University of Pennsylvania
ScholarlyCommons

Departmental Papers (CIS) | Department of Computer & Information Science

November 2002

XKvalidator: A Constraint Validator For XML

Yi Chen
University of Pennsylvania

Susan B. Davidson
University of Pennsylvania, susan@cis.upenn.edu

Yifeng Zheng
University of Pennsylvania

Follow this and additional works at: http://repository.upenn.edu/cis_papers

Recommended Citation
Yi Chen, Susan B. Davidson, and Yifeng Zheng, "XKvalidator: A Constraint Validator For XML", . November 2002.

Copyright ACM, 2002. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution.
The definitive version was published in *Proceedings of the 11th International Conference on Information and Knowledge Management 2002*, pages 446-452.
Publisher URL: http://doi.acm.org/10.1145/584792.584866

This paper is posted at ScholarlyCommons. http://repository.upenn.edu/cis_papers/122
For more information, please contact repository@pobox.upenn.edu.

XKvalidator: A Constraint Validator For XML

Abstract
The role of XML in data exchange is evolving from one of merely conveying the structure of data to one that also conveys its semantics. In particular, several proposals for key and foreign key constraints have recently appeared, and aspects of these proposals have been adopted within XMLSchema. In this paper, we examine the problem of checking keys and foreign keys in XML documents using a validator based on SAX. The algorithm relies on an indexing technique based on the paths found in key definitions, and can be used for checking the correctness of an entire document (bulk checking) as well as for checking updates as they are made to the document (incremental checking). The asymptotic performance of the algorithm is linear in the size of the document or update. Furthermore, experimental results demonstrate reasonable performance.

Keywords
XML, constraints, key validation, finite state machine

Comments
Copyright ACM, 2002. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in *Proceedings of the 11th International Conference on Information and Knowledge Management 2002*, pages 446-452.
Publisher URL: http://doi.acm.org/10.1145/584792.584866

This conference paper is available at ScholarlyCommons: http://repository.upenn.edu/cis_papers/122

XKvalidator: A Constraint Validator For XML

Yi Chen, Susan B. Davidson and Yifeng Zheng
Dept. of Computer and Information Science
University of Pennsylvania
yicn@saul.cis.upenn.edu, *susan@cis.upenn.edu*, *yifeng@saul.cis.upenn.edu*

ABSTRACT

The role of XML in data exchange is evolving from one of merely conveying the structure of data to one that also conveys its semantics. In particular, several proposals for key and foreign key constraints have recently appeared, and aspects of these proposals have been adopted within XMLSchema.

In this paper, we examine the problem of checking keys and foreign keys in XML documents using a validator based on SAX. The algorithm relies on an indexing technique based on the paths found in key definitions, and can be used for checking the correctness of an entire document (bulk checking) as well as for checking updates as they are made to the document (incremental checking). The asymptotic performance of the algorithm is linear in the size of the document or update. Furthermore, experimental results demonstrate reasonable performance.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Textual Databases

General Terms

Design

Keywords

XML,constraints, key validation, finite state machine

1. INTRODUCTION

Keys are an essential aspect of database design, and give the ability to identify a piece of data in an unambiguous way. They can be used to describe the correctness of data (constraints), to reference data (foreign keys), and to update data unambiguously.

The importance of keys for XML has recently been recognized, and several definitions have been introduced [1]. Aspects of these proposals have found their way into XMLSchema by the addition of UNIQUE and KEY constraints [2]. These proposals overcome a number of problems with the older notion of "ID" (and "IDREF"): First, IDs are like oids (pointers) and carry no real meaning in their value. In contrast, a key is value-based, and is not restricted to be unary (a single attribute); it may be a *set* of attributes or text nodes. Second, IDs must be globally unique. In contrast, a key can have a scope within a sub-document; for example, a student ID can be used to identify students within the context of a single university instead within the entire document.

