A Macro-Benchmarking Library for the QLever SPARQL Engine

Andre Schlegel

Institute for computer science at the University of Freiburg, Germany

02.08.2024
Problem

Figure: Project page of the “QLever SPARQL engine” [1]
Join operator

<table>
<thead>
<tr>
<th>Left input table</th>
<th>Right input table</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
</tbody>
</table>
## Join operator

<table>
<thead>
<tr>
<th>Left input table</th>
<th>Right input table</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>
Join operator

<table>
<thead>
<tr>
<th>Left input table</th>
<th>Right input table</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 4 1</td>
<td>9 1 9 3</td>
</tr>
<tr>
<td>6 4 3</td>
<td>5 3 5 3</td>
</tr>
<tr>
<td>2 4 9</td>
<td>8 4 9 3</td>
</tr>
<tr>
<td>2 2 3</td>
<td></td>
</tr>
<tr>
<td>9 1 9</td>
<td></td>
</tr>
</tbody>
</table>

Output table

| 6 4 1 9 9 3 |
| 6 4 3 5 5 3 |
| 2 2 3 5 5 3 |
Merge and galloping join algorithm

- Used by “QLever SPARQL engine” [1]
- Input tables sorted by join column
- Output table sorted by join column
Hash join algorithm

- Input tables not sorted
- Output table not sorted
Simplified hash join algorithm

- Hash map
  - Key: Join column entries smaller input table
  - Value: Lists of all smaller input table rows with the key as join column entry
- Iterate bigger input table join column
Which has the shorter execution time?

More importantly: **When?**
Which has the shorter execution time?

- Always?
- Only specific situations or inputs?
  - Switch between algorithms
Which has the shorter execution time?

- **Not** theoretical algorithm execution time
- **Implementation** execution time
Implementation execution time

- Implementation details
- Used library functions execution times
- Used third-party library functions execution times
Questions?
Macro-benchmarking library
“a computer program that measures the ... speed of computer software ... ” [2]

Execution time
Benchmark

- Micro benchmark
  - Short execution times
  - Normally shorter than 0.1 seconds

- Macro benchmark
  - Long execution times
  - Normally longer than 1 second
Macro benchmark

- “QLever SPARQL engine” focus short execution times [1]
- In context: multiple seconds to multiple days
- Macro benchmark better fit
Macro benchmark suite

Set of:
Freestanding macro benchmarks

CLI output excerpt from execution of “BenchmarkExamples.cpp” in “QLever SPARQL engine”, adjusted for readability [1]

Single measurement 'Exponentiate once'
  time: 1e-06s

Single measurement 'Recursively exponentiate multiple times'
  metadata: {
    "amount-of-exponentiations": 100000000000
  }
  time: 7.55948s
Macro benchmarks organized in tables

CLI output excerpt from execution of “BenchmarkExamples.cpp” in “QLever SPARQL engine”, adjusted for readability [1]

<table>
<thead>
<tr>
<th>Basis</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>3</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Time difference</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>2^10</td>
<td>2^11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0000</td>
<td>0.0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Values written out:
- 1024+1024 and 1024+2048
- 2048+2048

Table: 'Adding exponents'
metadata: {"manually-set-fields":"Row 2"}
Freestanding macro benchmarks and tables organized into groups

CLI output excerpt from execution of “BenchmarkExamples.cpp” in “QLever SPARQL engine”, adjusted for readability [1]

Group ‘loopAdd’
metadata: {"operator":"+"}
Measurements:
  Single measurement ’42+69’
    time: 0s

Tables:
  Table ’Addition’
    + 42 24
      42 0.0000 0.0000
      24 0.0000 0.0000
Option 'max-memory' [string]
Value: "0B"
Description: Max amount of memory that any 'IdTable' is allowed to take up. '0' for unlimited memory. When set to anything else than '0', configuration option 'max-bigger-table-rows' is ignored.
Example: 4kB, 8MB, 24B, etc. ...
Option ‘max-time-single-measurement’ [float]
Value: 0.000000
Description: The maximal amount of time, in seconds, any function measurement is allowed to take. ’0’ for unlimited time. Note: This can only be checked, after a measurement was taken.
Required invariants:
- ‘max-time-single-measurement’ must be bigger than, or equal to, 0.
Macro-benchmarking library output

- Display
- “JSON” file [3]
Questions?
Evaluation of my macro-benchmarking library

Example macro benchmark comparison:

- Merge and galloping join algorithm implementation
- Hash join algorithm implementation
Macro benchmark suites executions

