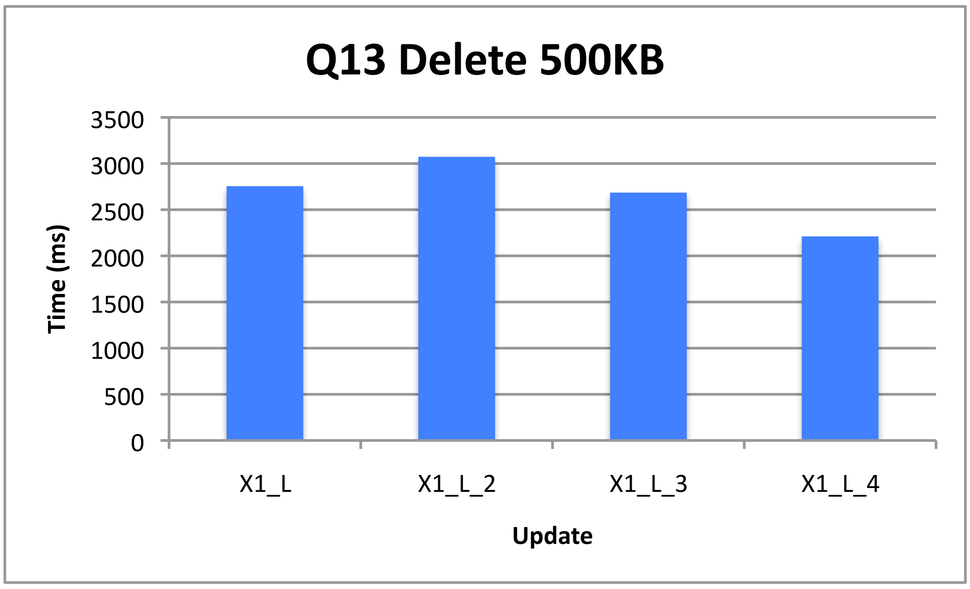
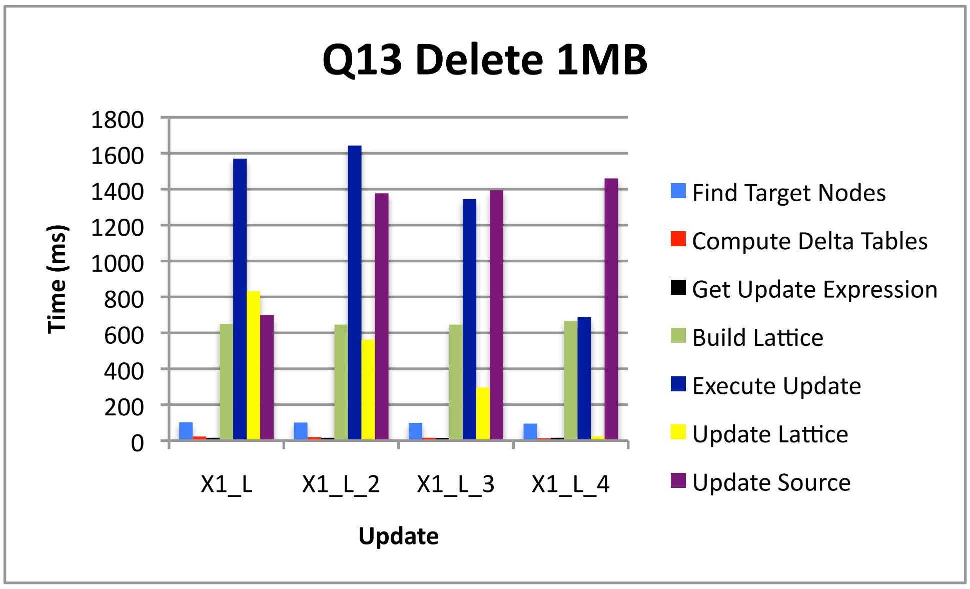