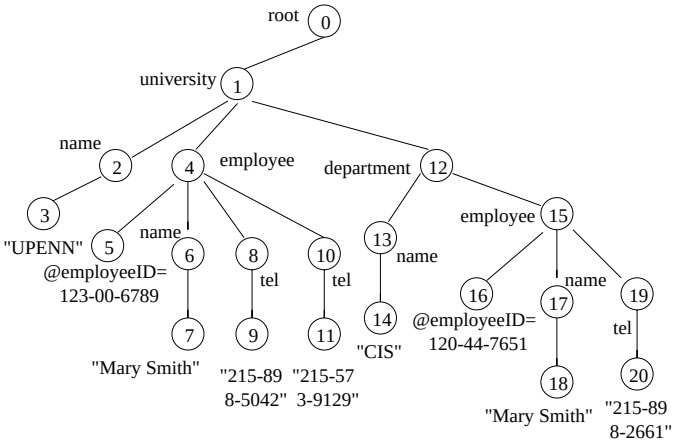

Figure 1: Tree representation of universities.xml

As an example of keys of XML, consider the sample document "universities.xml" represented in tree form in Figure 1. The document describes a set of universities, each of which has a set of departments. Employees can either work directly for the university or within a department. We might wish to state that the key of a university is its name. We might also wish to state that within a university an employee can be uniquely identified by her/his employeeID attribute. (The symbol @ in Figure 1 denotes that employeeID is an attribute.) Another key for an employee might be her/his telephone number (which can be a set of numbers) together with name, within the context of the whole repository.

Although key definitions are being adopted for XML, there is as yet no standard technique for validating that a document conforms to a set of XML keys. Building an efficient validator for key constraints entails a number of challenges: First, unlike relational databases, keys are not localized but may be spread over a large part of the document. In our example, since university elements occur at a top level of nesting, the names of universities will be widely separated

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CIKM'02, November 4–9, 2002, McLean, Virginia, USA.
Copyright 2002 ACM 1-58113-492-4/02/0011 ...$5.00.

in the document. Second, keys can be defined within a particular context. In our example, employees are identified by their employeeID within the scope of a university. Third, an element may be keyed by more than one key constraint or may appear at different levels in the document (as with employees). Fourth, the validator should be incremental. That is, if an XML document has already been validated and an update occurs, it should be possible to validate just the update rather than the entire updated document, assuming that key information about the XML document is maintained.

In this paper, we present a SAX-based validator for XML keys called *XKvalidator*. Our validator is based on a persistent data structure called the *key index*, and techniques to efficiently recognize the paths present in XML keys. Although other XMLSchema validators exist, they do not appear to support XMLSchema KEY and UNIQUE constraints. In particular, Microsoft XML Parser 4.0(MSXML)[3], an XMLSchema validating parser, does not support regular expressions which are essential in defining keys. The University of Edinburgh also has an on-going schema validator project called XSV, but does not appear to have fully implemented XMLSchema keys [4]. The XKvalidator presented in this paper also differs from these XMLSchema validators by supporting a broader definition of XML keys than that given in [2] and by allowing incremental updates to the XML document.

The contributions of this work are as following:

1. An XML constraint validator, which can be used for XMLSchema KEY and UNIQUE constraints as well as for those in [1].
2. Bulk loading and incremental checking algorithms with complexity that is proportional to the size of the affected context (assuming a fixed number of keys are currently activate), hence is near optimal.
3. Experimental results showing the performance of the algorithms.

The rest of the paper is organized as follows: Section 2 introduces a definition of keys and presents our XML constraint validator. Section 3 presents experimental results showing the trade-off between our approach and one based on relational technology. Section 4 surveys related work, and concludes with a summary of contributions and discussion of future work.

2. XML KEYS AND THE XKVALIDATOR

In defining a key for XML we specify three things: the context in which the key must hold, a set on which we are defining a key, and the values which distinguish each element of the set. Since we are working with hierarchical data, specifying the context, target, and key involve path expressions.

Using the syntax of [1][1] a key can be written as

$$(Q,\ (Q',\ \{P_1, \ldots, P_p\}))$$

where Q, Q', and $P_1, \ldots, P_p$ are path expressions. Q is called the *context path*, Q' the *target path*, and $P_1, ..., P_p$ the *key paths*. The idea is that the context path Q identifies a set of *context nodes*; for each context node c, the key constraint must hold on the set of *target nodes* reachable from c via Q'. The *key values* are identified by $P_1, \ldots, P_p$.

For example, using XPath notation for paths, the keys of Section 1 can be written as:

$KS_1 = (/, (./university, \{./name\}))$: Within the context of the whole document ("/" denotes the empty path from the root), a university is identified by its name.

