Exploiting Code Generation for Efficient LIKE Pattern Matching

Adrian Riedl, Philipp Fent, Maximilian Bandle, Thomas Neumann

Technical University of Munich

ADMS @ VLDB 2023
Strings are everywhere

Require efficient text operations

From: Vogelgesang et al., “Get Real: How Benchmarks Fail to Represent the Real World”
Strings are everywhere

- Require efficient text operations
- Should a DBMS use third-party libraries for string processing?
Strings are everywhere

- Require efficient text operations
- Should a DBMS use third-party libraries for string processing?
- Expensive string conversions due to DBMS specific format
Strings are everywhere

- Require efficient text operations
- Should a DBMS use third-party libraries for string processing?
- Expensive string conversions due to DBMS specific format
- Per-tuple overhead causes significant performance loss
Strings are everywhere

Require efficient text operations
Should a DBMS use third-party libraries for string processing?
Expensive string conversions due to DBMS specific format
Per-tuple overhead causes significant performance loss

We need to integrate text operations better!
select count(*) from uni where name like 'TUM';
Approaches

select count(*) from uni where name like '%TUM%';

Naïve

Tuple Stream

for each tuple

Result

= preprocessing phase

= search phase
**Approaches**

```sql
select count(*) from uni where name like '%%TUM%%';
```

Diagram:
- **Naive**
  - Tuple Stream
  - For each tuple
  - Result

- **Preprocessed**
  - Tuple Stream
  - For each tuple
  - Result

= preprocessing phase
- = search phase
select count(*) from uni where name like '%TUM%';

- **Naïve**
  - Tuple Stream
  - for each tuple

- **Preprocessed**
  - Tuple Stream
  - for each tuple

- **Generated**
  - Tuple Stream
  - for each tuple
  - 100
  - 1010
  - 01

**Result**

- = preprocessing phase
  - = search phase
Knuth-Morris-Pratt Algorithm
From Naïve to Preprocessed

Naïve approach

KMP(text, pattern):
    lpsTable = preprocess(pattern);
    pPos = 0;
    pSize = pattern.size();
    tPos = 0;
    tSize = text.size();
    while (tPos - pPos + pSize <= tSize)
        if (pattern[pPos] == text[tPos])
            pPos++; tPos++;
            if (pPos == pSize) return true;
        else
            shift = lpsTable[pPos];
            if (shift < 0) pPos = 0; tPos++;
            else pPos = shift;
    return false;
Knuth-Morris-Pratt Algorithm
From Naïve to Preprocessed

Naïve approach

KMP(text, pattern):
  lpsTable = preprocess(pattern);
  pPos = 0;
  pSize = pattern.size();
  tPos = 0;
  tSize = text.size();
  while (tPos - pPos + pSize <= tSize)
    if (pattern[pPos] == text[tPos])
      pPos++; tPos++;
      if (pPos == pSize) return true;
    else
      shift = lpsTable[pPos];
      if (shift < 0) pPos = 0; tPos++;
      else pPos = shift;
  return false;
Knuth-Morris-Pratt Algorithm
From Naïve to Preprocessed

Naïve approach

KMP(text, pattern):
    lpsTable = preprocess(pattern);
    pPos = 0;
    pSize = pattern.size();
    tPos = 0;
    tSize = text.size();
    while (tPos - pPos + pSize <= tSize)
        if (pattern[pPos] == text[tPos])
            pPos++; tPos++;
            if (pPos == pSize) return true;
        else
            shift = lpsTable[pPos];
            if (shift < 0) pPos = 0; tPos++;
            else pPos = shift;
    return false;
Knuth-Morris-Pratt Algorithm

From Naïve to Preprocessed

Naïve approach

KMP(text, pattern):
    lpsTable = preprocess(pattern);
    pPos = 0;
    pSize = pattern.size();
    tPos = 0;
    tSize = text.size();
    while (tPos - pPos + pSize <= tSize)
        if (pattern[pPos] == text[tPos])
            pPos++; tPos++;
            if (pPos == pSize) return true;
        else
            shift = lpsTable[pPos];
            if (shift < 0) pPos = 0; tPos++;
            else pPos = shift;
    return false;
Knuth-Morris-Pratt Algorithm
From Naïve to Preprocessed

