Query them by the following query which not strongly-specified since it does not satisfy condition (*ii*) of Definition 4.6

$$\texttt{self} :: a[\texttt{child} :: \texttt{node}].$$

$\{X,Y\}$ is an optimal projector for this query, but the presence of the condition `self :: node` makes the system to include also W in the inferred projector, thus breaking completeness. Concerning the presence of backward axes in predicates, consider the query `self` :: a[`descendant :: node/ancestor ::` a] which does not satisfy condition (*i*). An optimal projector for this query on the same DTD is $\{X,Y\}$. However, since the `ancestor` condition is true for all descendants of a nodes, $\{W,Z\}$ is included in the projector as well. Finally, it is straightforward to check that the query `self` :: a[`child` :: b `or child` :: c], which does not satisfy condition (*iii*), is not complete for the same DTD.

Of course, it is possible to state completeness for other classes of queries but, once more, this seems an excellent compromise between simplicity and generality.

THEOREM 4.8 (DECIDABILITY). *Given a path P, a* DTD *E, and an environment* Σ *well-formed with respect to E, the inference of a context* Σ' *and a type* τ *such that* $\Sigma \vdash_E P : \Sigma'$ *and* $\Sigma \Vdash_E P : \tau$ *is decidable.*

4.3 Adding sibling, preceding and following axes.

We could deal with the missing XPath axes by adding specific inference rules. Instead we opt to use an approximation of these axes in term of the previous ones, since it appears as the best compromise between simplicity and efficiency.

The approximation is performed by two logical rewriting passes. In the first pass we rewrite preceding and following axes as specified in the W3C specifications [4]. Namely, we substitute each step *Axis* :: *Test* with *Axis* $\in$ {`preceding`,`following`} by the following equivalent path `ancestor-or-self :: node/(`*Axis*`-sibling) :: node/descendant-or-self ::` *Test*

The second pass is the one which introduces the approximation since it replaces all steps of the form *Axis*::*Test* with *Axis* $\in$ {`preceding-sibling`,`following-sibling`} by the path `parent::node/child::`*Test*.

Clearly, the static analysis of the approximation yields a less precise projection than the one we could obtain by working directly on the original query. However, we still achieve good precision of pruning in practice as we will show in Section 6. For instance, by applying the above rewriting to XPathMark queries Q9 and Q11, we were able to prune a document down to 7.5% of its original size.

5. EXTENSION TO XQUERY

In this section we extend the technique to XQuery. More precisely to the FLWR core of XQuery described by the following grammar:

```
q  ::=  () | q,q | <tag>q</tag> | Exp
     |  for x in q return q | let x = q return q
     |  if q then q else q
```

where the definition of *Exp* (given in Section 3.3) is extended with variables, and with generic XPath expressions Q of Section 3.3 that can be rooted at a variable or at `/` :

$$Exp ::= x \mid Q \mid x/Q \mid /Q \mid Exp\,op\,Exp \mid f(Exp,..,Exp) \mid AExp$$

Without loss of generality, we assume that FLWR expressions do not occur in `if`-conditions nor in predicates (every query can be put into this form by adding appropriate `let`-expressions). Also, we do not consider either queries which first construct new elements and then navigate on them (these are rarely used in practice), nor those containing XQuery clauses like `order_by`, `switch_case`, etc.: our approach can be easily extended to both cases.

In order to apply the previous analysis to infer a projector for q, we first extract a set of XPath$^\ell$ expressions from q, denoting the data needs for q. This set of paths is extracted from the query by the extraction function $\boldsymbol{E}$, whose definition is given in Figure 3. The extraction function has the form $\boldsymbol{E}(q,\Gamma,m)$. The first parameter is the query at issue. The second parameter Γ is an environment that keeps track of bindings of the form (x; for P) or (x; let P), whose scope q is in (see the definition of Γ' in the last two lines of Figure 3, and observe, by a simple induction reasoning, that environments contain paths already in XPath$^\ell$). Finally, m is a flag indicating whether q is a query that serves to materialise a partial or final result ($m=1$), or that just selects a set of nodes whose descendants are not needed ($m=0$). Thus, the set of path expressions (possibly containing qualifiers) extracted from a top-level query q is $\boldsymbol{E}(q,\varnothing,1)$.