$KS_2 = (/university,$
$(.//employee, \{./@employeeID\}))$:
Within a university an employee can be uniquely identified by his/her employeeID (".//" refers to any sequence of labels).

$KS_3 = (/, (.//employee, \{./name, ./tel\}))$: Within the context of the whole document, an employee can be identified by his/her name and set of telephone numbers.

Definition 2.1: An XML tree T is said to *satisfy* a key if and only if for each context node c and for any target nodes t_1, t_2 reachable from c via Q', whenever there is a non-empty intersection of values for each key path $P_1, \ldots, P_p$ from t_1, t_2,then t_1 and t_2 must be the same node. ■

For example, KS_3 is satisfied in the XML tree of Figure 1 since employee *123-00-6789* and *120-44-7651* are both within the context of the same university (*UPENN*), and although they share the same name (*Mary Smith*) they do not share any telephone number. That is, although there is a non-empty intersection on the first key path $./name$, there is an empty intersection on the second key path $./tel$. The key would also hold if we eliminated the telephone number for the second Mary Smith (*120-44-7651*) since $\{215-898-5042,\ 215-573-9129\} \cap \emptyset = \emptyset$. However, KS_3 would not hold if the second Mary Smith (*120-44-7651*) were given another phone number of *215-898-5042*, since there would be a non-empty intersection on both key paths (./name and ./tel) with the first Mary Smith (*123-00-6789*).

Although in these examples the key values are all sets of elements of type string (text), in general the key values may be sets of XML trees. In this case, the notion of equality used to compute set intersection must be extended to one of tree equality[1].

Note that the class of keys supported in XMLSchema is a subset of the ones defined here. In XMLSchema, key values are restricted to be text values and must be be unique. In contrast, we allow key values to be attributes or elements, and allow them to occur several times. For example, in KS_3 we allowed a phone (element) to be a component of a key for person, and its value is set-valued.

The XMLSchema UNIQUE constraint, which states that within a particular context a label occurs at most once, can also be captured by a key of form $(Q, (Q', \{\}))$. To see this, suppose that for some context node c reachable from the root via Q (the context) there are two target nodes t_1, t_2 reachable from c via the label Q'. Since there are no key paths to distinguish them, they must be the same node.

Following XMLSchema, we use a restriction of XPath to define paths in keys. This restriction allows navigation along the child axis, disjunction at the top level, and wildcards in paths. This path language can be expressed as follows:

$$c \ ::= \ . \ | \ / \ | \ .q \ | \ /q \ | \ .//q \ | \ c|c$$

$$q \ ::= \ l \ | \ q/q \ | \ _$$

[1]We adopt this because it is more concise than that of XMLSchema.

Key	*Context Node Id*	*Key Path*	*Key Value*	*KV SC*
KS_1	0	name	UPENN	{1}
KS_2	1	emplo-	123-00-6789	{4}
		yee id	120-44-7651	{15}
KS_3	0	name	Mary Smith	{4,15}
		tel	215-898-5042	{4}
			215-573-9129	{4}
			215-898-2661	{15}

Figure 2: Key index for universities.xml

Here "/" denotes the root or is used to concatenate two path expressions, "." denotes the current context, l is an element tag or attribute name, "_" matches a single label, and "//" matches zero or more labels out of the root.

Note that just using key KS_2, we are not able to uniquely identify an *employee* node by its *employeeID*. That is, KS_2 is scoped within the context of a *university* node rather than within the scope of the root of the XML tree. However, given a key for the context node of KS_2, i.e. the *name* of a *university* (KS_1), we can then identify an *employee* node by its *employeeID*. The ability to recursively define context nodes up to the root of the tree is called a *transitive* set of keys [1]: $\{KS_1, KS_2\}$ is a transitive set of keys, as is $\{KS_3\}$ since its KS_3's context is already the root of the tree.

2.1 The XML Key Index

The XML constraint validator is based on a key index, which can be thought of in levels. The top level is the *key specification level*, which partitions target nodes in the XML tree according to their key specifications. Since a target node may match more than one key specification, it may appear in more than one partition. The second level is the *context level*, which further groups target nodes by their context. The third level is the *key path level*, which groups target nodes based on key paths. The fourth level is the *key value level*, which groups target nodes by equivalence classes called *key value sharing classes* (*KVSC*). The KVSCs are defined such that the nodes in a class have some key nodes which are value-equivalent, following the same key path under the same context in a particular key. If key values are arbitrary XML trees, we store their serialized value (see [5] for details).

