Querying XML : benchmarks and recursion
Afanasiev, L.

Citation for published version (APA):
Chapter 7

A Micro-Benchmark for Value-Based Equi-Joins

In Section 3.7, we investigated the part of the Michigan benchmark that was designed to test value-based joins expressed in XQuery and we found that its query set is not sufficient for a satisfactory performance analysis of this query operation. In this chapter, we provide an accurate and more comprehensive micro-benchmark inspired by the join queries of MBench.

We present a micro-benchmark for evaluating the performance of query processing techniques for value-based joins expressed in XQuery (Section 7.2). The benchmark allows for a detailed analysis of engine performance with respect to seven query and data parameters. Using this micro-benchmark, we conduct an extensive analysis of performance of four open-source XQuery engines (Section 7.3). Our analysis indicates that the join-processing techniques deployed by the engines have room for improvement and that the micro-benchmark we propose provides an accurate and comprehensive tool for testing value-based joins (Section 7.4).

The micro-benchmark we present here is an extended version of the micro-benchmark previously published in Afanasiev and Marx [2008]. We extended the benchmark with more parameters and with a more systematic way of varying their values and analyzing their impact on the performance results.

7.1 Introduction

In relational databases, the join operation is one of the fundamental query operations. It combines information from two different relations based on their Cartesian product. It is inherently one of the most difficult operations to implement efficiently, as no predefined links between relations are required to exist. Since it is executed frequently and it is expensive, for more than 30 years now, a lot of research effort has been invested in the optimization of join processing [Mishra and Eich 1992].
In the settings of XML databases and XQuery, we distinguish two types of joins: **value-based joins** and **structural joins**. Just as in the relational case, the value-based join combines information from two sequences of items based on their Cartesian product and the join condition is expressed on the atomic values of the items (e.g., attribute values, the string value of the item, etc.). The structural join, on the other hand, expresses conditions on the structural relationships between the pair of nodes in the XML tree. The *where* clause of the FLWOR expression in XQuery is especially designed to express joins. Nevertheless, XQuery allows for other equivalent ways of expressing joins (we count four different syntactic patterns), which adds to the complexity of the join-processing task.

Early on, much of the research effort on XML databases, focused on optimizing structural joins as it is a new and essential challenge to querying them [Gou and Chirkova, 2007]. The consolidation of the language standard and the need for improving performance of XQuery engines, draws more attention to improving the handling of data values, including optimizing the value-based join. Recall from Chapter 3 where we evaluated the performance of four XQuery engines on five XQuery benchmarks, that the most challenging benchmark queries involved value-based joins. In this chapter, we focus on value-based joins.

As performance evaluation is essential for the development of query processing techniques, a benchmark is needed for testing the performance of value-based joins. The MBench micro-benchmark [Runapongsa et al., 2002] makes a first attempt to address the performance of value-based joins by dedicating four queries to testing them. However, as we argue in Section 3.7, the four queries are not sufficient to get a satisfactory view of the performance of an engine on value-based joins. The previously proposed application benchmarks for XQuery are of no help either: although they contain value-based joins in their queries, as we discuss extensively in Chapter 3, their queries and measures do not focus on a particular language operation and thus, they are not suitable for a detailed analysis of value-based joins in particular. A benchmark that targets an accurate and comprehensive evaluation of this operation is needed.

Our goal in this chapter is to design a micro-benchmark targeting the performance of processing techniques for value-based joins. The micro-benchmark should provide a comprehensive view on the performance of an engine on these joins, taking into account performance-critical query and data parameters. The micro-benchmark should allow developers to accurately evaluate their join-processing techniques and it should allow users to analyze the performance of an engine with respect to external parameters they can control.

The research question we address in this chapter is:

**7.1. Question.** How to measure the performance of value-based joins expressed in XQuery? What is a suitable measure and which parameters are important to consider?
7.2. A micro-benchmark for value-based equi-joins

Our approach to developing the micro-benchmark is to follow the general MemBeR micro-benchmarking methodology as a design guideline. We draw inspiration from the literature on optimizing joins and construct a list of parameters that are important for the performance of joins in relational databases. Further, we consider a new, XQuery-specific parameter, namely the syntactic pattern used for expressing the join. We observed in Section 3.7, that this parameter has a significant impact on the performance of at least one XQuery engine. Finally, we measure the impact of each parameter on the query processing time(s).

We evaluate our micro-benchmark by analyzing the performance of four open-source XQuery engines: SaxonB, Qizx/Open, Galax, and MonetDB/XQuery. As a result, we obtain the most comprehensive analysis of these engines with respect to value-based joins to date (September 2009). For this analysis, we assume the role of a user that treats the engine as a black box and we explain the performance of the engines entirely in terms of the impact of the micro-benchmark parameters.

7.2 A micro-benchmark for value-based equi-joins

In this section, we describe our micro-benchmark in accordance with the MemBeR methodology.

7.2.1 Target

We present a micro-benchmark that targets the performance of the join-processing mechanism (the component under study) of an XQuery engine (the system under test). The targeted language feature is value-based equi-joins, i.e., joins that express equality on data values. We consider equi-joins on numeric data values stored in XML attributes.

7.2.2 Measure

Following the MemBeR methodology (see Chapter 6), the general measure of the micro-benchmark is the performance time as a function of its parameters. We consider six query parameters and one data parameter: syntactic pattern, number of join conditions, Boolean connectives, join-type, join-value data type, join selectivity, and document size. We define these parameters in the next section.