- Five different random number generator seeds for the random generation of input table entries
- For every seed 20 executions
- Why repeat seed? Explained later
Sample specifications - Computer hardware

Computer “Ural” from the institute for computer science at the University of Freiburg

- AMD Ryzen 7 3700X 8-Core Processor
- RAM 130 GiB
Macro benchmark suites measurements evaluation focus

- Row ratio
- Sorting configuration
- Hash join speedup
Row ratio

Number rows bigger input table
Number rows smaller input table
Sorting configuration

Sorting status of the smaller and bigger input table before merge and galloping join algorithm implementation
Hash join speedup

Execution time merge and galloping join + Execution time sorting tables

Execution time hash join
Hash join speedup

- Multiple executions of same macro benchmarks with same random number generator seed
- Multiple results of the same hash join speedup
- Approximate hash join speedup mean
Macro benchmark suites measurements evaluations

Approximated speedup mean sorted by sorting configuration

Approximated hash join speedup mean

- None sorted
- Only smaller table sorted
- Only bigger table sorted
- Both sorted

André Schlegel (Univ. Freiburg)
Macro benchmark suites measurements evaluations

None sorted:
Approximated speedup mean sorted by row ratio

Approximated hash join speedup mean

Row ratio

Andre Schlegel (Univ. Freiburg)
Macro-Benchmarking Library
02.08.2024 36/54
Only smaller table sorted:
Approximated speedup mean sorted by row ratio
Macro benchmark suites measurements evaluations

Only bigger table sorted:
Approximated speedup mean sorted by row ratio
Macro benchmark suites evaluation

Hash join algorithm implementation faster than/equal to merge and
galloping join algorithm implementation, when:

- Bigger input table not sorted by join column
- Minimum row ratio $4$
How did my macro-benchmarking library help with building those macro benchmark suites and evaluating their measurements?
Macro benchmark suites broad structure

- Macro benchmark suite for every generalized input table situation
  - Example: Smaller input table number rows grows and row ratio constant
- Tables of macro benchmarks for more specific input table situations
  - Example: Smaller input table number rows grows and row ratio always 10
Macro benchmark suites broad structure

- Every table of macro benchmarks repeated with every sorting configuration
- Every table of macro benchmarks same structure
Macro benchmark suites broad structure

CLI output excerpt from execution of “JoinAlgorithmBenchmark.cpp” in “QLever SPARQL engine”, adjusted for readability [1]

<table>
<thead>
<tr>
<th>Rows in smaller table</th>
<th>Time for sorting</th>
<th>Merge and galloping join</th>
<th>Sorting + merge and galloping join</th>
<th>Hash join</th>
<th>Rows in result table</th>
<th>Hash join speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>0.1493</td>
<td>0.0024</td>
<td>0.1517</td>
<td>0.0142</td>
<td>2962</td>
<td>10.6522</td>
</tr>
<tr>
<td>100000</td>
<td>1.6001</td>
<td>0.0240</td>
<td>1.6241</td>
<td>0.1301</td>
<td>29229</td>
<td>12.4864</td>
</tr>
</tbody>
</table>
Macro benchmark suites broad structure

CLI output excerpt from execution of “JoinAlgorithmBenchmark.cpp” in “QLever SPARQL engine”, adjusted for readability [1]

```json
metadata: {
  "smaller-table-sorted": false,
  "bigger-table-sorted": false,
  "ratio-rows": 100.0
}
```
Macro benchmark suites broad structure

CLI output excerpt from execution of “JoinAlgorithmBenchmark.cpp” in “QLever SPARQL engine”, adjusted for readability [1]

General metadata: {
    "value-changing-with-every-row": "smaller-table-num-rows",
    "overlap-chance": 50.0,
    "random-seed": 1235993526,
    "smaller-table-num-columns": 20,
    "bigger-table-num-columns": 20,
    "max-time-single-measurement": "infinite",
    "max-memory": 20000000000
}
Advantages over other libraries and frameworks:

+ Measurements better organized
Macro benchmark suites broad structure

Advantages over other libraries and frameworks:
+ Measurements better organized
+ Easier identification of behavior over multiple rows
Macro benchmark suites broad structure

Advantages over other libraries and frameworks:

+ Measurements better organized
+ Easier identification of behavior over multiple rows
+ Additional data for easier processing and easier evaluations
Noise

Noise: Time between measured and actual execution time
Algorithms for minimizing noise influence on evaluation require repeated macro benchmark execution

Macro-benchmarking library designed for single execution

Multiple execution with same random generator seed
Advantages over other libraries and frameworks:

+ Easier adjustment parameter
Runtime configuration options

Advantages over other libraries and frameworks:

+ Easier adjustment parameter
+ Easier parameter experimentation
Runtime configuration options

Advantages over other libraries and frameworks:

+ Easier adjustment parameter
+ Easier parameter experimentation
+ Reminder through generated runtime configuration option documentation
Questions?