For example, the index structure for KS_1, KS_2 and KS_3 on the XML data in Figure 1 is shown in Figure 2. Note that nodes 4 and 15 are each keyed by KS_2 and KS_3, and that they share the same *name* value.

Given a new target node t within a context node c of an XML key K, the validator checks if t shares some key value with another target node under c for every key path. That is, for each key path P_i ($1 \leq i \leq p$) of K, it unions all the KVSCs that t belongs to and produces the set of nodes S_i that share some key value with t. It then computes the intersection of the S_i's of each key path, $S = S_1 \cap ... \cap S_p$, which is the set of nodes that share some key value with t for all the key paths. If S contains more than one node (t), then K is violated.

For example, suppose we were validating the XML document of Figure 1 with respect to KS_1, KS_2 and KS_3. To check KS_3, as we parse through node 18 we find that the KVSC for *Mary Smith* is {4, 15}. As we continue the parse through node 20 we find that the KVSC for *215-898-2661* is {15}. Finishing the parse of the substructures of node 15, we check that $\{4, 15\} \cap \{15\} = \{15\}$, and so the constraint is valid.

Although the primary purpose of the index is to efficiently check keys, it can also be used to find a node using a transitive set of keys. This property will be used later when we talk about updating XML trees by specifying an update node.

Example 2.1: Suppose we want to find the *employee* whose *employeeID* is *120-44-7651* at *UPENN*. Since $\{KS_1, KS_2\}$ is a transitive set of keys, the query to locate the *employee* must specify a key for each context node. Here we use XQuery [6] for syntax.

```
<result>
{
for $a in document("universities.xml")/university
    $b in $a//employee
where boolean-and($a/name = "UPENN" ,
    $b/@employeeID = "120-44-7651" )
return $b
}
</result>
```

From the index of KS_1 in Figure 2, we know that *name* is a key of *university* and that the context is the root (node 0). The KVSC of *university* nodes with the key value *"UPENN"* following key path *name* is {1}. Since *@employeeID* is the key path of an *employee* node under the context of a *university* (KS_2), we can get the KVSC of *employee* nodes with the key value *"120-44-7651"* following the key path *@employeeID* under the context node 1. This class contains node 15.

2.2 Index Construction

The index can be constructed in one pass over the XML file using a SAX parser and a set of finite state automatas (DFAs) which recognize the context (Q), target (Q') and key paths ($P_1, \ldots, P_p$) for each XML key K. As the document streams in, each node is assigned a unique internal id. The internal id and tag of each node (the *node info*) is then communicated to the DFAs, which may trigger a state change.

Since a target node can only appear after its context node, $DFA(Q')$ is only created when the accept state of $DFA(Q)$ has been reached. Similarly, $DFA(P_i), i = 1 \ldots p$ are only created when the accept state of $DFA(Q')$ has been reached. Since the context and target path expressions are regular expressions and may contain .//, several context nodes for one key specification and several target nodes for one context node can be activate at the same time.

DFAs have two final states: an *accept* state which signifies that the path has been found; and a *terminate* state which ends the execution of the DFA. When the accept state of the DFA for a context path is reached, the id of the context node is added to the index. When the accept state for the DFA of a key path is reached, we store the key value in the index. When the terminate state of the DFA for a target path is reached, the process to check satisfaction of the key specification is invoked (see Section 2.1); if the key is not satisfied, the index is rolled back to the state it was in initially.

Several optimizations are used: Since constructing the DFA from a regular expression is very time-consuming [7], DFAs are constructed once and only as needed. Rather than terminating a DFA, it is inactivated until next used. Furthermore, since it is common for several DFAs for the same path to be active at the same time, we create one DFA with a set of current states; each current state represents the progress of one of the active DFAs.

2.3 Incremental Maintenance

To describe how XKvalidator handles updates, we focus our attention on two basic tree operations: the insertion of a new tree below an update node, and the deletion of the tree below an update node. These updates are specified as $insert(n, T_u)$ and $delete(n)$, where n is the internal id of the node to be updated and T_u is a tree to be inserted.

For example, an update to *universities.xml* which gives the employee node 15 another telephone number "215-898-5042" could be written as $insert(15, T_u)$, where the content of T_u is `<tel> 215-898-5042 </tel>`. Node 15 could also be identified by a transitive set of key values as shown in Example 2.1.