FIGURE C.106: Q13 Delete Graph 500KB

Query	TN	DT	UE	BL	EU	UL	US
DeleteX1_L	102	23	16	650	1570	832	699
DeleteX1_L_2	101	20	16	646	1643	563	1377
DeleteX1_L_3	99	16	15	646	1345	296	1394
DeleteX1_L_4	95	13	16	666	687	24	1460

TABLE C.107: Q13 View Delete 1Mb



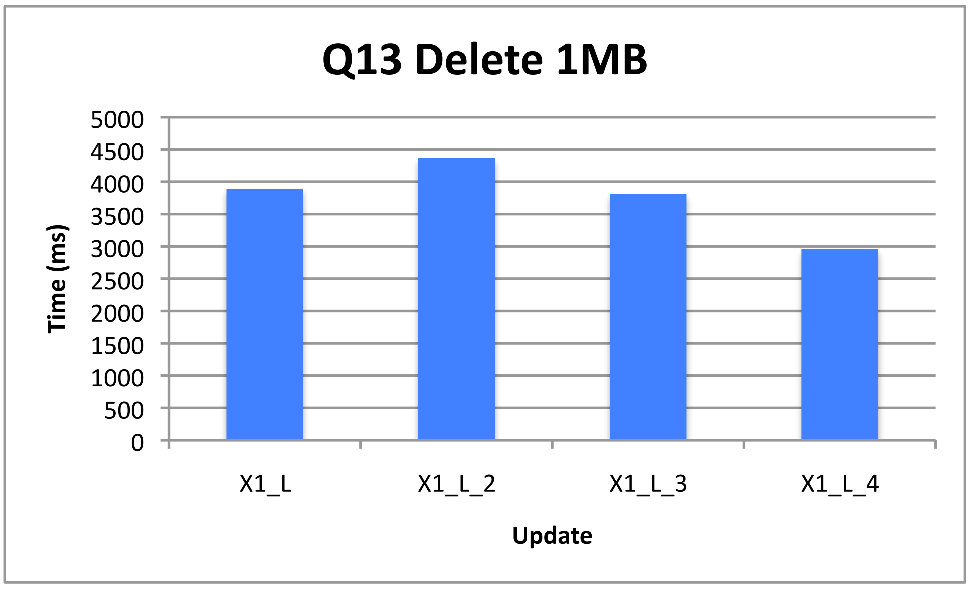
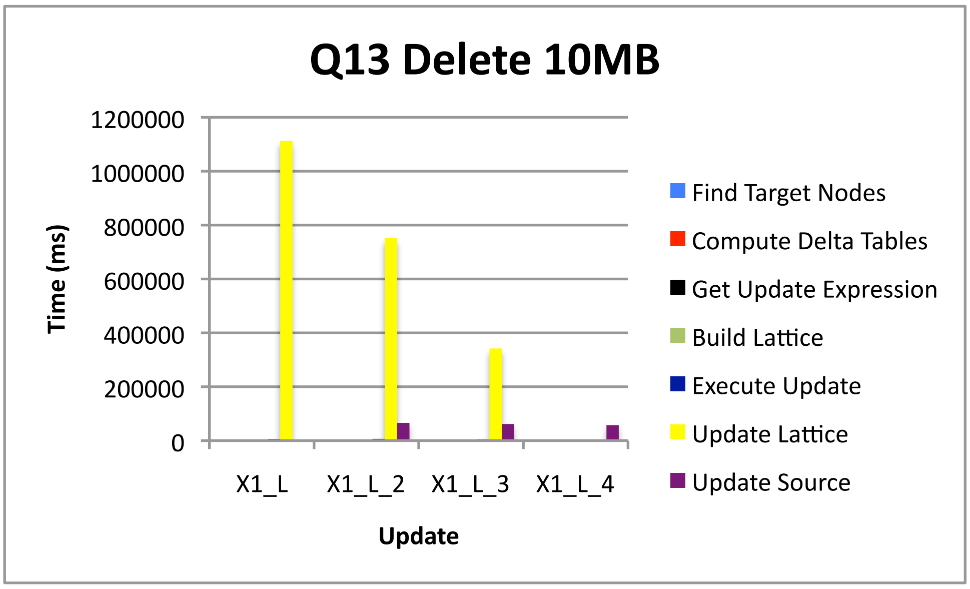