For two of the parameters, the measure can be described in a different way. Measuring the impact of the syntactic pattern used to express the join on performance, can be seen as measuring the robustness of the engine against syntactic variations. The idea behind this measure is to compare the performance of queries expressed in different syntactic variants. The difference between the performance times of each variant indicates the robustness of the engine.
Chapter 7. A Micro-Benchmark for Value-Based Equi-Joins

Where:
for $a$ in A, $b$ in B
where $a/@att1 = b/@att2$
return ($a/@att1, b/@att2$)

If:
for $a$ in A, $b$ in B
return if( $a/@att1 = b/@att2$ )
then ($a/@att1, b/@att2$)
else ()

Predicate:
for $a$ in A, $b$ in B
$\mathbf{B}[$a/@att1 = ./@att2] return ($a/@att1, b/@att2$)

Filter:
for $a$ in A, $b$ in B
return ($a/@att1, b/@att2$)

A, B: path expressions; att1, att2: attribute names

Figure 7.1: Four logically equivalent ways of expressing an equi-join.

The impact of the document size on performance measures the scalability of join-processing techniques.

The benchmark targets mainly the total query processing time, which is the time the engine spends to process the query, from when it was fired to the moment of returning the results. However, it is also interesting to measure the query compilation time and the query execution time apart. Query compilation time is the time the engine spends in the static analysis phase of the XQuery processing model [World Wide Web Consortium, 2007]. This time includes static optimizations. Query execution time is the time the engine spends on the dynamic evaluation phase of the XQuery processing model [World Wide Web Consortium, 2007]. This time includes dynamic optimizations. The performance metric is CPU time.

7.2.3 Parameters

Following the MemBeR methodology, we design the micro-benchmark to analyze the performance of value-based joins by measuring the impact of different parameters on performance. We vary parameters that are known to have an impact on join processing techniques in relational databases [Mishra and Eich, 1992, Lei and Ross, 1998]:

p1. syntactic pattern – the syntactic construction used to express the join;

p2. number of join conditions – the number of join conditions used in one join;

p3. Boolean connectives – the Boolean connectives used to combine multiple join conditions within a join;
7.2. A micro-benchmark for value-based equi-joins

p4. *join type* – whether the path-expressions $A$ and $B$ (Figure 7.1) are the same or not;

p5. *join-value data type* – the data type of the attributes on which the join is expressed;

p6. *join selectivity* – the number of pairs of items returned by the join; and

p7. *join input size* – the sizes of the two sequences participating in the join, i.e., the sequences selected by the path-expressions $A$ and $B$ in Figure 7.1. We control this parameter by varying the *document size*—the size of the document on which the join is expressed.

Below we explain each parameter in detail.

**p1. Syntactic patterns** We consider four equivalent syntactic variants for expressing value-based joins: *where*, *if*, *pred*, and *filter* shown in Figure 7.1. where $A$ and $B$ are path expressions and $att1$ and $att2$ are attribute names.

A common way of expressing joins in XQuery is by using the *where* clause of a FLWOR expression. For example, all five XQuery benchmarks discussed in Chapter 3 contain joins expressed in this way. In accordance with the XQuery semantics [World Wide Web Consortium, 2007b], the *where* clause is normalized to an *if* expression in XQuery Core, the complete fragment of XQuery that is used to specify the formal semantics. Thus, the *where* and the *if* join patterns have the same normal form in XQuery Core. Engines that use this normalization are guaranteed to treat these two syntactic patterns equivalently.

In the *predicate* pattern, the join is expressed in a predicate. Two out of the five benchmarks we studied in Chapter 3 (XMach-1, XBench) contain joins expressed with this pattern. In the *filter* pattern, the same predicate condition appears in the return clause as a filter to the sequence construction.

**p2. Number of join conditions** and **p3. Boolean connectives** We consider joins with one, two, or three different join conditions combined with conjunction or disjunction between them. For example, the following join pattern contains two join conditions combined with conjunction:

```xquery
for $a$ in $A$, $b$ in $B$
where
$a/@att1 = b/@att2$ and $a/@att3 = b/@att4$
return ($a$, $b$)
```

where $att3$ and $att4$ are attribute names.
p4. Join type  We consider two different types of joins, self-joins and general joins. If the path-expressions \( A \) and \( B \) of a join (Figure 7.1) are the same, then the join is called a self join, otherwise it is a general join. Thus the self-join is a special case of the general join, where the input sequence is joined with itself.

p5. Join-value data type  We consider joins on attributes of data value types integer and id/idref. In the presence of a DTD, the integer attributes are declared as CDATA and the id/idref attributes are of type ID and IDREF. In the presence of XML Schema, the integer attributes are declared as xs:integer and the id/idref attributes are of type xs:ID and xs:IDREF. The micro-benchmark data set contains both a DTD and an XML Schema to describe the documents.

p6. Join selectivity  The number of pairs of items returned by the join, or in other words the join result size, we measure as a percentage of the number of pairs of the underlying Cartesian product of the join. We vary the selectivity in four discrete steps: tiny (XS, 0.002%), small (S, 0.2%), medium (M, 14%), and large (L, 62%).

p7. Join input size and Document size  By fixing the selectivity of path-expressions \( A \) and \( B \) to a percentage of the number of nodes in the queried document, we tie the join input size directly to the document size. Then we consider documents ranging from 1MB to 46MB (approximately 1,500 to 67,000 nodes). The result size of \( A \) and \( B \) is \( \frac{1}{64} \times N \) (1.6%), where \( N \) is the number of nodes in the document.

7.2.4 Documents

We use the documents and schema of MBench [Runapongsa et al., 2002] for our micro-benchmark. In Section 3.7 of Chapter 3, we have seen that these documents have data value distributions that allow us to easily control the selectivity of the benchmark queries and are the key to the micro-benchmark’s ability to vary parameters in isolation. Below, we briefly recall their structure and properties.

MBench documents have two types of elements, eNest and eOccasional. Most (99%) of the elements of the MBench data are of type eNest. The eNest element has six numeric attributes with precise value distributions:

- \( a\text{Unique1} \): A unique integer indicating the element’s position in the data tree in breadth-first order; it serves as the element identifier (type ID);
- \( a\text{Unique2} \): A unique integer generated randomly;
- \( a\text{Level} \): An integer whose value equals the length of the path from the node to the root;
7.2. A micro-benchmark for value-based equi-joins

- **aFour**: An integer set to \( a\text{Unique2} \mod 4 \);
- **aSixteen**: An integer set to \( a\text{Unique1} + a\text{Unique2} \mod 16 \); and
- **aSixtyFour**: An integer set to \( a\text{Unique2} \mod 64 \).