Note that the XML standard for updates, XMLUpdate, currently includes many other operations, including specifying order in insertion, append, update and rename [8]. These operations could be handled within our framework, however limiting the updates considered simplifies the discussion.

Since our index is hierarchical, updates may affect the index at different levels. We can divide them into four cases by the effect of this update:

1. Entries at the context level are inserted or deleted.
2. One or more target nodes along with their key values are inserted or deleted.
3. One or more key value(s) of an existing target node are inserted or deleted.
4. The key value is changed.

Note that case 4 can only occur when the key value is a tree instead of a text node, and we are inserting or deleting a subtree of a key node under an existing target node.

It is clear that since insertions introduce new values, the index must be maintained whenever the insertion interacts with the context, target or key path of some key. Deletion is more surprising: Although deletions in relational databases can never violate a key constraint, in the context of XML they may change some key value. Therefore the index must be maintained whenever a deletion interacts with some key path (case 4 above).

For example, consider a modified version of the tree in Figure 1 in which the *name* of *employee* node has two children: *firstname* and *lastname* (e.g. node 6 with label *name* has a *firstname* node with value *Mary* and a *lastname* node with value *Smith*). If we delete the *firstname*, then the key value of the *employee* node 4 in KS_3 will be changed and the key constraint needs to be checked.

The next question is how to determine when an update "interacts" with a context, target or key path expression. This can be done by reasoning about the concatenation of labels from the root to the update node n, and the paths Q, $Q.Q'$ and $Q.Q'.P_j$. Details can be found in the technical report [5].

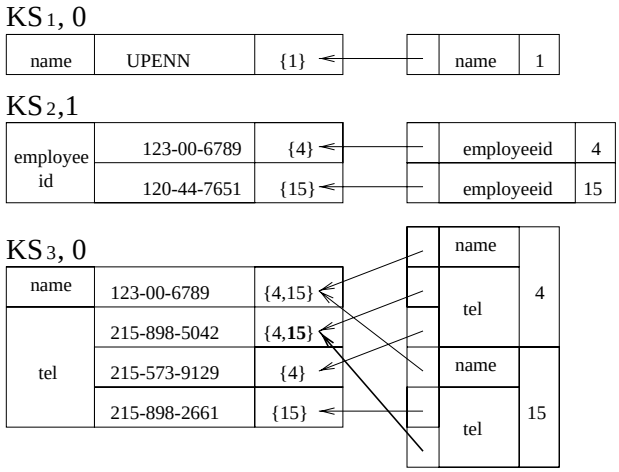

Figure 3: Updated key index example

It turns out that when a new key value for an existing node is inserted into the index (cases 3 and 4 above), key checking using only the key index presented so far is very inefficient since it entails retrieving all the key values of the updated node. We therefore build an auxiliary index on the key index to retrieve these key values efficiently, which indexes each target nodes under their context node. Figure 3 illustrates this with the key index on the left and the auxiliary index on the right.[2] In the auxiliary index, for each key path P_j a pointer is kept to the key values for the target node. For example, the first entry in the auxiliary index of Figure 3 points to the *name* path value for target node 1.

Example 2.2: Consider the insertion of a telephone number *215-898-5042* under the *employee* with *employeeID =120-44-7651* within the *university* whose *name= UPENN* (i.e. under node 15).

It is easy to see that this update does not affect key specifications KS_1 and KS_2. It does, however, affect KS_3 by inserting a new key value (case 3). Processing the insertion will result in the index structure of Figure 3, where a *tel* pointer for node 15 is inserted in the auxiliary index structure, and element 15 is inserted into the KVSC for tel 215-898-5042(indicated in bold). Following the pointers for node 15 in the auxiliary index structure, we can find all the KVSCs it belongs to: the KVSC for *Mary Smith* ({4, 15}), the KVSC for *215-898-2661* ({15}), and the KVSC for *215-898-5042* ({4, 15}). To check KS_3, we union the two KVSCs for key path *tel* and get a set {4, 15}. When we intersect this with the KVSC for key path *name* we get a conflicting node set, {4, 15}. Since a violation is discovered, the update is rolled back.