Figure C.107: Q13 Delete Graph 1MB

Query	TN	DT	UE	BL	EU	UL	US
DeleteX1_L	372	145	18	2046	6280	1112427	0
DeleteX1_L_2	361	130	21	2065	6769	751974	65854
DeleteX1_L_3	356	113	19	2067	5382	342064	61806
DeleteX1_L_4	347	93	32	2046	3222	1746	57349

Table C.108: Q13 View Delete 10Mb



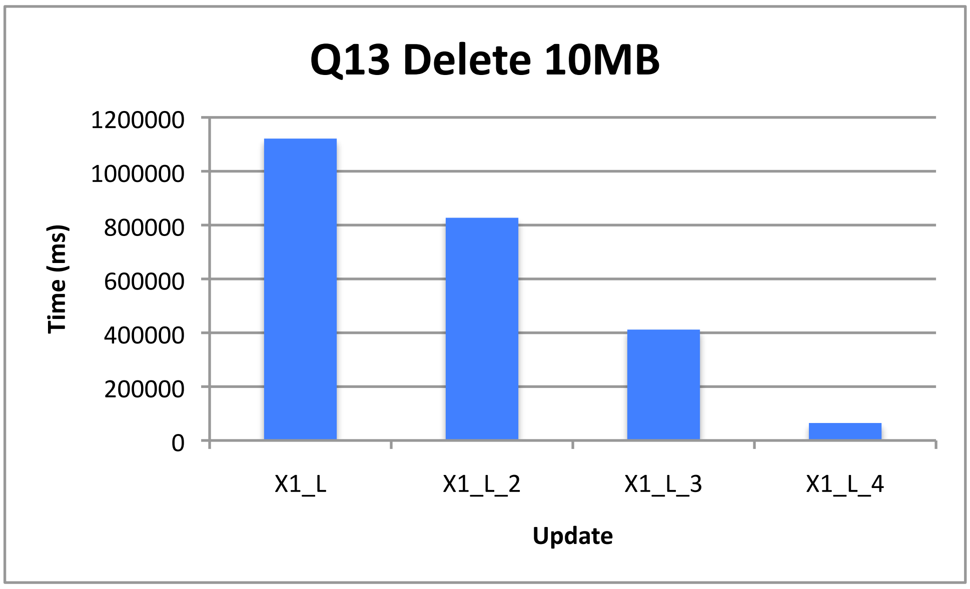
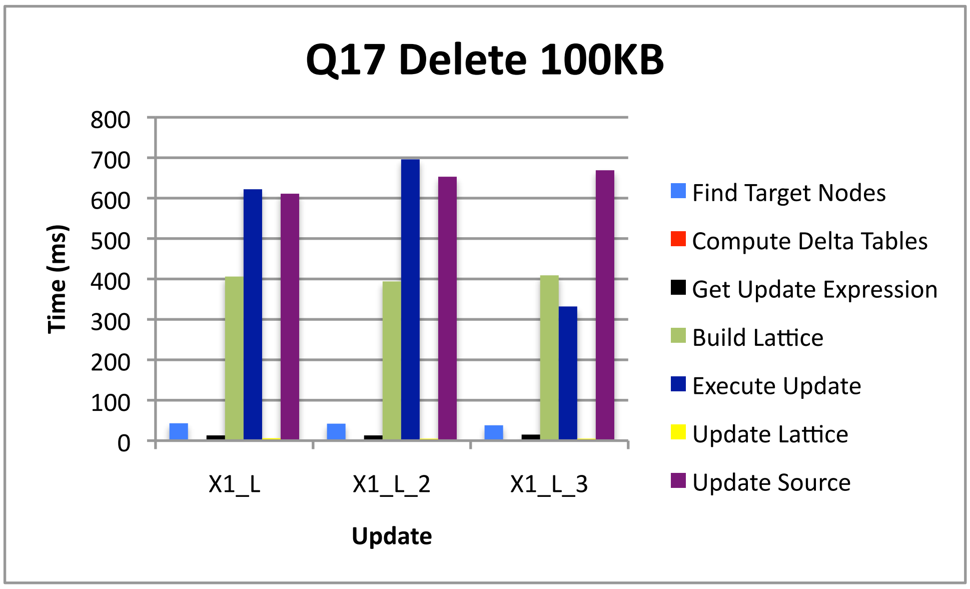