Naïve approach

KMP(text, pattern):

1. \texttt{lpsTable = preprocess(pattern);} 
2. \texttt{pPos = 0;} 
3. \texttt{pSize = pattern.size();} 
4. \texttt{tPos = 0;} 
5. \texttt{tSize = text.size();} 
6. \texttt{while (tPos - pPos + pSize <= tSize)} 
   1. \texttt{if (pattern[pPos] == text[tPos])} 
      1. \texttt{pPos++; tPos++;} 
      2. \texttt{if (pPos == pSize) return true;} 
   2. \texttt{else} 
      1. \texttt{shift = lpsTable[pPos];} 
      2. \texttt{if (shift < 0) pPos = 0; tPos++;} 
         2. \texttt{else pPos = shift;} 
5. \texttt{return false;}
Knuth-Morris-Pratt Algorithm
From Naïve to Preprocessed

Naïve approach

KMP(text, pattern):

```
lpsTable = preprocess(pattern);
pPos = 0;
pSize = pattern.size();
tPos = 0;
tSize = text.size();
while (tPos - pPos + pSize <= tSize)
  if (pattern[pPos] == text[tPos])
    pPos++; tPos++;
    if (pPos == pSize) return true;
  else
    shift = lpsTable[pPos];
    if (shift < 0) pPos = 0; tPos++;
    else pPos = shift;
return false;
```
Knuth-Morris-Pratt Algorithm
From Naïve to Preprocessed

Naïve approach

KMP(text, pattern):

\[ \text{\textcolor{red}{lpsTable = preprocess(pattern)}}; \]

pPos = 0;
pSize = pattern.size();
tPos = 0;
tSize = text.size();
while (tPos - pPos + pSize <= tSize)
    if (pattern[pPos] == text[tPos])
        pPos++; tPos++;
    if (pPos == pSize) return true;
    else
        shift = lpsTable[pPos];
        if (shift < 0) pPos = 0; tPos++;
        else pPos = shift;
return false;

Preprocessed approach

KMP(text, pattern, \textcolor{red}{lpsTable}):

pPos = 0;
pSize = pattern.size();
tPos = 0;
tSize = text.size();
while (tPos - pPos + pSize <= tSize)
    if (pattern[pPos] == text[tPos])
        pPos++; tPos++;
    if (pPos == pSize) return true;
    else
        shift = lpsTable[pPos];
        if (shift < 0) pPos = 0; tPos++;
        else pPos = shift;
return false;
Knuth-Morris-Pratt Algorithm
From Preprocessed to Generated for ’%TUM%’

Preprocessed approach

KMP(text, pattern, lpsTable):
   pPos = 0;
   pSize = pattern.size();
   tPos = 0;
   tSize = text.size();
   while (tPos - pPos + pSize <= tSize)
      if (pattern[pPos] == text[tPos])
         pPos++; tPos++;
         if (pPos == pSize) return true;
      else
         shift = lpsTable[pPos];
         if (shift < 0) pPos = 0; tPos++;
         else pPos = shift;
   return false;
Preprocessed approach

```
KMP(text, pattern, lpsTable):
  pPos = 0;
  pSize = pattern.size();
  tPos = 0;
  tSize = text.size();
  while (tPos - pPos + pSize <= tSize)
    if (pattern[pPos] == text[tPos])
      pPos++; tPos++;
      if (pPos == pSize) return true;
    else
      shift = lpsTable[pPos];
      if (shift < 0) pPos = 0; tPos++;
      else pPos = shift;
  return false;
```
Knuth-Morris-Pratt Algorithm
From Preprocessed to Generated for '%TUM%'

Preprocessed approach

KMP(text, pattern, lpsTable):

\[
pPos = 0;
\]
\[
pSize = \text{pattern.size()};
\]
\[
tPos = 0;
\]
\[
tSize = \text{text.size();}
\]
\[
\text{while (tPos - pPos + pSize} \leq \text{tSize)}
\]
\[
\text{if (pattern[pPos] == text[tPos])}
\]
\[
pPos++; tPos++;
\]
\[
\text{if (pPos == pSize) return true;}
\]
\[
\text{else}
\]
\[
\text{shift = lpsTable[pPos];}
\]
\[
\text{if (shift < 0) pPos = 0; tPos++;}
\]
\[
\text{else pPos = shift;}
\]
\[
\text{return false;}
\]