3. EXPERIMENTAL RESULTS

Since there are no other XML validators that can check KEY and UNIQUE constraints as defined in XMLSchema,[3] we evaluate the performance of our validator against us-

[2]To save space, in the key index the key name and context node id, e.g. "$KS_1, 0$" appear above the remaining levels of the index (c.f. Figure 2).

[3]Microsoft XML Parser 4.0(MSXML)[3] checks KEY and UNIQUE constraints, but does not currently support regular expressions.

ing a relational database for constraint checking. Specifically, we store XML documents in a commercial relational database system using hybrid inlining [9].[4] and handcode the key constraints. Given the relational schema produced by hybrid inlining, the key constraints are optimized to use UNIQUE/PRIMARY key features whenever possible; when UNIQUE/PRIMARY key features cannot be used indices are set up to optimize the checking performed by stored procedures. All experiments run on the same 1.5GHz Pentium 4 machine with 512MB memory and one hard disk with 7200rpm. The operating system is Windows 2000, and the DBMS is DB2 universal version 7.2 using the high-performance storage option. We use Java 2 to code the evaluator, and JDBC to connect to the database.

3.1 Data set and keys

We use a synthetic data set generated by an XML Generator from the XML Benchmark project [10]. (We also ran experiments on real data sets, EMBL [11]. Since the results were similar, we omit them.) XML Generator was modified to generate a series of XML files of different sizes, according to DTD shown in Figure 4. Using hybrid-inlining, the following relational tables are created:

- *University(uID, name)*
- *School(sID, name, parentID)*
- *Department(dID, name, parentID, parentCode)*
- *ResearchGroup(rID, name, parentID)*
- *Employee(eID, name, employeeID, parentID, parentCode).*

In these tables, *uID*, *sID*, *dID*, *rID* and *eID* are internally generated keys (sysid's), and *parentID*, *parentCode* are used to record the owning tuple information (i.e. *parentCode* is the name of the relation the owning tuple is stored in and *parentID* the key of the owning tuple).

The keys to be validated are as follows:

$KS_4 = (/, (./university, \{./name\}))$: Each *university* is identified by its *name*.

$KS_5 = (/university, (.//department, \{./name\}))$: Within a *university*, each *department* is identified by its *name*.

$KS_6 = (/university, (.//employee, \{./employeeID\}))$: Within a *university*, at whatever level they occur, each *employee* is uniquely identified by his/her *employeeID*.

To check KS_4, we specify attribute *name* to be the primary key for the *university* table. To check KS_5, we need to join *school* with *department* whose parent is *school* and union it with the *department* whose parent is *university* to get all possible *(university.uid, department.name)* pairs, and then check if there are any duplicates. Checking KS_6 is similar to KS_5 except more joins and unions are needed to get *(university.uid, employeeID)* pairs. Indices on *(parentCode, parentID)* or *(parentID)* are built on every table where applicable. To speed up key checking, we also build index (*name*, *parentCode*, *parentID*) on *Department*, and (*employeeID*, *parentCode*, *parentID*) on *Employee*.

[4]We omit experiments using shared inlining and edge mapping since hybrid inlining offers better performance.

```
<!ELEMENT db(university*)>
<!ELEMENT university(name,school*,
          department*,employee*)>
<!ELEMENT school(name, department*,
          employee*)>
<!ELEMENT department(name,
          researchgroup*,employee*)>
<!ELEMENT researchgroup(name,
          employee*)>
<!ELEMENT employee(name, employeeID)>
<!ELEMENT name(#PCDATA)>
```

Figure 4: DTD of universities.xml

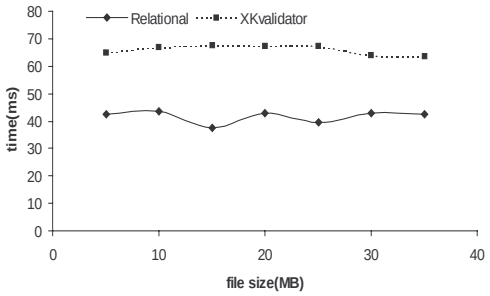

Figure 5: Time to incrementally check KS_4

3.2 Experiments

We model incremental updates by inserting a delta XML document of size 100KB into XML documents of different sizes. For XKvalidator, the time measured is the time spent updating the key indices and performing the key check condition; it does not include the parse time of the delta XML document. It should also be noted that although XKvalidator is designed to check very large documents and therefore uses buffering to persistent storage as needed, the size of the XKvalidator indices in this experiment were small enough to fit in main memory. For the relational case, the delta XML document is parsed to produce an update set of tuples. The time measured is only the time to check the inserted tuples; it does not include the time to produce the tuples nor the time to insert the tuples. This is calculated by taking the taking the difference of the time to insert the update set with constraints turned on and the time to insert the update set with constraints turned off.