FIGURE C.108: Q13 Delete Graph 10MB

Query	TN	DT	UE	BL	EU	UL	US
DeleteX1_L	43	3	13	406	622	6	611
DeleteX1_L_2	42	3	13	394	696	5	653
DeleteX1_L_3	38	2	15	409	332	5	669

TABLE C.109: Q17 View Delete 100Kb



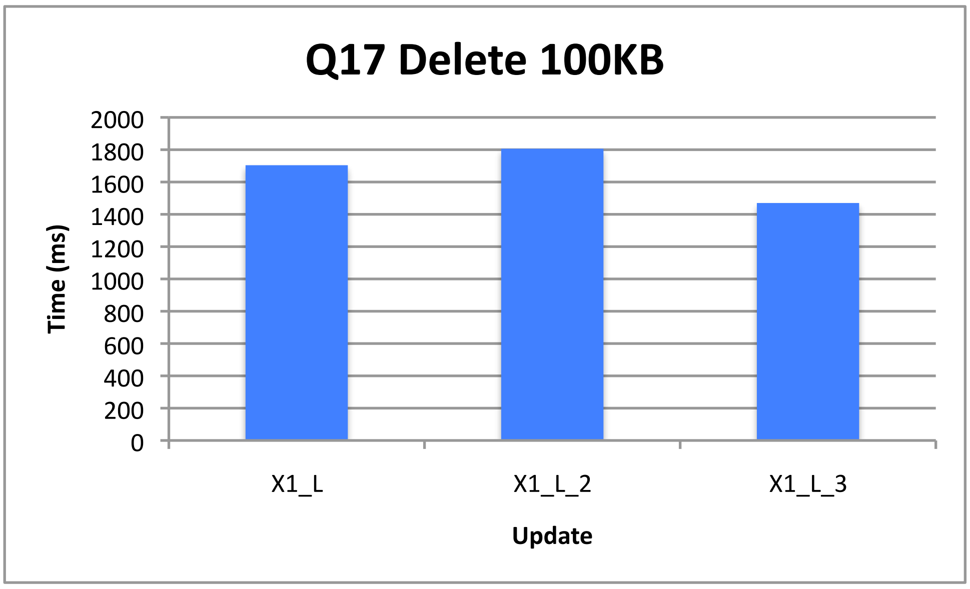
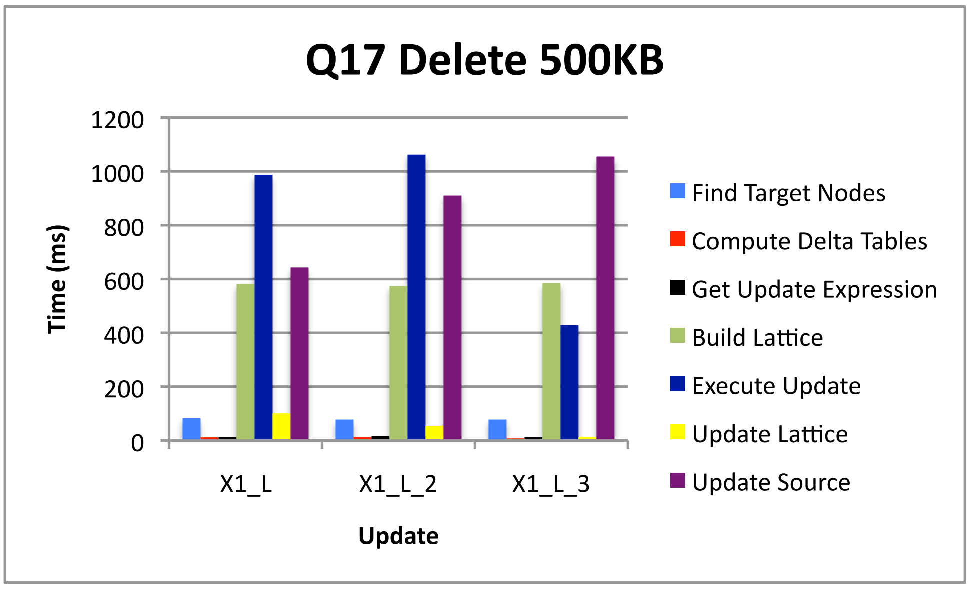