Appendix
Merge and galloping join algorithm

- Chooses between:
  - Merge join algorithm
  - Galloping join algorithm
Simplified merge join algorithm

- Number rows bigger input table
  Number rows smaller input table < 1000
- Iterate left input table join column
- Linear search right input table join column
Simplified galloping join algorithm

- \[ \frac{\text{Number rows bigger input table}}{\text{Number rows smaller input table}} \geq 1000 \]

- Iterate smaller input table join column

- Binary search bigger input table join column
Setting runtime configuration options

Two ways:

- “JSON” file [3]
- Shorthand configuration string
Setting runtime configuration options - “JSON” file [3]

CLI output excerpt from execution of “BenchmarkExamples.cpp” in “QLever SPARQL engine”, adjusted for readability [1]

```json
{
  "date": "22.3.2023",
  "num-signs": 10000,
  "coin-flip-try": [
    false,
    false
  ],
  "accounts": {
    "personal": {
      "steve": -41.900001525878906
    }
  }
}
```
Setting runtime configuration options - Shorthand configuration string

JSON like syntax for use with CLI.
Setting runtime configuration options - Shorthand configuration string

Simplified definition for the shorthand configuration language from “ConfigShorthand.g4” in “QLever SPARQL engine” [1].

```
shortHandString : assignments EOF;
assignments : (listOfAssignments+=assignment ',')*
    listOfAssignments+=assignment;
assignment : NAME ': ' content;
object : '{' assignments '}'
list : '['(listElement+=content ',')*
    listElement+=content ']'
content : LITERAL|list|object
```
Simplified definition for the shorthand configuration language from “ConfigShorthand.g4” in “QLever SPARQL engine” [1].

```plaintext
// The literals.
LITERAL : BOOL | INTEGER | FLOAT | STRING;
BOOL : 'true' | 'false';
INTEGER : '−'?[0−9]+;
FLOAT : INTEGER'. '[0 −9]+;
STRING : '”’.*?’”’;
NAME : [a-zA-Z0−9\−_]+;
```
Setting runtime configuration options - Shorthand configuration string

Example

date : "2.8.2024", accounts : { steve : 20 }
Macro benchmark suites structure - Table entries generation

Given parameter:

- Number of rows
- Number of columns
- Which column is join column
- Join column sample size ratio
- Probability for overlap
- Seed for random number generator
Macro benchmark suites structure - Table entries generation

Simplified algorithm overview:

1. Create two randomly filled tables with disjoint join columns elements
2. Create join column overlap
Single table entries generation:

1. Fill all columns except join column with random numbers.
2. Define random set as a set of numbers with size “Number of rows” · “Sample size ratio”.
3. Fill join column randomly with elements from the random set using uniform distribution.
Overlap creation:

1. Go through all elements in set of smaller table join column elements.
2. Every element has same given overlap probability for overlap.
3. If no overlap event, nothing happens.
If overlap event:

1. Choose random elements in set of bigger table join column elements with uniform distribution.

2. Replace all occurrences of smaller table join column element with bigger table join column element.
Macro benchmark suites executions - Random generator seed

<table>
<thead>
<tr>
<th>Macro benchmark suit execution</th>
<th>Seed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 to 20</td>
<td>146081905</td>
</tr>
<tr>
<td>21 to 40</td>
<td>193901340</td>
</tr>
<tr>
<td>41 to 60</td>
<td>288613237</td>
</tr>
<tr>
<td>61 to 80</td>
<td>155923003</td>
</tr>
<tr>
<td>81 to 100</td>
<td>4648133</td>
</tr>
</tbody>
</table>
Noise

Causes:

- Computer switches between process
- Benchmark overhead
- Hardware
- Etc.
Noise

- Micro benchmarks noise range: Milliseconds (0.001 seconds)
Noise

- Micro benchmarks noise range: Milliseconds (0.001 seconds)
- Theory: Macro benchmark noise ignorable
Noise

- Micro benchmarks noise range: Milliseconds (0.001 seconds)
- Theory: Macro benchmark noise ignorable
- Reality: Macro benchmark noise can be bigger than one second
Macro benchmark measurement problems:

