REVISITING XML QUERY PROCESSING: FROM RELATION DECOMPOSITION TO JSONB AND NOSQL

YU-CHENG CHANG[1], YI-HUNG LIN[1], TSING-LIM TSHUA[1], HUAN-XUN ZENG[1], LI-ZONG HUANG[1]

[1]Department of Computer Science and Information Engineering, National Taiwan University of Science and Technology, Taipei, Taiwan

E-MAIL: M11415015@mail.ntust.edu.tw, M11415084@mail.ntust.edu.tw, M11415038@mail.ntust.edu.tw, M11415105@mail.ntust.edu.tw, M11415056@mail.ntust.edu.tw

Abstract:

The classic paper "Relational Databases for Querying XML Documents: Limitations and Opportunities" argued that XML documents stored in relational tables suffer from high query overhead, and suggested that performance could be improved by exploiting relational schemas and indexes more carefully. However, the original evaluation mainly focused on query execution time, without explicitly accounting for data loading and preprocessing costs. In this work, we revisit this problem using a synthetic XML corpus generated from a NITF DTD and compare four approaches: (1) querying raw XML with XPath in PostgreSQL (baseline), (2) extracting key attributes into generated relational columns with indexes, (3) transforming XML into JSONB with functional indexes, and (4) storing the same logical content in a NoSQL database (MongoDB). For each approach, we measure both preprocessing time (loading, transformation, index creation) and steady-state query latency for a fixed selection query repeated 50 times. Our preliminary results show that while relational XML+XPath is the slowest at query time, XML with generated columns and JSONB with indexes achieve one to two orders of magnitude faster queries, and MongoDB provides competitive latency without explicit schema design. Our study provides an updated baseline for XML query processing and illustrates how modern semi-structured types can largely overcome the limitations identified in earlier work.

Keywords:

XML, JSONB, NoSQL, query processing, benchmarking

1. Introduction

XML has long been used as a standard format for semi-structured data. A classical study, "Relational Databases for Querying XML Documents: Limitations and Opportunities", showed how XML trees can be shredded into relational tables and queried using SQL. While this approach enables the use of mature relational optimizers and indexes, the paper also reported non-trivial performance overhead and pointed out several limitations related to recursion, path navigation, and schema design.

Since that work, database systems have evolved significantly. Modern relational engines support native XML, JSON and JSONB types, and NoSQL systems have become widely available. These developments provide new options for storing and querying XML-like data without fully decomposing documents into many relational tables.

In this paper, we revisit the problem of querying XML documents in a relational setting and compare it with a NoSQL alternative. In contrast to the original study, we explicitly distinguish between preprocessing costs (loading, transformation, and index construction) and steady-state query latency.

In their conclusion, Schmidt and Grust not only report the limitations of relational decomposition for XML, but also sketch several promising directions, including flexible comparison operators and indices for semi-structured values, multiple-query optimization of regular path expressions, and more powerful forms of recursion.

Twenty years later, mainstream systems provide many of these features out of the box through native XML/JSON types, expression indexes, and document stores. This paper revisits their setting from this modern perspective and makes the following contributions:

1) We construct a small, reproducible XML benchmark based on MSV-generated NITF documents, closely mirroring the DTD - driven workload assumed in the original paper.

2) We empirically compare four query-processing schemes:
   (i) raw XML + XPath in PostgreSQL (baseline),

(ii) generated columns with relational indexes,
(iii) JSONB with functional indexes, and

(iv) a MongoDB document store. The latter three can be seen as concrete realizations of the "flexible index on semi-structured data" direction suggested by Schmidt and Grust.

3) We quantify both preprocessing cost and query latency for a representative selection predicate. Our results show that index-friendly relational and JSONB layouts speed up XPath-style selection by roughly one to two orders of magnitude compared to the baseline, while MongoDB offers an intermediate point in the design space.

2. Background and Original Suggestions

The work by Schmidt et al. on querying XML with relational databases proposed to store XML documents in a set of relational tables capturing paths, nodes and values. Their experiments showed that naïve approaches that simply store XML as large strings are inefficient, and they suggested that carefully designed schemas and indexes could mitigate some of the performance issues.