Figure C.109: Q17 Delete Graph 100KB

Query	TN	DT	UE	BL	EU	UL	US
DeleteX1_L	83	12	14	581	987	101	643
DeleteX1_L_2	78	13	16	574	1062	55	910
DeleteX1_L_3	78	8	14	585	429	13	1055

Table C.110: Q17 View Delete 500Kb



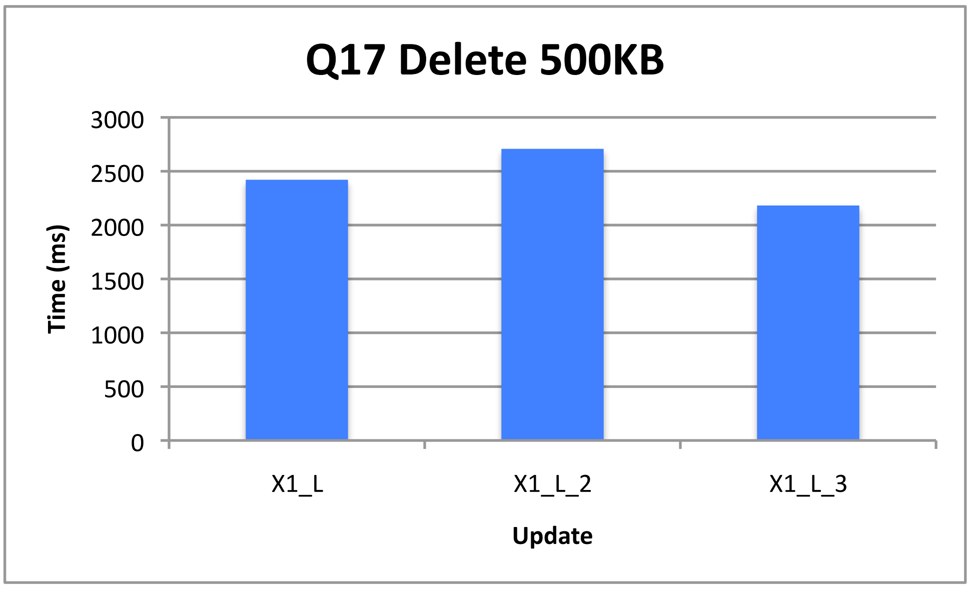
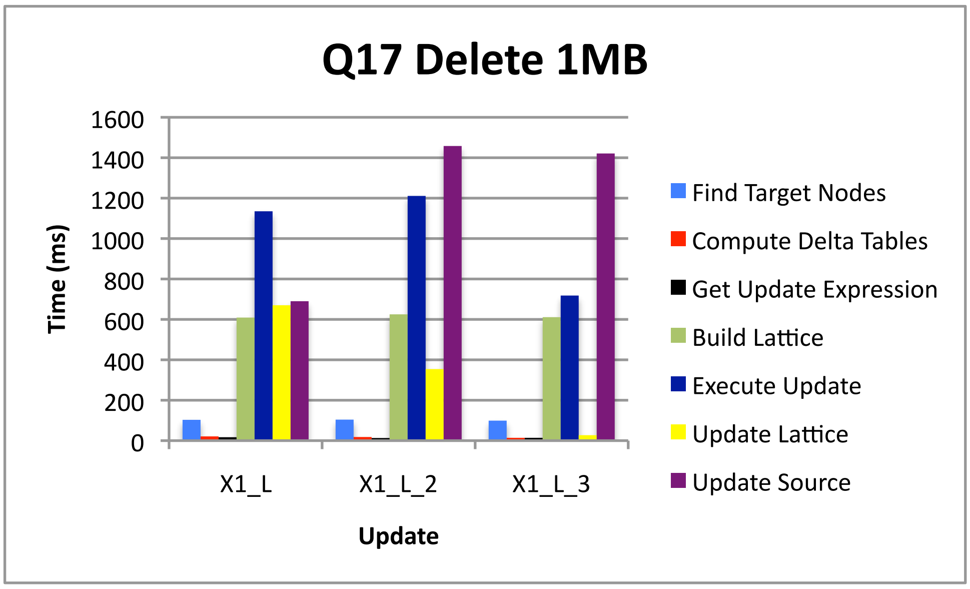