Generated approach

whileLoopHeader:
\[
\text{check tPos - pPos + 3} \leq \text{text.size()}
\]
\[
\text{return false}
\]
Knuth-Morris-Pratt Algorithm
From Preprocessed to Generated for ’%TUM%’

Preprocessed approach
KMP(text, pattern, lpsTable):
  pPos = 0;
pSize = pattern.size();
tPos = 0;
tSize = text.size();
  while (tPos - pPos + pSize <= tSize)
    if (pattern[pPos] == text[tPos])
      pPos++; tPos++;
      if (pPos == pSize) return true;
    else
      shift = lpsTable[pPos];
      if (shift < 0) pPos = 0; tPos++;
      else pPos = shift;
  return false;

Generated approach

whileLoopHeader:
  check tPos - pPos + 3 ≤ text.size()
  return false

  check pPos = 0
  check text[tPos + 0] = ‘T’
  check pPos = 1
  check text[tPos + 1] = ‘U’
  check pPos = 2
  check text[tPos + 2] = ‘M’
  unreachable
  return true

performShift:
  shift = ϕ[-1, 0, 0]
  isNegative = shift < 0
  pPos = isNegative ? 0 : shift
  tPos = isNegative ? tPos + 1 : tPos
Knuth-Morris-Pratt Algorithm
From Preprocessed to Generated for ’%TUM%’

Preprocessed approach

KMP(text, pattern, lpsTable):

\[ \text{pPos} = 0; \]
\[ \text{pSize} = \text{pattern.size();} \]
\[ \text{tPos} = 0; \]
\[ \text{tSize} = \text{text.size();} \]
while (tPos - pPos + pSize <= tSize)
if (pattern[pPos] == text[tPos])
    pPos++; tPos++;
if (pPos == pSize) return true;
else
    shift = lpsTable[pPos];
    if (shift < 0) pPos = 0; tPos++;
    else pPos = shift;
return false;

Generated approach

whileLoopHeader:
check tPos - pPos + 3 <= text.size()
return false
check pPos = 0
check text[tPos + 0] = ‘T’
check pPos = 1
check text[tPos + 1] = ‘U’
check pPos = 2
check text[tPos + 2] = ‘M’
unreachable
return true

performShift:
shift = \phi [-1, 0, 0]
isNegative = shift < 0
pPos = isNegative ? 0 : shift
tPos = isNegative ? tPos + 1 : tPos
Evaluation

Multi-threaded performance on Clickbench: url like ’%google%’
Evaluation

Multi-threaded performance on Clickbench: url like '%google%'

Knuth-Morris-Pratt

Boyer-Moore

Two-Way

Hybrid-Search

Throughput [Tuples/s]

Naïve Preprocessed Generated
Hybrid-Search

- Uses SSE instruction: `pcmpistri`
Hybrid-Search

- Uses SSE instruction: `pcmpistri`
- No preprocessing needed

```cpp
HS(text, pattern):
    if (pattern.size() <= 16 && text.size() >= 16)
        iter = text.begin(), end = text.end();
        safeMatch = 17 - pattern.size();
        pattern16 = load16(pattern);
        while ((iter + 16) < end)
            match = `pcmpistri`(pattern16, load16(iter));
            if (match < safeMatch) return true;
            iter += safeMatch;
        if (iter < end)
            match = `pcmpistri`(pattern16, load16(end - 16));
            return match < safeMatch;
    return false;
return fallback(text, pattern);
```
Hybrid-Search

- Uses SSE instruction: `pcmpistri`
- No preprocessing needed
- Consumes 16 bytes of input text at once

```cpp
HS(text, pattern):
    if (pattern.size() <= 16 && text.size() >= 16)
        iter = text.begin(), end = text.end();
        safeMatch = 17 - pattern.size();
        pattern16 = load16(pattern);
        while ((iter + 16) < end)
            match = `pcmpistri(pattern16, load16(iter));`
            if (match < safeMatch) return true;
            iter += safeMatch;
            if (iter < end)
                match = `pcmpistri(pattern16, load16(end - 16));`
                return match < safeMatch;
        return false;
    return fallback(text, pattern);
```
Hybrid-Search