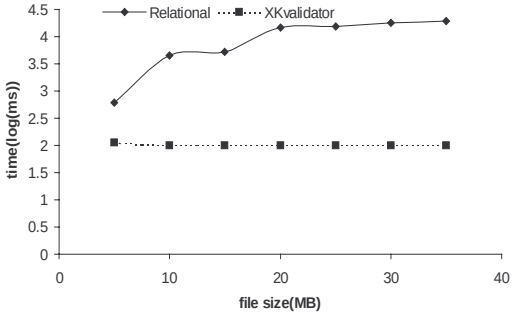

Figure 6: Time to incrementally check KS_5

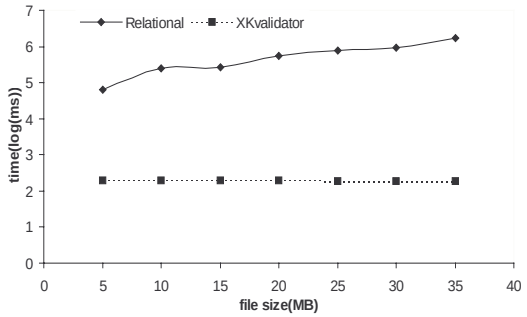

Figure 7: Time to incrementally check KS_6

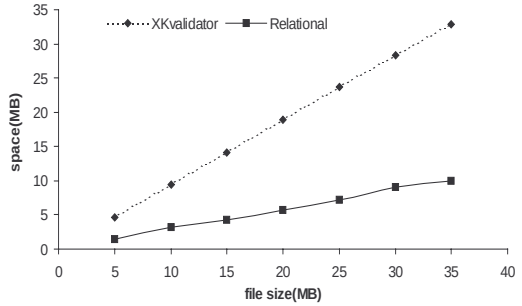

Figure 8: Index size for KS_6

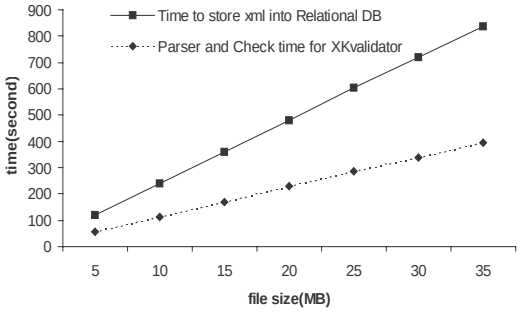

Figure 9: Time to store XML document in RDBMS vs. XKvalidator

The performance of the relational approach versus XKvalidator using a series of files of increasing sizes are shown in Figure 5 for KS_4, 6 for KS_5, and 7 for KS_6. Note that the time for XKvalidator is roughly constant since the update size is constant, and that a log scale is used for the Y axis in Figures 5 and 6. Figure 5 shows that XKvalidator performs roughly on a par with relational technology for XML keys that can be mapped into relational keys. However, Figures 6 and 7 show that XKvalidator drammatically outperforms relational technology when XML keys are mapped into stored procedures.

A comparison of the XKvalidator key index size versus that of relational indices specifically designed for checking KS_6 is shown in Figure 8; results for KS_5 are similar. Our index is somewhat larger than the relational indices, however, XKvalidator is not currently optimized for space.

Figure 9 compares the time to store an XML document in a relational database against the total time to run XKvalidator. In the relational case, the time measured is the time to insert all tuples generated from the XML document; for XKvalidator it is the time to parse the document as well as to create and check the index structures.

A number of conclusions can be drawn from these results: First, XKvalidator is an effective technique for checking key constraints in XML documents since it is roughly on a par in terms of time and only slightly larger in terms of space as compared with relational primary key technology. Second, checking XML keys using relational technology is most effective if they can be mapped to primary keys. Since hybrid inlining and other relational schema design techniques do not take keys into account, they do not produce schemas in which keys can be effectively checked. Third, unless an RDBMS is already being used as the storage strategy for the XML document, it is not worth creating one just to check keys since the time needed just to store the XML document in the relational database is much larger than the total time for XKvalidator.