FIGURE C.110: Q17 Delete Graph 500KB

Query	TN	DT	UE	BL	EU	UL	US
DeleteX1_L	103	21	17	609	1135	670	690
DeleteX1_L_2	104	18	13	625	1211	354	1458
DeleteX1_L_3	99	14	14	611	718	27	1421

TABLE C.111: Q17 View Delete 1Mb



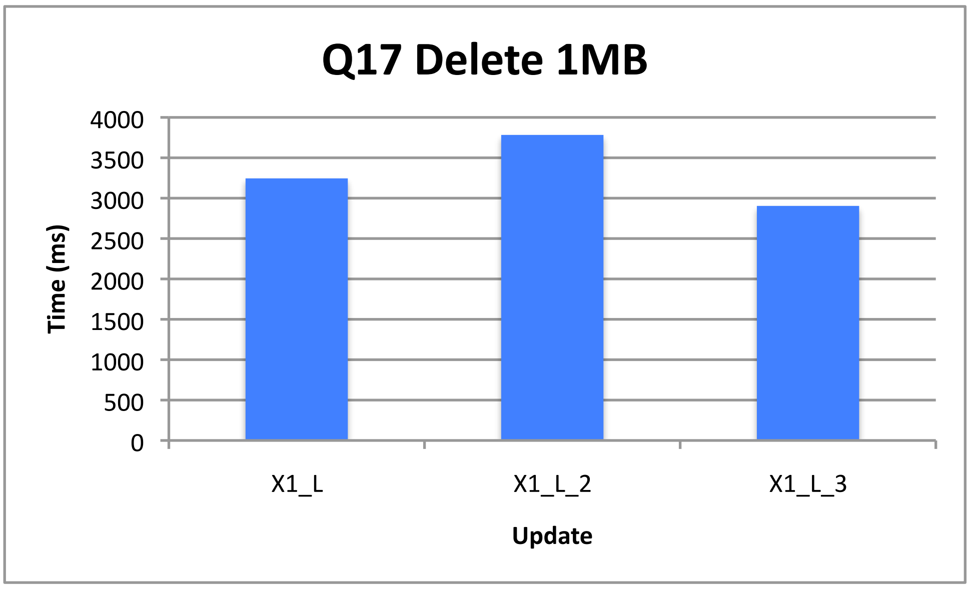
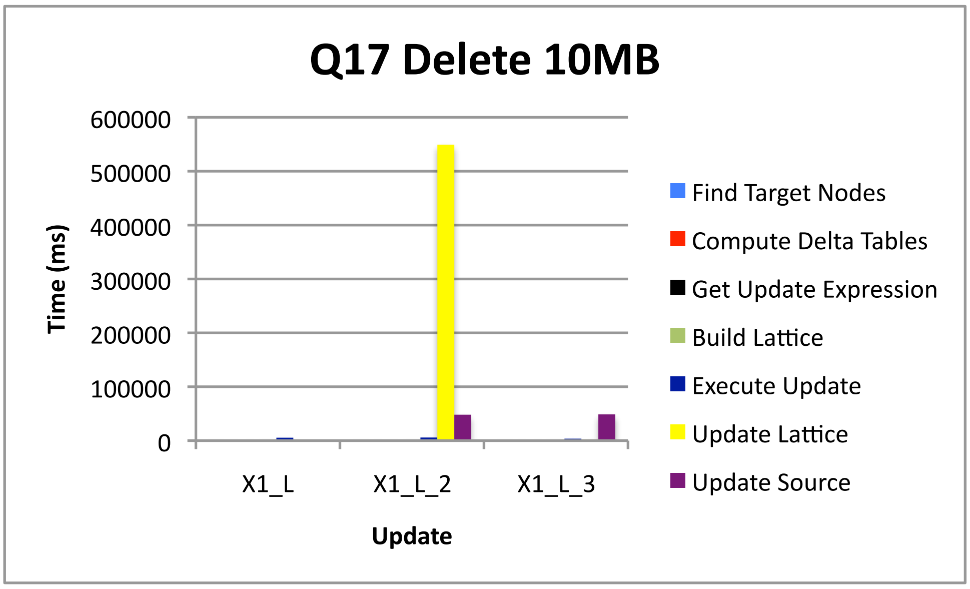