Once the set of paths are extracted from a query q, we use it to infer a projector for q according to rules in Section 4.2. Formally, for each P_i extracted from q we deduce a projector π_i, and use for the whole q the union of these projectors (projectors are closed by union). Also, note that the extracted path of a closed query will not contain free variables since possible free variables are persistent roots that must be solved before the analysis.

Most of the rules in Figure 3 are not difficult to understand, therefore only few of them deserve further commentary. The flag is needed since each path determining the result ($m=1$) must be extended with `descendant-or-self`, in order to project on all nodes needed in the query result. This is done by the lines 6, 8, and 10 of the definition. Expressions are dealt in a way similar to the path extractor $\boldsymbol{P}$ of Section 3.3; the extractor $\boldsymbol{P}$ itself is used in line 12 to produce simple paths (where we used the notation `or`$(\{P_1,...,P_n\})$ for P_1 `or` $\ldots$ `or` P_n, and omitted the—straightforward—rules for single step paths). Also note that when a result is computed (lines 2 and 5) paths in "for"-environments are added ("let" are added only if their binding variable is used).

These rules subsume and enhance the whole Marian and Siméon's technique [14]. In particular, (*i*) the technique we use to exclude useless intermediate paths is simpler and more compact, (*ii*) we do not need to distinguish between two kinds of extracted paths but, more simply, we always manage a unique set of path expressions, and (*iii*) last but not least, our path extractor can be used even if the user cannot access an XQuery to XQuery-Core compiler, which is necessary for [14].

Before applying the extraction function $\boldsymbol{E}$ to a query q we apply some heuristics that rewrite q so to improve the pruning capability of the inferred paths. Among these heuristics the most important is the one that rewrites

```
for y in Q/descendant-or-self::node
  return if C(y) then q else ()
```

into

```
for y in
  Q/descendant-or-self::node[C(self::node)]
return q
```

whenever $C(y)$ is a condition referring only to y and does not use external functions (C(`self :: node`) is obtained by replacing `self :: node` for all occurrences of y free in C). If we apply $\boldsymbol{E}$ to the first query, then a path ending by `descendant-or-self::node` is extracted thus annulling further pruning: the entire forest selected

1.	$\boldsymbol{E}((),\Gamma,m)$	=	$\varnothing$
2.	$\boldsymbol{E}(AExp,\Gamma,1)$	=	$\{P \mid (x;\ \text{for}\ P)\in\Gamma\}$
3.	$\boldsymbol{E}(AExp,\Gamma,0)$	=	$\varnothing$
4.	$\boldsymbol{E}((q_1\texttt{,}q_2),\Gamma,m)$	=	$\boldsymbol{E}(q_1,\Gamma,m)\cup\boldsymbol{E}(q_2,\Gamma,m)$
5.	$\boldsymbol{E}(\text{<}tag\text{>}q\text{</}tag\text{>},\Gamma,m)$	=	$\{P \mid (x;\ \text{for}\ P)\in\Gamma\}\cup\boldsymbol{E}(q,\Gamma,1)$
6.	$\boldsymbol{E}(x,\Gamma,1)$	=	$\{P/\texttt{descendant-or-self}::\texttt{node} \mid (x;\ -P)\in\Gamma\}$
7.	$\boldsymbol{E}(x,\Gamma,0)$	=	$\{P \mid (x;\ -P)\in\Gamma\}$
8.	$\boldsymbol{E}(/P,\Gamma,1)$	=	$\{/P/\texttt{descendant-or-self}::\texttt{node}\}$
9.	$\boldsymbol{E}(/P,\Gamma,0)$	=	$\{/P\}$
10.	$\boldsymbol{E}(x/P,\Gamma,1)$	=	$\{P'/P/\texttt{descendant-or-self}::\texttt{node} \mid (x;\ -P')\in\Gamma\}$
11.	$\boldsymbol{E}(Step/q,\Gamma,m)$	=	$Step/\boldsymbol{E}(q,\Gamma,m)$
12.	$\boldsymbol{E}(Step[Exp]/q,\Gamma,m)$	=	$Step[\texttt{or}(\boldsymbol{P}(Exp))]/\boldsymbol{E}(q,\Gamma,m)$
13.	$\boldsymbol{E}(Exp_1\ op\ Exp_2,\Gamma,m)$	=	$\boldsymbol{E}(Exp_1,\Gamma,m)\cup\boldsymbol{E}(Exp_2,\Gamma,m)$
14.	$\boldsymbol{E}(f(Exp_1,\ldots,Exp_n),\Gamma,m)$	=	$\bigcup_{i=1,n}(\boldsymbol{E}(Exp_i,\Gamma,0)/\boldsymbol{F}(f,i))\cup\{\texttt{self}::\texttt{node}\}$
15.	$\boldsymbol{E}(\texttt{if}\ q\ \texttt{then}\ q_1\ \texttt{else}\ q_2,\Gamma,m)$	=	$\boldsymbol{E}(q,\Gamma,0)\cup\boldsymbol{E}(q_1,\Gamma,1)\cup\boldsymbol{E}(q_2,\Gamma,1)\cup\{P \mid (x;\ -P)\in\Gamma\}$
16.	$\boldsymbol{E}(\texttt{for}\ x\ \texttt{in}\ q_1\ \texttt{return}\ q_2,\Gamma,m)$	=	$\boldsymbol{E}(q_1,\Gamma,0)\cup\boldsymbol{E}(q_2,\Gamma\cup\Gamma',m)$ (where $\Gamma'=\{(x;\ \text{for}\ P) \mid P\in\boldsymbol{E}(q_1,\Gamma,0)\}$)
17.	$\boldsymbol{E}(\texttt{let}\ x\ \texttt{=}\ q_1\ \texttt{return}\ q_2,\Gamma,m)$	=	$\boldsymbol{E}(q_1,\Gamma,0)\cup\boldsymbol{E}(q_2,\Gamma\cup\Gamma',m)$ (where $\Gamma'=\{(x;\ \text{let}\ P) \mid P\in\boldsymbol{E}(q_1,\Gamma,0)\}$)