However, the authors also acknowledged two important challenges. First, the recursive nature of XML paths leads to complex joins and large intermediate results, even when path indexes are used. Second, the cost of shredding XML documents into many relational tables and reconstructing results was not fully captured in the reported query times. The paper mainly measured query execution over pre-populated relational data, while loading and preprocessing costs were only discussed qualitatively.

Our study takes these observations as a baseline. We treat "XML stored in a relational database and queried via path expressions" as the traditional approach, and we ask how modern features such as generated columns, JSONB types, and NoSQL document stores change the performance landscape.Tables

3. Experimental Design

3.1. Data Generation

To obtain a controlled XML workload, we use the MSV XML generator and the NITF 3.0 DTD to synthesize 50 news-like XML documents. Each document is valid with respect to the DTD and contains attributes such as `change.time` and a nested `<body>` element. This setting allows us to reproduce the idea of "DTD-driven XML data" from the original work while keeping the schema simple enough for manual inspection.

3.2. Systems and Storage Schemes.

We compare four storage and query schemes:

(1) Raw XML + XPath in PostgreSQL (baseline)

All documents are stored in a single table nitf_xml_raw(id, filename, doc XML).
Queries use PostgreSQL's xpath function to filter on attributes, e.g. xpath('/nitf/@change.time', doc).
This configuration serves as the **baseline** corresponding to the traditional "relational database + XML type + XPath" approach.

(2) Generated columns with relational indexes (XML + index-friendly columns)

From the same doc column we define a stored generated column

```
change_time TEXT GENERATED ALWAYS AS
    ((xpath('/nitf/@change.time', doc))[1]::text) STORED
```

and build a B-tree index on change_time.
Queries are then written in purely relational form, e.g. WHERE change_time = '14:00'.
This configuration follows the original paper's suggestion to exploit relational indexes and schema information more aggressively instead of evaluating XPath at query time.

(3) JSONB representation with functional index.

Each XML document is transformed into a compact JSONB object that records the root tag, the extracted change_time, and a simplified body_text.
The results are stored in 'nitf_jsonb(id, filename, data JSONB)' together with a functional index on (data->>'change_time').
Queries use JSONB operators such as'data->>'change_time' = '14:00'.'This represents a hybrid approach that keeps the data semi-structured while allowing PostgreSQL's JSONB indexes and optimizer to accelerate lookups.

(4) NoSQL document store (MongoDB)

We insert one JSON document per file into a

MongoDB collection with fields `filename`, `change_time`, and `body_text`. An index is created on `change_time`, and we query with `{ change_time: "14:00" }`. This configuration serves as a representative NoSQL baseline for document-oriented XML/JSON storage.

3.3. Preprocessing and Query Benchmarking

For each scheme we separate:

- Preprocessing time:

  including data loading, transformation (XML → generated columns / JSONB / MongoDB documents) and index creation.

- Query time:

  we fix a selection query on `change_time = '14:00'` and execute it 50 times in each system.

  We record the average, median, minimum and maximum latency using Python scripts for PostgreSQL and PyMongo for MongoDB.

  All measurements are taken with `EXPLAIN ANALYZE` (for PostgreSQL) or client-side timing, after warming up caches.

4.Result

We evaluated four query-processing configurations on the synthetic NITF XML corpus generated from the official NITF 3.0 DTD using msv-generator. For each configuration we executed the same equality predicate on the change.time attribute (change_time = '14:00') 50 times and recorded average, median, minimum, and maximum response time at the client side. All PostgreSQL experiments used the same containerized PostgreSQL 16 instance; MongoDB experiments used a separate MongoDB 7 container on the same host.

Table I summarizes the results for the four relational configurations. The xml_xpath setup corresponds to the traditional baseline in which each document is stored as a single xml value and the predicate is evaluated by calling xpath('/nitf/@change.time', doc) in the WHERE clause. This configuration achieved an average latency of 1.813 ms per query, with a median of 1.437 ms and a maximum of 4.949 ms over 50 runs. This matches the behavior reported by Grust et al. for path-based querying over purely hierarchical storage: even simple attribute predicates trigger repeated XML parsing and path evaluation at query time.