The remainder (1%) of the elements in the data set are of type **eOccasional**. An **eOccasional** element is added as a child to an **eNest** element if its **aSixtyFour** attribute is 0. The **eOccasional** element contains only one attribute **aRef** of type **IDREF**. The value of **aRef** is set to the **aUnique1** attribute of the parent minus 11 (\( a\text{Unique1} − 11 \)), i.e., **aRef** refers to an **eNest** element that precedes the parent of **eOccasional** with 11 positions in the breadth-first order (if it exists, otherwise it refers to the root).

MBench contains three documents of varying sizes. In Section 3.6, we have seen that the smallest document of 46MB is already large enough to seriously challenge the tested engines on join queries. Therefore, for our micro-benchmark we scale the document size by cutting off the 46MB MBench document at different depths starting with depth 9. The original document is of depth 16. As a result we obtain the following data set:

<table>
<thead>
<tr>
<th>Depth</th>
<th>Size</th>
<th># of <strong>eNest</strong> elements ((\times 10^3))</th>
</tr>
</thead>
<tbody>
<tr>
<td>d1</td>
<td>9</td>
<td>1.1 MB</td>
</tr>
<tr>
<td>d2</td>
<td>10</td>
<td>1.4 MB</td>
</tr>
<tr>
<td>d3</td>
<td>11</td>
<td>2 MB</td>
</tr>
<tr>
<td>d4</td>
<td>12</td>
<td>3.3 MB</td>
</tr>
<tr>
<td>d5</td>
<td>13</td>
<td>5.9 MB</td>
</tr>
<tr>
<td>d6</td>
<td>14</td>
<td>12 MB</td>
</tr>
<tr>
<td>d7</td>
<td>15</td>
<td>22 MB</td>
</tr>
<tr>
<td>d8</td>
<td>16</td>
<td>46 MB</td>
</tr>
</tbody>
</table>

The document set can be extended in the same manner and granularity by taking as bases the MBench documents of medium (496MB) and large (4.8GB) sizes.

The MBench documents are accompanied by a DTD and an XML Schema.

### 7.2.5 Queries

It is difficult to design a set of queries that covers all valid value combinations of the parameters we consider. Even if possible, the set might be too big to be manageable. With p1 having 4 values, p2 and p3 together having 6 value combinations, p4 having 2 values, p5 having 2 values, and p6 having 4 values, there are 384 possible queries. We choose a set of value combinations in which the values of p1, the values one and two of p2, and the values of p3 are combined in all possible ways, while the value combinations for p2 (the value three), p4, p5, and p6 respect the following rule: for every two values of each of these parameters
there are two queries in the set that differ only by these values. In this way, the difference in performance times of these queries can be safely attributed to the influence of the parameter that is varied and to particular values being chosen. This leads to 80 different queries.

The query set is divided into six classes. Within each class p1 is fully varied creating a set of logically equivalent queries, and p2 (values one and two) and p3 are varied together to obtain valid value combinations in such a way that it creates a set of document equivalent queries, i.e., the queries return the same result on the MBench documents. This means that the queries in each equivalence class return the same result. Thus, within one class, p1, p2, and p3 are varied, while p4, p5, and p6 are fixed. From one class to another only one of the parameters p4, p5, or p6 is varied. Figure 7.2 shows which parameter varies between classes. The class \( \overline{\text{AbM}} \) contains two more document equivalent queries with the parameter p2 having value three.

All in all, the query set consists of

---

Table 7.1: Micro-benchmark queries and their corresponding parameter values.

<table>
<thead>
<tr>
<th>( p_2 )</th>
<th>( p_3 )</th>
<th>( p_4 )</th>
<th>( p_5 )</th>
<th>( p_6 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>int</td>
<td>S</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>int</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>int</td>
<td>S</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>int</td>
<td>S</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td>int</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>int</td>
<td>S</td>
</tr>
</tbody>
</table>
20 joins (80 syntactically different queries). Table 7.1 lists the joins and their corresponding parameter values. The name of a query is an encoding of its properties. If the query is a self-join, i.e., the path expression A is the same as the path expression B, then the query name contains the capital letter “A”, otherwise it contains the capital letter “B”. If any join condition in a query (any query can have one, two or three join conditions) is expressed between two attributes with the same name, then the query contains the small letter “a”, otherwise “b”. Further, the join name contains “&” and/or “v” for every conjunct and/or disjunct it contains. Queries whose name contains “Ref” are id/idref chasing joins. Finally, the join selectivity of a query is indicated by the suffix “XS”, “S”, “M”, and “L”.

For example, query Ab&vM is a self-join containing three join conditions on different attributes connected by a conjunction and a disjunction, and it has a medium selectivity. The query has the following general pattern:

```xml
for $a in A, $b in A
where
$a/@att1 = $b/@att2 and
$a/@att3 = $b/@att4 or
$a/@att5 = $b/@att6
return ($a,$b)
```

Each query belongs to one of the six classes of document-equivalent queries. The classes are named after their member with only one join condition. For example, Ab&vM is document equivalent to AbM and both queries fall into the AbM class.

The `where` variant of the actual queries can be found in Figure 7.3 and 7.4. The whole set of queries can be found online at [http://ilps.science.uva.nl/Resources/MemBeR/mb-joins/output/outcome.html](http://ilps.science.uva.nl/Resources/MemBeR/mb-joins/output/outcome.html).