Figure C.111: Q17 Delete Graph 1MB

Query	TN	DT	UE	BL	EU	UL	US
DeleteX1_L	349	125	18	2095	5498	0	0
DeleteX1_L_2	332	116	22	2064	5820	549155	48164
DeleteX1_L_3	329	93	29	2075	3711	2434	48718

Table C.112: Q17 View Delete 10Mb



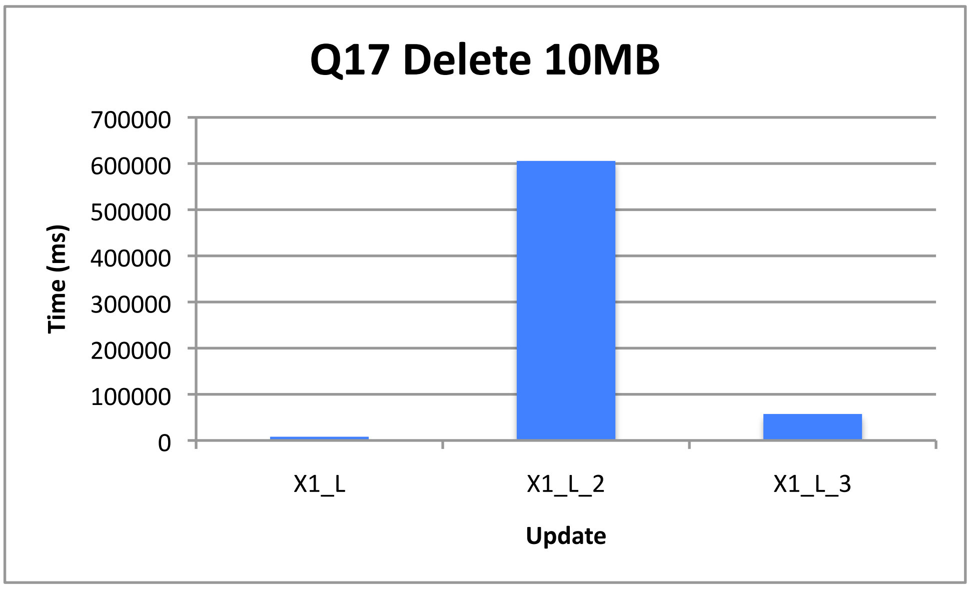


FIGURE C.112: Q17 Delete Graph 10MB