Figure 3: XQuery path extraction

by Q is loaded in main memory. This also happens with the approaches of Bressan *et al.* [9] and of Marian and Siméon [14]. In our and Marian and Siméon's approach the query can be rewritten as above (this is not possible in [9] since their subset of XQuery does not include predicates). However, Marian and Siméon's path based pruning degenerates (no further pruning is performed) also for the second query, since the `descendant-or-self::node` ends up in the set of pruner paths, thus selecting all nodes. This is because their approach cannot manage predicates. In our approach instead predicates are taken into account and therefore only nodes satisfying $C(y)$ are kept by the projector, thus yielding a very precise pruning.

It is important to stress that despite their specific form the first kind of queries is very common in practice since they are generated from XQuery→XQuery-Core compilation of a non negligible class of queries (for instance Q13 of the XPathMark) or when rewriting upward axes into downward ones. This latter observation shows that the application of rewriting rules rules of [15] to extend Marian and Siméon's approach to upward axes is not feasible since the rewriting may completely compromise pruning.

6. EXPERIMENTS

We have implemented a complete version of the algorithm defined for full XPath. The code (available at `http://www.lri.fr/~kn`) is written in OCaml, uses the PXP library for parsing XML documents, and its correctness was verified for all tests. After the path extraction of Section 5, it performs the rewriting presented in Sections 3.3 and 4.3, and the static analysis defined in Section 4. The latter is extended to deal with attributes, with the wildcard test `element()`, with `{descendant,ancestor}-or-self` and `{preceding,following}-siblings` axes, and with absolute paths. It also uses a couple of heuristics. One heuristic rewrites the DTD E so that every name Y defined as $Y \to String$ occurs exactly once in the right hand side of an edge of E; this enhances the precision of pruning by reducing the number of conflicts on the leaves of the tree. The other heuristic keeps track of the depth of elements in the paths in order to improve pruning, especially in presence of recursive DTDs (this latter heuristics could be embedded in the formal treatment, but we preferred to keep it simpler). Pruning is then performed *in streaming* and merely consists of a one-pass traversal of the document. We also added an optional validation option, that makes it possible to prune the document while validating it. Programs that use an external validator can therefore prune their document without any overhead.