Note that we fix the selectivity of path A and B in all the queries by filtering the `eNest` elements with a particular property that always yields approximately 1/64th (1.6%) of all `eNest` elements. For example, the path-expression `//eNest[@aSixtyFour=0]` returns all `eNest` elements whose unique random number stored in `aUnique2` is divisible by 64 (@aUnique2 mod 64 = 0). The path-expression `//eNest[eOccasional]` returns the same elements as the expression `//eNest[@aSixtyFour=0]`, since `eOccasional` occurs whenever an `eNest` has the attribute `aSixtyFour` equal to 0.

We further exploit the correlation between the attribute data of the MBench documents to create document equivalent queries and thus maintain a fixed join selectivity within each class. This equivalence does not hold for all documents conforming to the MBench schema (given by DTD or XML Schema), but for those that respect value dependencies used in the creation of the MBench documents.
AaS:
for $e_1$ in doc()//eNest[@aSixtyFour=0],
 $e_2$ in doc()//eNest[@aSixtyFour=0]
 where $e_2/@aUnique2= e_1/@aUnique2$
 return (data($e_1/@aUnique1$),data($e_2/@aUnique1$))

Aa&S:
for $e_1$ in doc()//eNest[@aSixtyFour=0],
 $e_2$ in doc()//eNest[@aSixtyFour=0]
 where $e_2/@aUnique2= e_1/@aUnique2$
 return (data($e_1/@aUnique1$),data($e_2/@aUnique1$))

AbM:
for $e_1$ in doc()//eNest[@aSixtyFour=0],
 $e_2$ in doc()//eNest[@aSixtyFour=0]
 where $e_1/@aFour= e_2/@aSixteen$
 return (data($e_1/@aUnique1$),data($e_2/@aUnique1$))

Ab&M:
for $e_1$ in doc()//eNest[@aSixtyFour=0],
 $e_2$ in doc()//eNest[@aSixtyFour=0]
 where $e_1/@aFour= e_2/@aSixteen$
 return (data($e_1/@aUnique1$),data($e_2/@aUnique1$))

AbvM:
for $e_1$ in doc()//eNest[@aSixtyFour=0],
 $e_2$ in doc()//eNest[@aSixtyFour=0]
 where $e_1/@aSixtyFour= e_2/@aSixteen$
 return (data($e_1/@aUnique1$),data($e_2/@aUnique1$))

BaS:
for $e_1$ in doc()//eNest[Occasional],
 $e_2$ in doc()//eNest[Occasional]
 where $e_1/@aUnique2= e_2/@aUnique2$
 return (data($e_1/@aUnique1$),data($e_2/@aUnique1$))

Ba&M:
for $e_1$ in doc()//eNest[Occasional],
 $e_2$ in doc()//eNest[Occasional]
 where $e_1/@aUnique2= e_2/@aUnique2$
 return (data($e_1/@aUnique1$),data($e_2/@aUnique1$))

Figure 7.3: The where variant of the micro-benchmark query set, classes AaS, AbM, and BaS.

7.2.6 Running scenarios

For measuring the impact of the query parameter, we fix the document size to the largest value (d8, 46MB) and execute all queries on this document.

For measuring data scalability, all queries can be run on a subset (including a small, medium, and a large size document) of the proposed documents or on all document sizes. The latter case generates 640 measurements. Another approach is to first analyze the impact of the query parameters and then choose a few query
7.2. A micro-benchmark for value-based equi-joins

**BaL:** for $e1$ in doc()//eNest[@aSixtyFour=4], $e2$ in doc()//eNest[@aSixtyFour=0] where $e1/@aLevel$ = $e2/@aLevel$ and $e1/@aFour$ = $e2/@aFour$ return (data($e1/@aUnique1$), data($e2/@aUnique1$))

**Ba&L:** for $e1$ in doc()//eNest[@aSixtyFour=4], $e2$ in doc()//eNest[@aSixtyFour=0] where $e1/@aLevel$ = $e2/@aLevel$ or $e1/@aSixtyFour$ = $e2/@aSixtyFour$ return (data($e1/@aUnique1$), data($e2/@aUnique1$))

**BbXS:** for $e1$ in doc()//eNest[eOccasional], $e2$ in doc()//eNest[@aSixtyFour=4] where $e2/@aUnique2$ = e1/eOccasional/@aRef or $e2/@aSixtyFour$ = $e1/@aSixtyFour$ return (data($e1/@aUnique1$), data($e2/@aUnique1$))

**Bb&XS:** for $e1$ in doc()//eNest[eOccasional], $e2$ in doc()//eNest[@aSixtyFour=4] where $e2/@aUnique2$ = e1/eOccasional/@aRef and $e2/@aFour$ = $e1/@aFour$ return (data($e1/@aUnique1$), data($e2/@aUnique1$))

**BbRefXS:** for $e1$ in doc()//eNest[eOccasional], $e2$ in doc()//eNest[@aSixtyFour=4] where $e2/@aUnique1$ = e1/eOccasional/@aRef and $e2/@aFour$ = $e1/@aFour$ return (data($e1/@aUnique1$), data($e2/@aUnique1$))

**Bb&RefXS:** for $e1$ in doc()//eNest[eOccasional], $e2$ in doc()//eNest[@aSixtyFour=4] where $e2/@aUnique1$ = e1/eOccasional/@aRef and $e2/@aFour$ = $e1/@aFour$ return (data($e1/@aUnique1$), data($e2/@aUnique1$))

Figure 7.4: The *where* variant of the micro-benchmark query set, classes BaL, BbXS, and BbRefXS.

Parameter value combinations—worst performing, medium performing, and best performing—for the scalability analysis. We implement this approach for the analysis of four XQuery engines presented in Section 7.3.

7.2.7 Analyzing benchmark results

In this section, we explain how the benchmark results should be analyzed.

**Impact of p1** The robustness of the join recognition mechanism is measured by comparing the average performance times computed on queries expressed using
one syntactic pattern. Note that all 20 joins of the micro-benchmark are expressed using four different patterns, thus by taking the average for each syntactic variant we cancel the influence of the other parameters on the performance times.