1. More noise than measurement ⇒ Useless
2. Noticeable difference in repeated macro benchmark measurement ⇒ Need true mean
Noise

Solution: Algorithms for approximating the true mean and identifying measurement with more noise than measurement.
Filtering measurements

- Problem: Measurements with more noise than measurement
- Self-made algorithm for filtering
Reminder: Macro benchmark suites broad structure

<table>
<thead>
<tr>
<th>Rows in smaller table</th>
<th>Time for sorting</th>
<th>Merge and galloping join</th>
<th>Sorting + merge and galloping join</th>
<th>Hash join</th>
<th>Rows in result table</th>
<th>Hash join speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>0.1493</td>
<td>0.0024</td>
<td>0.1517</td>
<td>0.0142</td>
<td>2962</td>
<td>10.6522</td>
</tr>
<tr>
<td>100000</td>
<td>1.6001</td>
<td>0.0240</td>
<td>1.6241</td>
<td>0.1301</td>
<td>29229</td>
<td>12.4864</td>
</tr>
</tbody>
</table>

Table from execution of “JoinAlgorithmBenchmark.cpp” in “QLever SPARQL engine”, adjusted for readability [1].
Filtering measurements - My algorithm

For every list of all executions of same table row with same seed:

1. Max noise = Longest measurement – Shortest measurement
2. Calculate max noise for every macro benchmark in the row
3. If all measurements of single execution smaller than their max noise, delete execution
“1.5 IQR rule” [4]

- List of numbers
- Defines outliers based on number distribution
Filtering measurements - My algorithm

4. Identify hash join speedup outlier via “1.5 IQR rule” [4]

5. If single execution has hash join speedup outlier, delete execution
Mean problem

Normal mean not fit for macro benchmarks
Mean problem

Example macro benchmark A

- Minimum measured execution time A: 1 second
- Maximum measured execution time A: 2 seconds
- Measured execution time A true mean: 1.1 second
Mean problem

Example macro benchmark A

Repeated executions:

1. 1.1 second
2. 2 seconds
3. 1.5 seconds

Mean: $\sim 1.53$
My algorithm for true mean approximation

- Not a general solution
- After filtering of measurements
- Focus on hash join speedup true mean
“99% confidence interval” [5]

- Statistic formula
- Approximates lower/upper bound of true mean of list of numbers
- 99% chance correct
## Reminder: Macro benchmark suites broad structure

<table>
<thead>
<tr>
<th>Rows in smaller table</th>
<th>Time for sorting</th>
<th>Merge and galloping join</th>
<th>Sorting + merge and galloping join</th>
<th>Hash join</th>
<th>Rows in result table</th>
<th>Hash join speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>0.1493</td>
<td>0.0024</td>
<td>0.1517</td>
<td>0.0142</td>
<td>2962</td>
<td>10.6522</td>
</tr>
<tr>
<td>100000</td>
<td>1.6001</td>
<td>0.0240</td>
<td>1.6241</td>
<td>0.1301</td>
<td>29229</td>
<td>12.4864</td>
</tr>
</tbody>
</table>

Table from execution of “JoinAlgorithmBenchmark.cpp” in “QLever SPARQL engine”, adjusted for readability [1].
My algorithm for true mean approximation

For every filtered list of all executions of same table row with same seed:

1. If the list has less than two entries, no return value
2. Calculate hash join speedup “99% confidence interval” lower and upper bound [5]
3. If lower bound less than zero and upper bound bigger than one, no return value
My algorithm for true mean approximation

4. If lower bound less than zero, set lower bound to 0.01
5. Return lower and upper bound
Macro benchmark suites executions - Important runtime configuration options values

- Minimum number smaller input table rows: 10000
- Minimum number bigger input table rows: 100000
- Number columns smaller input table: 20
Macro benchmark suites executions - Important runtime configuration options values

- Number columns bigger input table: 20
- Overlap-chance: 50%
- Sample size ratio join column smaller input table: 1
Macro benchmark suites executions - Important runtime configuration options values

- Sample size ratio join column bigger input table: 1
- Max memory for single table: 20 GB
- Benchmarking sample size ratios join columns: [0.1, 1, 10]
List of compared micro-benchmarking libraries

- “Picobench” [6]
- “Nanobench” [7]
- “Catch2” [8]
- “Sltbench” [9]
- “Folly benchmark component” [10]
- “Hyperfine” [12]
- “Celero” [13]
- “Google benchmark” [14]