- Uses SSE instruction: pcmpistri
- No preprocessing needed
- Consumes 16 bytes of input text at once
- Generated approach is straightforward

```plaintext
HS(text, pattern):
    if (pattern.size() <= 16 && text.size() >= 16)
        iter = text.begin(), end = text.end();
        safeMatch = 17 - pattern.size();
        pattern16 = load16(pattern);
        while ((iter + 16) < end)
            match = pcmpistri(pattern16, load16(iter));
            if (match < safeMatch) return true;
            iter += safeMatch;
        if (iter < end)
            match = pcmpistri(pattern16, load16(end - 16));
            return match < safeMatch;
        return false;
    return fallback(text, pattern);
```
Hybrid-Search

- Uses SSE instruction: pcmpestri
- No preprocessing needed
- Consumes 16 bytes of input text at once
- Generated approach is straightforward
- Limited to short patterns

```c
HS(text, pattern):
    if (pattern.size() <= 16 && text.size() >= 16)
        iter = text.begin(), end = text.end();
        safeMatch = 17 - pattern.size();
        pattern16 = load16(pattern);
        while ((iter + 16) < end)
            match = pcmpestri(pattern16, load16(iter));
            if (match < safeMatch) return true;
            iter += safeMatch;
        if (iter < end)
            match = pcmpestri(pattern16, load16(end - 16));
            return match < safeMatch;
        return false;
    return fallback(text, pattern);
```
Hybrid-Search

- Uses SSE instruction: pcmpistri
- No preprocessing needed
- Consumes 16 bytes of input text at once
- Generated approach is straightforward
- Limited to short patterns

HS(text, pattern):

if (pattern.size() <= 16 & text.size() >= 16)
    iter = text.begin(), end = text.end();
    safeMatch = 17 - pattern.size();
    pattern16 = load16(pattern);
    while ((iter + 16) < end)
        match = pcmpistri(pattern16, load16(iter));
        if (match < safeMatch) return true;
        iter += safeMatch;
    if (iter < end)
        match = pcmpistri(pattern16, load16(end - 16));
        return match < safeMatch;
    return false;
return fallback(text, pattern);

- Code generation allows to generate code specifically for longer patterns
SSE Search

Example: ’%Technical University of Munich%’
SSE Search

Example: '%Technical University of Munich%'
SSE Search

Example: '%Technical University of Munich%'
SSE Search

Example: ‘%Technical University of Munich%’

whileLoopHeader:
  check \( tPos + 30 \leq \text{text.size()} \)

\( \text{data} = \text{load16(text, tPos)} \)
\( \text{match} = \text{pcmpistri(’Technical Univer’, data)} \)
check match == 0

\( tPos = tPos + 1 \)

return false
SSE Search

Example: '%Technical University of Munich%'
Example: '%Technical University of Munich%'
Evaluation

Long patterns

<table>
<thead>
<tr>
<th>Workload A</th>
<th>Workload B</th>
<th>Workload C</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 characters</td>
<td>160 characters</td>
<td>291 characters</td>
</tr>
<tr>
<td>'%google.ru/arts/searchAutoSearch%'</td>
<td>'%moscow.ru/recipes/&amp;bt=7&amp;bn=1&amp;bodystyle%'</td>
<td>'%id=634575772&amp;ch=UTF-8.../galle/firms/9869%'</td>
</tr>
</tbody>
</table>

Throughput [Tuples/s]

KMP BM TW SSE
Exploiting Code Generation for Efficient LIKE Pattern Matching

Replacing generic function calls with pattern-specific code

Adrian Riedl
Technical University of Munich
Adrian.Riedl@in.tum.de
Exploiting Code Generation for Efficient LIKE Pattern Matching

- Replacing generic function calls with pattern-specific code
- Throughput improvement by up to $2.5 \times$

Adrian Riedl
Technical University of Munich
Adrian.Riedl@in.tum.de
Exploiting Code Generation for Efficient LIKE Pattern Matching

- Replacing generic function calls with pattern-specific code
- Throughput improvement by up to $2.5 \times$
- Generating specialized code with SSE instructions for longer pattern

Adrian Riedl
Technical University of Munich
Adrian.Riedl@in.tum.de