If the average performance times for each value of p1 are similar (not significantly different), then we can conclude that the engine is robust at recognizing the equivalence of the syntactic join patterns.

**Impact of p2** The impact of p2, number of join conditions, is measured by comparing the average performance times computed for each value of p2, while the other query parameters are varied.

Note that the set of queries that have the value of p2 fixed on *one* and the set of queries that have the value of p2 fixed on *two* vary the other parameters in all possible ways (note that in the case where p2 has value *one*, p3 has only one possible value). The set of queries that correspond to p2 equals *three* is composed of eight queries from the class $\tilde{A}bM$ varying only p1 and p3, while p4–p6 are fixed. Thus, the difference between the average performance times for the first two values of p2 show the impact of p2, indifferent of the values of the other parameters. The average performance time computed for the third value of p2 must be interpreted in the settings of the fixed values of p4–p6.

**Impact of p3** The impact of p3, the Boolean connectives, is measured by comparing the average performance times computed for each value of p3, while the other query parameters are varied.

The set of queries that have no Boolean connectives, the set of queries that have one conjunct, and the set of queries with one disjunct as the Boolean connective of two join conditions vary all the other parameters completely. The difference between the average times computed on these query sets indicate the impact of the Boolean connective used to combine join conditions, indifferent of the values of the other tested parameters.

The average times computed on the set of queries that have two Boolean connectives to combine three join conditions (queries $Ab&&M$ and $Ab&vM$) must be interpreted relative to the fixed values of p4–p6 corresponding to the class $\tilde{A}bM$.

**Impact of p4** The impact of p4, the self or general join type, is measured by comparing the average performance times computed for the $\tilde{A}aS$ and $\tilde{B}aS$ classes. The results must be interpreted relative to the fixed values of p5 and p6, all the other parameters are varied exhaustively.

**Impact of p5** The impact of p5, the data-value type, is measured by comparing the average performance times computed for the $BbXS$ and $BbRefXS$ classes. The results must be interpreted relative to the fixed values of p4 and p6, all the other parameters are varied completely.
7.3. The micro-benchmark in action

Impact of p6 The impact of p6, join selectivity, can be measured by comparing the average performance times computed on $\tilde{BbXS}$ and $\tilde{BbRefXS}$, on $\tilde{AaS}$ and $\tilde{BaS}$, on $\tilde{AbM}$, and on $\tilde{BaL}$, i.e., for each fixed value of p6. Note that p4 and p5 vary non-systematically among these classes—by taking the first average we cancel the impact of p5, the second average cancels the impact of p4, while the third and fourth averages have different fixed value combinations for p4 and p5—thus we cannot draw any definite conclusion with respect to this measure. We use this measure only as an indication of the impact of p6 relative to the variance of other parameters.

An alternative analysis of the impact of p6 can be done relative to fixed values for p4 and p5. For example, we can compare the average performance times computed on $\tilde{BbXS}$, on $\tilde{BbS}$, and on $\tilde{BaL}$ to determine the impact of p6 relative to p4 fixed to $general$ and p5 fixed to $integer$. Or we can compare the average performance times computed on $\tilde{AaS}$, and on $\tilde{AbM}$ to determine the impact of p6 relative to p4 fixed to $self$ and p5 fixed to $integer$.

Data scalability (impact of p7) We compare the data scalability of the (chosen) different queries by comparing the angle of the scalability curves.

7.3 The micro-benchmark in action

In this section, we execute the join micro-benchmark on four XQuery engines and analyze their performance. Our primary goal is to evaluate the design of the micro-benchmark.

Experimental setup Our choice fell on the following open-source engines, mainly because of their availability and ease of use: SaxonB v9.1 [Kay 2009], Qizx/Open v3.0 [Axyana Software 2009], Galax v0.5.0 [Fernández et al. 2006], and MonetDB/XQuery v0.30.0 [Boncz et al. 2006b]. SaxonB, Qizx/Open, and Galax, are main-memory XQuery engines, while MonetDB/XQuery is a DBMS handling XML databases and XQuery. SaxonB and Qizx/Open are open-source counterparts of commercial engines SaxonA and Qizx, while Galax and MonetDB/XQuery are research prototypes. All engines are of similar maturity and language coverage [World Wide Web Consortium 2006b]. The engines are run with their default settings, without special indices or tuning for this particular task.

The experiments are conducted on a Fedora 8 machine, with a 64 bit compilation, with 8 CPUs, Quad-Core AMD Opteron(tm) of 2.3GHz, and 20GB RAM. When running the Java implementations, SaxonB and Qizx/Open, we set the Java Virtual Machine maximum heap size to 10GB.

The experiments are run with XCheck, the testing platform presented in Chapter 5. The time measurements are computed by running each query 4 times and
taking the average performance time(s) of the last 3 runs. We interrupt the
query executions that take longer than 500 seconds. This is approximately two
orders of magnitude larger than the best performance time of each engine on our
benchmark.

For the present analysis we consider only the total query processing time. The
complete experimental data and results, containing more detailed time measures-
ments, can be found at http://ilps.science.uva.nl/Resources/MemBeR/
mb-joins/output/outcome.html

Analysis of the results  For determining the impact of every parameter on the
engines’ performance we follow the instructions presented in Section 7.2.7. When
the results are not conclusive due to large variances of the time measurements
within a study group, we perform more detailed analysis by fixing parameter
p1, syntactic patterns, to a subset of its values and analyzing the impact of the
remaining parameters.

To determine whether the impact of a query parameter on performance times
is significant, we use analysis of variance (ANOVA) with the significance level
set to 0.05 (α = 0.05). ANOVA is designed to test whether the means of several
groups are equal. It does so by computing the fraction of the variance between
the tested groups and the variance within the groups. This fraction is referred
to as $F$. The probability of obtaining the value of $F$ assuming that the groups
have the same mean is referred to as $p$. In our case, a group corresponds to the
set of join queries that have the tested parameter set to a particular value. For
example, when testing the impact of p1, we consider four groups of queries and
analyze whether the performance times obtained for those groups have the same
mean. If $F$ is a large number and $p$ is close to zero, then there are at least two
groups whose means are significantly different, thus there are at least two values
of the parameter on which the engines perform significantly different. When we
find that a parameter has a significant impact, we determine the impact of every
parameter value using a pairwise comparison analysis called Least Significant
Difference (LSD). LSD is a significance test similar to the t-test, commonly used
for analyzing whether the means of two groups are the same.