Benchmark	Avg	Median	Min	Max
Xml_xpath	1.813	1.437	1.414	4.949
Xml_generated_col	0.040	0.032	0.031	0.118
JSONB	0.037	0.037	0.036	0.048
NoSQL	0.265	0.259	0.250	0.422

Table 1 Experiment result

In the **xml_generated_col** setup we materialized the change.time attribute as a *generated column* of type TEXT, defined as
change_time GENERATED ALWAYS AS ((xpath('/nitf/@change.time', doc))[1]::text) STORED,
and created a B-tree index on this column. With this schema-level optimization, the same predicate change_time = '14:00' no longer touches the XML value at query time. The average latency dropped to **0.040 ms**, with a median of **0.032 ms**, corresponding to roughly a **45× speedup** over the raw XPath baseline.

The **jsonb_change_time** configuration represents a semi-structured variant: each document is pre-transformed once into a JSONB value with fields root_tag, change_time, and body_text, and an index is built on (data ->> 'change_time'). The equality predicate then becomes data->>'change_time' = '14:00'. Query latency in this case is essentially identical to the generated-column approach: an average of **0.037 ms** and median of **0.037 ms** over 50 runs. This suggests that, for simple attribute predicates, PostgreSQL's optimizer can exploit B-tree indexes over both relational columns and JSONB expressions equally well, as long as the predicate is indexable.

To provide a NoSQL comparison point, we loaded the same change_time and body_text fields into a MongoDB collection and issued the equivalent query { change_time: "14:00" } using PyMongo. Over 50 runs, the MongoDB configuration achieved an average latency of **0.265 ms**, a median of **0.259 ms**, a minimum of **0.250 ms**, and a maximum of **0.422 ms**. Thus, in our setting, MongoDB is still about an order of magnitude faster than the unindexed XPath baseline, but clearly slower than the indexed relational and JSONB configurations. This gap can be reasonably attributed to driver overhead and network round-trips between the Python client and the MongoDB container, whereas the PostgreSQL benchmarks are executed inside a single process.

For fairness, we also measured the one-time preprocessing overhead needed to enable the "modern" configurations. Materializing the JSONB representation directly inside PostgreSQL via a single INSERT … SELECT statement took about **17 ms** to transform 50 documents (≈0.34 ms per

document), and building the JSONB index required roughly **29 ms**. Similar costs apply to adding the generated column and its index. Compared to the steady-state query latency (tens of microseconds per request), these preprocessing steps are negligible for any realistic workload in which documents are queried more than a few times.

5. Conclusion

The original study "Relational Databases for Querying XML Documents: Limitations and Opportunities" argued that storing XML as large strings and evaluating queries via path expressions leads to poor performance, and suggested that carefully designed relational schemas and indexes could alleviate some of these problems. Our revisit of this topic, using a synthetic NITF workload and modern database systems, confirms this view in a simplified setting.

First, our baseline configuration that stores each document as a single XML value and filters it with `xpath` exhibits millisecond-level latency even for a very simple equality predicate. Second, exposing the `change.time` attribute either as a generated relational column or as a JSONB field with an index reduces query time by roughly two orders of magnitude, down to tens of microseconds per request. In practice, these indexed configurations make XML-derived data competitive with ordinary relational queries, while incurring only modest one-time preprocessing costs for transformation and index creation. Third, a NoSQL document store such as MongoDB provides sub-millisecond latency for the same query without any explicit schema design, but still falls behind the optimized relational and JSONB setups in our micro-benchmark.

These observations suggest that, for workloads dominated by simple attribute predicates over DTD-conforming XML, modern relational systems with semi-structured types already address many of the limitations highlighted in earlier work, without resorting to heavy-weight shredding into large numbers of tables. Our study is limited by the small scale of the dataset, the focus on a single NITF DTD, and the use of a single equality predicate; more complex recursive path expressions and heterogeneous result construction remain outside our scope. As future work, we plan to extend the benchmark to larger corpora, richer query sets, and native XML/JSON path indexes, and to systematically compare them with the Shared and Hybrid decomposition schemes proposed in the original paper.

References

[1] J. Shanmugasundaram *et al.*, "Relational databases for querying XML documents: Limitations and opportunities," *Proceedings of VLDBpages*, pp. 302–314, 2008.