4. RELATED WORK AND CONCLUSIONS

In this paper we present an XML key constraint validator based on SAX. XKvalidator considers a broad class of XML keys, in which the value of keys may be XML trees rather than simple text and key paths can be set valued, which subsumes those definable in XMLSchema. Our XML key constraint validator can be used both for bulk-loading (i.e. one pass over the entire document) as well as for incremental checking (i.e. XML updates to the document can be processed and checked against a persistent key index for the file). XKvalidator can also be used with a little modification to check referential integrity in XMLSchema (KEYREF), since it already provides the ability to find a node according to its key value.

There are several other XMLSchema checkers and validators: IBM's XML-Schema-Quality-Checker [12] takes as input an XMLSchema and diagnoses improper uses of the schema language. However, it is not a validating parser, that is, it does not take as input a document and validate it against the schema. Microsoft XML Parser 4.0(MSXML)[3] is a validating parser, but does not currently support regular expressions. The University of Edinburgh also has an on-going schema validator project called XSV, but does not appear to have implemented XMLSchema keys [4].

The salient differences between the approach taken in

these XMLSchema key validators and the one suggested in this paper are as follows. First, our definition of XML keys follows that of [1] which is more general than that given in XMLSchema. However, our key checker can easily be used to validate XMLSchema keys. Second, we have designed an incremental validation algorithm which verifies updates to an XML document. Other approaches are designed to parse the entire updated XML file to check the key constraints.

At the heart of XKvalidator is the key index introduced in Section 2. Compared with other XML index structures [13, 14, 15, 16], the index captures both the structure and the content information of the data. A query evaluator can therefore use this index together with information about path restriction and value conditions to optimize queries on keys. Preliminary results shows that our key index performs better than that of [16] for queries involving key look-ups.

Another approach to validating XML key constraints is to use relational technology as demonstrated in Section 3. Our experiments show that the performance of XKvalidator is roughly the same as PRIMARY KEY/UNIQUE checks in a relational database. However, XKvalidator performs better by several orders of magnitude when the key checks use complex stored procedures. In future work we plan to develop relational storage techniques for XML documents which take XML key and foreign key constraints into account, and produce relational schemas in which such constraints can be checked using relational key and foreign key constraints [17].

5. REFERENCES

[1] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Keys for XML. In *WWW10*, pages 201–210, 2001.

[2] H. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML Schema Part 0: Primer , May 2001. http://www.w3.org/TR/xmlschema-0/.

[3] Microsoft XML Parser 4.0(MSXML). Available at: http://msdn.microsoft.com.

[4] XML Schema Validator. Available at: http://www.ltg.ed.ac.uk/ ht/xsv-status.html.

[5] Y.Chen, S. Davidson, and Y. Zheng. Indexing keys in hierarchical data. Technical Report MS-CIS-01-30, University of Pennsylvania, Computer and Information Science Department, 2001.

[6] XQuery 1.0: An XML query language, June 2001. http://www.w3.org/XML/Query.

[7] J. Hopcroft and J. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, 1979.

[8] XUpdate. http://www.xmldb.org/xupdate/xupdate-wd.html.

[9] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *The VLDB Journal*, pages 302–314, 1999.

[10] XMARK the XML-benchmark project, April 2001. http://monetdb.cwi.nl/xml/index.html.

[11] D. G. Higgins, R. Fuchs, P. J. Stoehr, and G. N. Cameron. The EMBL data library. *Nucleic Acids Research*, 20:2071–2074, 1992.

[12] XML Schema Quality Checker, February 2002. Available at: http://www.alphaworks.ibm.com/tech/xmlsqc.

[13] J. McHugh, J. Widom, S. Abiteboul, Q. Luo, and A. Rajaraman. Indexing semistructured data. Technical report, Stanford University, Computer Science Department, 1998.

[14] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, 1999.

[15] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for efficient indexing of paths in graph structured data. In *ICDE*, 2002.

[16] Quanzhong Li and Bongki Moon. Indexing and querying XML data for regular path expressions. In *The VLDB Journal*, pages 361–370, 2001.

[17] Y. Chen, S. B. Davidson, and Y. Zheng. Constraint Preserving XML Storage in Relations. In *WebDB*, 2002.