ANOVA and LSD make three assumptions. First, they assume that the value
distribution within the groups is normal. We do not have good reasons to be-
lieve that the performance times within the tested groups conform to a normal
distribution—most likely they do not. Nevertheless, analysis of variance has been
argued to be robust against violations of this assumption [Keppel 1973]. More-
ever, we use the statistical tests in a conservative manner: whenever we find a
significant impact, we are fairly confident that the impact indeed exists, while
when we do not state a significant impact, there might be another, more suitable
significance test to detect it. The second assumption is that the groups have
similar variances. Third, the measurements within each group are assumed to
be independent. The design of our queries guarantees that the last two assumptions hold. For more information on these statistical methods and their use in computer systems performance analysis, we refer to [Jain, 1991, Cohen, 1995].

After analyzing the impact of the query parameters, we divide the query set into subsets that show significant performance differences among each other. The queries within a subset perform similarly. Then we pick a query from each subset and analyze its scalability. We ignore the subsets that owe their impact on performance to queries that exceeded the timeout.

In the following sub-sections, we present the micro-benchmark results in detail for each engine separately. We use three types of plots to display the performance times. First, we plot the performance of different syntactic patterns of all benchmark queries. Next, we use boxplots to display the performance times grouped per parameter—for each query parameter, the corresponding boxplot groups the performance time per parameter value. The boxplots show the median, the lower and upper quartiles, the minimum, and maximum of the performance time within each group of queries. The boxplots also show the group outliers (indicated by empty circles). When we state a visible difference in performance time we refer to the plots and when we state a significant difference we refer to the statistical analysis mentioned above. Finally, we plot the performance time of a selection of queries against the benchmark documents, d1–d8.

In Section 7.3.5, we further discuss engines’ performance and the micro-benchmark design.

### 7.3.1 SaxonB

The performance times obtained for SaxonB on our micro-benchmark are shown in Figure 7.5.

For the impact of parameters p1–p6 see Figure 7.5(a). First, we observe that SaxonB performs significantly different on queries expressed via different syntactic patterns ($F = 38, p < 0.001$). The filter pattern performs best. The times for this pattern are similar on all queries revealing a robust and efficient join processing technique. The performance times on queries expressed via where and if patterns are very similar and are ranking second best. The queries in the predicate pattern perform worse. Pairwise comparison of the four groups of queries shows significant performance differences between where, predicate, and filter, while where and if are similar.

The performance of different syntactic patterns is very different. For example, where and if queries that contain a disjunction perform better than average, while for predicate the same queries perform worse than average. When analyzing the measurements for the whole query set, due to the large variances in the measurements (see Figure 7.5(a)), none of the remaining parameters shows a significant impact. Thus, we consider that the engine implements three different approaches and analyze them separately with respect to the remaining parameters.
Chapter 7. A Micro-Benchmark for Value-Based Equi-Joins

Figure 7.5: SaxonB results on the join micro-benchmark.

(a) The impact of query parameters p1–p6

(b) Document scalability (parameter p7)
On the performance of the *where* and *if* patterns only p3 has a significant impact \((F = 66.7, p < 0.001)\). The queries containing two join conditions connected by a *disjunction* perform significantly better than the rest.

On the performance of the *predicate* pattern parameters p3 and p6 have a significant impact \((F = 3.7, p = 0.049\) and \(F = 8.9, p = 0.0015\), respectively). The queries containing two join conditions connected by a *disjunction* perform significantly worse than the rest. The queries that have the smallest join selectivity, XS, perform significantly worse than the rest.

None of the parameters p2–p6 have a significant impact on the performance of the *filter* pattern.

For the scalability analysis we choose the following queries: *AaS* (Fl), *AaS* (Wh), *AavS* (Wh), *AaS* (Pr), *AavS* (Pr), *BbXS* (Pr), and *BbvXS* (Pr), one representative from each subset of queries obtained by slicing the whole query set conform the impact of p1, p3, and p6. Figure 7.5(b) shows the scalability curves.

Note that the curves fall into two distinctive groups: the curves for *AaS* (Fl) and *AavS* (Wh) have a similar slope growing slower than the curves of the rest of the queries that also share the slope angle. This indicates that the processing approaches used for *AaS* (Fl) and *AavS* (Wh) are essentially different (better performing) than the approaches used for the other queries.

**Interpreting the results** We were surprised to observe the *filter* variant to be the winner in the case of SaxonB, since this seems a less common way of expressing joins in XQuery. We shared the benchmark results with SaxonB’s developer, Michael Kay. The author acknowledged that *filter* is the only variant of the join queries in which the sub-expression B (see the query patterns in Figure 7.1) is being pulled out of the nested for-loop, thus it is evaluated once and the results held in memory. In the other cases this opportunity is not being recognized or exploited.

The fact that the *where* and *if* patterns perform the same is, as expected, due to the fact that they share the same normal form in XQuery Core. SaxonB first rewrites a query into its normal form and then executes it. As the developer explains, the disjunctive joins expressed in the *where* and *if* forms are evaluated much faster than the other queries from the same class due to the fact that for these queries the same processing strategy as for the *filter* queries is applied. This explains our results.

### 7.3.2 Qizx/Open

The performance results obtained for Qizx/Open on our micro-benchmark are shown in Figure 7.6.