We performed our tests on a GNU/Linux desktop, with 3GHz processor, 512 MB of RAM and a single S-ATA hard-drive, using DTDs, document generator, and queries of XMark and XPathMark (the latter is interesting because its queries use all the available axes). Queries were processed by the latest version of Galax (that is, the 0.5.0). Swap was disabled to test memory limits.

For what concerns the overhead of the optimisation, tests confirmed that it is always negligible, both in memory and time consumption: the only noticeable overhead is pruning time, which is linear in the size of the pruned document, but can be embedded in document parsing and/or validation (e.g., for 60MB documents computing the projector took around 0.5s while pruning and saving the pruned document to disk was always below 10s). These results were confirmed by further experiments on large DTDs (e.g. XHTML) and long XPath expressions (twenty steps or so).

In Table 1 we report part of the results of our tests. For space reasons just a selection of XMark (QM) and XPathMark (QP) queries are presented.

Projector efficiency. The fourth line of Table 1 reports the effect of inferred projectors and it is an indicator of the selectivity of the query. For several XMark queries the size of the pruned document is around 70-80% of the size of the original document. This is due to the fact that XMark documents contain mixed-content `<description>` elements which account for about 70% of the total size. Thus, queries whose execution requires the whole content of `<description>` elements, preserve a large part of the file. On the contrary, for very selective queries like QM06, 99.7% of the document is discarded. Finally, for queries that are very little selective, like QP13, the whole document has to be kept. It should be noted in Table 1, fourth line, that for all XMark queries but QM14 we could prune more than 95% of the original document.

Execution time and memory occupation. The comparison of performances of the Galax query engine on an original document and its pruned version is given in Figures 4 and 5, which respectively report the processing times and main memory occupation for documents of 56MB. They show that time and memory gains are

	QM03	QM06	QM07	QM14	QM15	QM19	QP01	QP02	QP03	QP04	QP05	QP06	QP07	QP08	QP09	QP10	QP11	QP12	QP13	QP21	QP23
Original Document Size (MB)	930	2048*	1100	202	2048*	964	112	313	258	291	123	190	168	123	459	123	369	134	79	224	403
Pruned Document Size(MB)	25	5,3	42	139	24	24	89	50	46	50	98	133	123	99	35	98	28	107	78	152	42
Main Memory Usage (MB)	374	90	380	512	245	512	391	399	433	434	418	485	467	466	466	483	456	460	504	459	465
Gain in Size (% of original)	2.5	0.3	3.4	69.6	1.15	2.5	80.4	15.7	17.5	16.8	80.4	69.6	73.2	80.4	7.5	80.4	7.5	80.4	98.2	67.9	10.4
Gain in Speed ($\times$ faster)	17.8	110.1	28.2	3.9	62.6	7.5	1.5	3.6	3.7	4.3	1.5	2.9	2.6	1.1	4.9	1.6	4.2	1.6	1.0	3.6	3.6

*: biggest file the XMark generator was able to produce.

Table 1: Sizes (in MBytes) of the biggest document processed thanks to pruning, size of its pruned version, and memory used to process the latter. Percent of the pruned document and speedup of the execution time for a 56MB document.