Figure 7.6(a) shows the impact of the query parameters. The queries expressed via the *where* syntactic pattern perform significantly better than the rest \((F = 36.4, p < 0.001)\). The performance of the other three patterns is very similar.
Parameter p3 has a significant impact of the performance of the where pattern \((F = 2e^4, p \approx 0)\), with all the queries containing a disjunction performing worse than the rest. None of the other parameters p2 and p4–p6 have a significant impact on the performance of Qizx/Open.

For the scalability analysis we choose the following queries: \text{AaS (Wh)} and \text{AavS (Wh)}, corresponding to the two subsets performing significantly different, obtained by slicing the query set along p1 and p3. Figure 7.6(b) shows the results. \text{AaS (Wh)} not only performs better than \text{AavS (Wh)}, but it also has a better data scalability, thus the processing technique applied for these queries are essentially different.

**Interpreting the results** The results for Qizx/Open indicate that the engine deploys a join recognition technique based on a syntactic pattern using the where clause. The three other forms are evaluated in the same way, we believe, by
materializing the Cartesian product as intermediate result. Moreover, it seems that the syntactic pattern used for recognizing joins using the where clauses fails to capture disjunctive join conditions, since these queries perform as bad as the queries written in the other syntactic forms.

### 7.3.3 Galax

The micro-benchmark results for Galax are shown in Figure 7.7. Note that Galax exceeded the timeout time on all queries expressed via the syntactic pattern predicate.

All four syntactic patterns perform significantly different from each other ($F = 6.2e+3$, $p \approx 0$). Where is the best performing, followed by if, then by filter, and the worst performing is predicate.

Note that the shape of the curves for the if and filter patterns in the first plot of Figure 7.7 are similar. Still they are significantly different, thus we analyze each pattern separately. Only parameter $p_3$ has a significant impact on the performance of these patterns ($F = 6.4,$ $p = 0.003$ and $F = 8.1,$ $p = 0.002$, respectively). The queries that contain two join conditions connected by a disjunction perform significantly worse than the ones with only one join condition. The rest of the value pairs perform similarly.

The performance of the where pattern is significantly influenced only by the $p_6$ parameter ($F = 164,$ $p < 0.001$). All four query groups corresponding to the join selectivity values show significantly different performances—the larger the join selectivity, the longer the processing times.

For the scalability analysis we choose the following queries: BbXS (Wh), AaS (Wh), AbM (Wh), BaL (Wh), AaS (If), AavS (If), AaS (Fl), and AavS (Fl), one representative from each subset of queries obtained by slicing the whole query set conform the impact of $p_1$, $p_3$, and $p_6$. Figure 7.7(b) shows the scalability results.

The queries expressed via the where pattern show slightly better scalability than the rest. Further analysis is required to determine how significant the differences are. This can be done by testing the scalability on documents of larger sizes. We indicate in Section 7.2.4 how to obtain larger size documents.

**Interpreting the results** Even though the Galax implementation pipeline [Fernández et al., 2006] indicates that all the queries are normalized to XQuery Core before processing, the differences between the where and if patterns indicate that this is not always the case. The performance on the predicate pattern suggests that the engine computes the Cartesian product and only then applies the join conditions.
Chapter 7. A Micro-Benchmark for Value-Based Equi-Joins

Figure 7.7: Galax results on the join micro-benchmark.
7.3. The micro-benchmark in action

(a) The impact of query parameters p1–p6

(b) Document scalability (parameter p7)

Figure 7.8: MonetDB/XQuery results on the join micro-benchmark.
Chapter 7. A Micro-Benchmark for Value-Based Equi-Joins

7.3.4 MonetDB/XQuery

The performance times obtained for MonetDB/XQuery on our micro-benchmark are shown in Figure 7.8. Note that MonetDB/XQuery exceeded the timeout time on the following queries expressed via the syntactic pattern filter: BbXS, Bb&XS, BbRefXS, and Bb&RefXS.

Our first observation is that MonetDB/XQuery performs significantly worse on queries expressed via filter, than those expressed via where, if, and predicate ($F = 7.9, p < 0.001$). The latter three perform almost identically.

Although there is a visible difference in the performance times for every value of $p_2$—the more conditions the join has, the longer the engine takes to evaluate it—the differences are not statistically significant due to large variances in the measurements within each group.

The queries that contain a disjunction perform visibly worse than the rest. Again, due to large variances in the performance times within each group, the difference is not statistically significant. If we analyze only the queries expressed via the syntactic patterns where, if, and predicate, thus excluding the impact of the filter pattern, the difference becomes significant ($F = 155, p \approx 0$). The pairwise value comparison reveals that the queries that contain only a disjunction and the queries that contain a conjunction and a disjunction perform significantly worse than the rest and each other. The other groups do not show a significant difference in performance.

There is no visible difference between the performance times of different values of parameter $p_4$ and $p_5$.

The times obtained for the query group corresponding to query selectivity set to XS are significantly worse than the rest ($F = 3, p = 0.03$). The difference is due to the four queries that take more than 500 seconds to run. Although there is a visible increase in performance times with the increase of query selectivity, there is no significant difference found among the other query groups. This is due to large variances in the measurements within each group.

In conclusion, varying $p_1$, $p_2$, $p_3$, and $p_6$ shows impact on MonetDB/XQuery performance. For $p_1$, $p_3$, and $p_6$ the impact is statistically significant. A further detailed analysis for subsets of the benchmark queries corresponding to particular parameter configurations might be interesting to consider.

For the scalability analysis we consider the following queries: AaS (Wh) and AavS (Wh), corresponding to the two subsets performing significantly different, obtained by slicing the query set along $p_1$ and $p_3$. Figure 7.8(b) shows the results. The curve for AavS (Wh) seems to grow faster than the curve for AaS (Wh), again showing essential differences in the processing approaches used for these queries. Considering larger document sizes might help to determine whether the slope angles are indeed significantly different.
Interpreting the results  The results for MonetDB/XQuery indicate that the engine’s join recognition mechanism detects the equivalence of the *where*, *if*, and *predicate* patterns. Nevertheless, the performance times for two queries, AaS (Wh) and BavL (Pr), deviate from the performance times of the queries expressed in the other two patterns. We do not have an explanation for this.