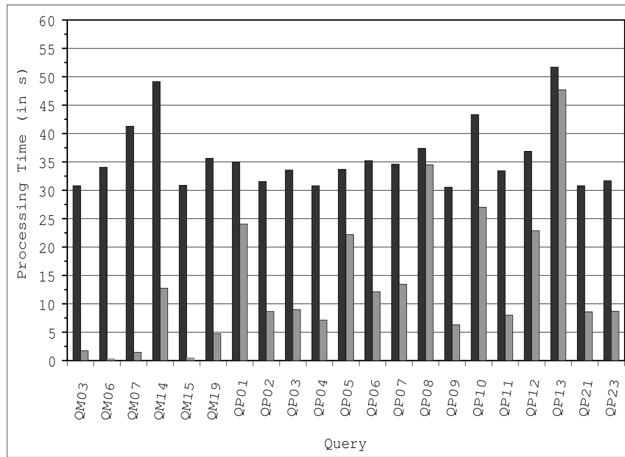

Figure 4: Processing time of a query on original (56MB) and pruned documents

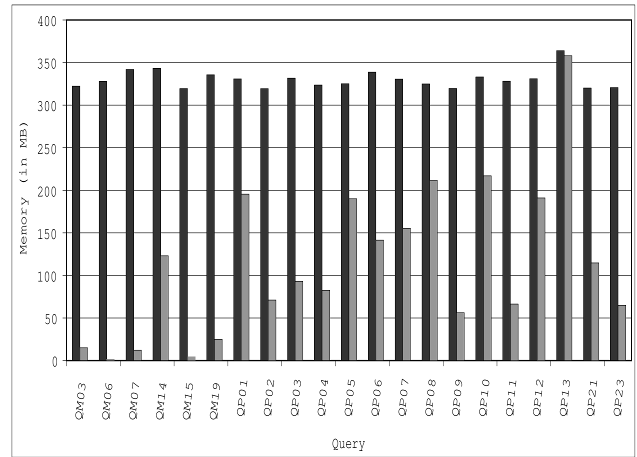

Figure 5: Memory used to process a query on original (56MB) and pruned documents

similar.

These gains translate in practice into much faster executions and the possibility to process much larger documents. The improvement can be measured by looking at the first and last lines of Table 1. The first line reports the size of the largest document it was possible to process thanks to pruning. This must be compared with the fact that, for all queries, the largest document that can be processed without pruning is 68MBytes large. The last line reports how many times the execution on a pruned document is faster than the execution on the original document. It is important to note that, depending on the nature of the query, the gain can be much higher than the proportion given by the percent of the size of the pruning. For instance, for queries such as QM14, QP6, and QP21 the size of the pruned document is two-thirds of the size of the original document, but they can then be processed from three to four times faster and, as Figure 5 shows, using three times less memory than when processed on the original. The latter is a huge gain when one knows that memory usage is one of the main bottlenecks for real life query processing (e.g., in DOM-based implementations of XPath or XSLT processors).

Quite informative, as well, is the data in the second line of Table 1 which reports, for each query, the size in MB of the maximum pruned document. It is interesting to see that, while the maximum size for an unpruned document is 68MB, we can process documents for which the projection has a size of 152MB (on disk). This is due to the fact that projecting a document not only reduces its size but also its *complexity* by reducing the number of types of nodes. This simplification of the document reduces the amount of extra-information the query engine has to keep for each node and, consequently, its memory usage. More precisely, the benefit of pruning out some (types of) nodes is twofold: first, the fan out of the document is reduced and this may impact memory usage for engines that chase sibling pointers and, second, the number of element names is reduced, which may reduce memory occupation when shredding.

These results are a clear-cut improvement over current technology. While we cannot directly compare processing performances since no implementation of the other pruning approaches is publicly available, we want to stress two points: (i) with one exception (QM14) the amount of pruning on common experiments is always equal or better with our approach than the others and (ii) performing pruning never is a bottleneck in our case thanks to fact that our solution consists of a single bufferless one pass traversal of the input document (on our 512MB machine we were able to efficiently prune arbitrary large documents, while in case of [14] pruning can end up using as much memory as the execution of the query).

7. CONCLUSION AND FUTURE WORK

The benchmarks show the clear advantages of applying our optimisation technique to query XML documents, and the characteristics of our solution make it profitable in all application scenarios. We discussed several aspects for which our approach improves the state of the art: for performances (better pruning, more speedup, less memory consumption), for the analysis techniques (linear pruning time, negligible memory and time consumption), for its generality (handling of all axes and of predicates), and, last but not least, for the formal foundation it provides (correctness formally proved, limits of the approach formally stated).

Future work will be pursued in three distinct areas: formal developments, database integration, and implementation issues.

For what concerns the formal treatment, we have to integrate in it the heuristics used in the implementation of the static analysis and to formally state the soundness and completeness of some approximations presented in the work. Also, it should be easy to adapt the approach to work in the absence of DTDs, by using dataguides/path-summaries instead. We intend also to adapt out technique to optimise queries written in ℂQL [7] the query language of ℂDuce [6]: as we said at the end of Section 3, their rich type system will allow us to assign more precise types to queries (for instance, it will be possible to capture by types many XPath predicates, since disjunction, conjunctions and negations can be handled by the corresponding type operators and the value of attributes and element contents can be expressed by singleton types) and thus to perform more selective pruning. Finally, we want to modify our approach so that it can yield efficient pruning also in the presence of XPath 2.0 predicates that test the XML Schema of nodes. Note indeed that such predicates are blockers for pruning: we have to leave the entire subtree intact so that the engine can verify that it has the specified schema. But since the projector inference algorithm already statically checks this property, the idea is to make the inference algorithm also rewrite predicates so as to push the schema tests down where they are strictly necessary, thus making further pruning possible.