The join processing strategy used for the *filter* pattern performs worse than the strategy used for the joins expressed in the other patterns. On queries BbXS, Bb&XS, BbRefXS, and Bb&RefXS, the engine stumbles and does not recognize the join operation—it seems that the engine is computing the Cartesian product as an intermediate result.

### 7.3.5 Lining up the micro-benchmark results

<table>
<thead>
<tr>
<th>SaxonB</th>
<th>Qizx/Open</th>
<th>Galax</th>
<th>MonetDB/XQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1 Fl,(Wh,If),Pr</td>
<td>Wh,(If,Pr,Fl)</td>
<td>Wh,If,Fl,Pr</td>
<td>(Wh,If,Pr),Fl</td>
</tr>
<tr>
<td>p2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p3 (Wh,If): (-&amp;,&amp;&amp;),(v,&amp;&amp;)</td>
<td>Wh: (-&amp;,&amp;&amp;),(v,&amp;&amp;)</td>
<td>If: (v,&amp;&amp;)</td>
<td>(Wh,If,Pr): (-&amp;,&amp;&amp;),v,&amp;&amp;</td>
</tr>
<tr>
<td>p4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p6 Pr: (S,M,L),XS</td>
<td>Wh: XS,S,M,L</td>
<td>(S,M,L),XS</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.2: Summary of which query parameters have significant impact on the performance of the four engines. The parameter values are given in the order of significantly decreasing performance. The values that perform similarly are grouped together.

Table 7.2 contains a summary of the results. For every engine we indicate which parameters have a significant impact and on which of its values the engine shows a significantly different performance. The parameter values are given in the order of significantly decreasing performance. The values that perform similarly are grouped together. Some results are indicated for parameter p1 being fixed to a subset of its values.

Note that parameters p1, p3, and p6 have a different impact on each engine.

None of the engines is able to recognize the equivalence of all four syntactic patterns. MonetDB/XQuery is the most robust with respect to this task by performing similarly on three of the four patterns. What is more surprising is that only two of the four engines have the same performance on the *where* and *if* patterns, in spite of the fact that an equivalence preserving translation of the *where* pattern into the *if* pattern is given by the W3C [World Wide Web Consortium, 2007b].
Though the joins with *conjunctive* connectives showed visible negative impact on the engines’ performance, all engines performed significantly different on joins with *disjunctive* connectives. One engine, SaxonB, showed better performance on disjunctive joins expressed via the *predicate* pattern. In all other cases, the disjunctive joins performed worse.

In general, we observed a tendency of Galax and MonetDB/XQuery to take more time to answer joins with larger selectivity. However, due to large variances of performance time measurements of queries within the same join selectivity class, the impact of this parameter is not always significant. SaxonB and MonetDB/XQuery showed significantly large performance differences on joins with the smallest selectivity, $XS$. We do not have an explanation for this (abnormal) behavior.

Parameters p2, p4, and p5 did not show a significant impact on any engine. Parameter p2 has a visible impact on performance. Generally, the more join conditions the longer it takes to evaluate the join. But due to large variances in performance times of queries that contain conjunctions and those that contain disjunctions the influence is not significant. This parameter might show significant impact on those engines on which the impact of p3 is not as large.

Parameters p4 and p5 did not show any visible impact on any of the engines. This raises the question of whether the engines miss optimization opportunities or whether these parameters do not have an impact on value-based join processing as opposed to processing of joins in relational databases. The previous work on optimizing join processing techniques that we cite in Section 7.2.3 and the fact that other XQuery benchmarks also account for these parameters (Chapter 3), are a strong indication that these two parameters are interesting to consider. The results obtained on four engines are not yet convincing arguments against the importance of these parameters. Thus, we believe that the first conclusion is more likely and the engines do not optimize for these parameters.

### 7.4 Conclusions

Our aim in this chapter was to create a micro-benchmark that targets the performance of query processing techniques for value-based joins. The research questions we addressed are: *How to measure the performance of value-based joins expressed in XQuery? What is a suitable measure and which parameters are important to consider?*

In designing the micro-benchmark we followed the MemBeR methodology. In particular, the benchmark measures the impact of seven query and data parameters on the performance times of an engine. In choosing the benchmark parameters we drew inspiration from the observations that we made previously when analyzing the Michigan benchmark in Chapter 3 and from the work on join optimization techniques in relational databases. The benchmark query set
is carefully designed to allow for testing the impact of every parameter value in isolation. For example, for every parameter and for every two of its values, there are two queries in the set that differ only by these values. This guarantees the accuracy of the benchmark measure with respect to the tested parameters.

We tested our benchmark by analyzing the performance of four XQuery engines. As a result, we obtained a comprehensive overview of the performance of each engine when it comes to evaluating joins and we identified some shortcomings of the engines. Out of seven benchmark parameters, five parameters, syntactic pattern, number of join conditions, Boolean connectives, join selectivity, and document size, showed visible or significant impact on the performance of at least one engine. None of the engines showed impact of the remaining two parameters, join-type and join-value data type. We believe that this indicates a missed chance for the four engines to optimize for these parameters and that these parameters are still interesting to consider. We therefore conclude that the benchmark achieves its target and it is a useful tool for profiling the performance of XQuery engines on value-based joins.

In Part I of the thesis, we were concerned with developing methodology and tools for performance evaluation of XQuery processing techniques and engines. We discussed existing XQuery benchmarks, we investigated how to ensure repeatability of experimental studies, we developed a tool for automatic execution of benchmarks, and, finally, we proposed a micro-benchmarking methodology and a micro-benchmark for testing value-based joins expressed in XQuery.

In the next part of the thesis, Part II, we are concerned with optimizing recursion in XQuery. In Section 8.7 we use the tools and methodology developed in Part I to evaluate the optimization we propose.