From a database perspective we want to study the integration of our optimisation technique with classical database ones. Our technique must be viewed as a preliminary step that can be further combined with more traditional database optimisations. More precisely, as our technique is able to take into account the workload, in the line of [8], it could help the database administrator to deduce relevant clustering strategies of XML data on disk and to define well-adapted indexes and/or materialised views. Second, our pruning technique can also be used for pruning indexes. For example, if indexes over element tags are present before query processing (like in the TIMBER system), the index can be pruned as well. In TIMBER, for a 472 MB document, such an index can reach a 241MB size [16], thus it is worth being pruned, in order to improve buffer management and concurrent query evaluations.

Finally, implementation-wise, the natural extension of our work is to interface our pruning system with a query processing engine. This would bring several advantages: (*i*) the pruning overhead would be diluted in the parsing/validation phase and (*ii*) an interaction between the query engine and the loading module would provide a way not only to prune the document but to start answering the query in streaming, when possible.

Acknowledgements. We would like to thank Haiming Chen for pointing us an error in the two typing systems of a preliminary version of this work. This work benefitted from several discussions with and suggestions from Ioana Manolescu and Carlo Sartiani. Two of the three VLDB anonymous referees provided very useful feedback. This work was partially funded by the French ACI project "Transformation Langages for XML: Logics and Applications" (TraLaLA) and the French ACI young researcher project "WebStand".

8. REFERENCES

[1] Galax. `http://www.galaxquery.org`.

[2] XML Path Language (XPath) 2.0. `http://www.w3.org/TR/xpath20`.

[3] XML Query Use Cases. `http://www.w3.org/TR/xquery-use-cases/`.

[4] XQuery 1.0 and XPath 2.0 Formal Semantics. `http://www.w3.org/TR/xquery-semantics`.

[5] XQuery 1.0 and XPath 2.0 Functions and Operators. `http://www.w3.org/xquery-operators`.

[6] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *ICFP '03, 8th ACM Int. Conf. on Functional Programming*, pages 51–63, 2003.

[7] V. Benzaken, G. Castagna, and C. Miachon. A full pattern-based paradigm for XML query processing. In *PADL '05, the 7th Int. Symp. on Practical Aspects of Declarative Languages*, number 3350 in LNCS. Springer, 2005.

[8] V. Benzaken, C. Delobel, and G. Harrus. Clustering strategies in O_2: an overview. In *Building an Object-Oriented Database System: the Story of O_2*. Morgan Kaufman, 1992.

[9] S. Bressan, B. Catania, Z. Lacroix, Y-G Li, and A. Maddalena. Accelerating queries by pruning XML documents. *Data Knowl. Eng.*, 54(2):211–240, 2005.

[10] D. Colazzo. *Path Correctness for XML Queries: Characterization and Static Type Checking*. PhD thesis, Dip. di Informatica, Università di Pisa, 2004.

[11] D. Colazzo, G. Ghelli, P. Manghi, and C. Sartiani. Types for Path Correctness for XML Queries. In *ICFP '04, 9th ACM Int. Conf. on Functional Programming*, 2004.

[12] M. Franceschet. XPathMark - An XPath benchmark for XMark generated data. In *XSym 2005, 3rd Int. XML Database Symposium*, LNCS n. 3671, 2005.

[13] D. Lee, M. Mani, and M. Murata. Reasoning about XML Schema Languages using Formal Language Theory. Technical report, IBM Almaden Research, 2000.

[14] A. Marian and J. Siméon. Projecting XML documents. In *VLDB '03*, pages 213–224, 2003.

[15] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. In *Proc. EDBT Workshop (XMLDM)*, volume 2490 of *LNCS*, pages 109–127. Springer, 2002.

[16] S. Paparizos and H.V. Jagadish. Pattern tree algebras: Sets or sequences? In *VLDB*, 2005.

[17] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *VLDB '02*, pages 974–985, 2002.