On-the-Fly Token Similarity Joins in Relational Databases

N. Augsten\textsuperscript{1}  A. Miraglia\textsuperscript{2}  T. Neumann\textsuperscript{3}  A. Kemper\textsuperscript{3}

\textsuperscript{1}University of Salzburg, Austria
nikolaus.augsten@sbg.ac.at

\textsuperscript{2}VU University Amsterdam, Netherlands
a.miraglia@student.vu.nl

\textsuperscript{3}TU München, Germany
\{neumann,kemper\}@in.tum.de

June 26, 2014
SIGMOD, Snowbird, Utah
Outline

1 Motivation

2 The Tokenize Operator
   - Efficient Implementation
   - Query Optimization

3 Experiments
Motivation

Outline

1. Motivation

2. The Tokenize Operator
   - Efficient Implementation
   - Query Optimization

3. Experiments
## Token Similarity Join

**Motivation**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>snowbird</td>
<td>snowbasin</td>
</tr>
<tr>
<td>canyons</td>
<td>snowbirds</td>
</tr>
</tbody>
</table>

\[
\mathcal{R} \setminus \text{sim}(A, B) \geq 70\%
\]

\[
\text{sim}(A, B) = \frac{|\alpha(A) \cap \alpha(B)|}{|\alpha(A) \cup \alpha(B)|}
\]
**Token Similarity Join**

\[ \forall \sim(A, B) \geq 70\% \]

\[ \sim(A, B) = \frac{|\alpha(A) \cap \alpha(B)|}{|\alpha(A) \cup \alpha(B)|} \]

\[ \text{sim}(A, B) = \frac{5}{14} = 36\% \]
Token Similarity Join

\[ \text{Motivation} \]

\[ \begin{align*}
\mathsf{A} & \quad \text{snowbird} \\
\text{canyons} & \quad \ldots
\end{align*} \]

\[ \begin{align*}
\mathsf{B} & \quad \text{snowbasin} \\
\text{snowbirds} & \quad \ldots
\end{align*} \]

\[ \forall \text{sim}(A, B) \geq 70\% \]

\[ \text{sim}(A, B) = \frac{|\alpha(A) \cap \alpha(B)|}{|\alpha(A) \cup \alpha(B)|} \]

\[ \begin{align*}
\text{sim}(A, B) &= \frac{5}{14} = 36\% \quad \times \\
\text{sim}(A, B) &= \frac{8}{11} = 73\% \quad \checkmark
\end{align*} \]
Motivation

Token Generation in Similarity Joins

\[ R \sim \Join S \]

\( R \)

\[ \text{generate token} \]

\( S \)

\[ \text{generate token} \]

\[ \text{set similarity join on tokens} \]

Part-Enum (VLDB'06)

All-Pairs (WWW'07)

PP-Join (WWW'08)

MP-Join (Inf. Syst.'11)

Adapt-Join (SIGMOD'12)

\[ \ldots \]

Goal: integrate token generation into query plan!
Motivation

Token Generation in Similarity Joins

$R \xrightarrow{\text{generate token}} R \sim S \xleftarrow{\text{set similarity join on tokens}} S \xrightarrow{\text{generate token}}$

Well studied

- Part-Enum (VLDB’06)
- All-Pairs (WWW’07)
- PP-Join (WWW’08)
- MP-Join (Inf. Syst.’11)
- Adapt-Join (SIGMOD’12)
- ...
Token Generation in Similarity Joins

**Motivation**

- Received little attention
- Well studied

\[ R \xrightarrow{\text{generate token}} R \sim \bowtie S \xrightarrow{\text{set similarity join on tokens}} R \sim \bowtie S \]

- Precomputed tokens assumed
- Token generation not part of query plan

- Part-Enum (VLDB’06)
- All-Pairs (WWW’07)
- PP-Join (WWW’08)
- MP-Join (Inf. Syst.’11)
- Adapt-Join (SIGMOD’12)
- ...
Token Generation in Similarity Joins

Recived little attention

$R$\ 
\[\rightarrow\]

generate token

$S$\ 
\[\rightarrow\]

generate token

Well studied

set similarity join on tokens

\[R \sim \Join S\]

Goal: integrate token generation into query plan!
Generating Tokens

- **Stand-alone client:** export data, generate tokens, import tokens
  - overhead for export/import
  - no integration into query plan
  - only good for precomputation
Generating Tokens

- **Stand-alone client:** export data, generate tokens, import tokens
  - overhead for export/import
  - no integration into query plan
  - only good for precomputation

- **Table function:**
  - UDF generates tokens on-the-fly
  - table function used like a table in query
Motivation

State-of-the-Art: Table Function

Customer tables $R, S$:
- join customers with similar names
- only customers from 'SLC' and 'Salt Lake'

```sql
SELECT TR.ssn, TS.ssn
FROM tblfunc('R', 'name') TR,
     tblfunc('S', 'name') TS
WHERE TR.city = 'SLC'
     AND TS.county = 'Salt Lake'
     AND TR.token = TS.token
GROUP BY TR.ssn, TS.ssn
HAVING COUNT(*) > k;
```
State-of-the-Art: Table Function

Customer tables $R$, $S$:
- join customers with similar names
- only customers from 'SLC' and 'Salt Lake'

```sql
SELECT TR.ssn, TS.ssn
FROM tblfunc('R', 'name') TR,
     tblfunc('S', 'name') TS
WHERE TR.city = 'SLC'
AND TS.county='Salt Lake'
AND TR.token = TS.token
GROUP BY TR.ssn, TS.ssn
HAVING COUNT(*) >= k;
```
Motivation

State-of-the-Art: Table Function

Customer tables $R, S$:

- join customers with similar names
- only customers from 'SLC' and 'Salt Lake'

\[
\begin{align*}
\text{SELECT} & \quad \pi TR.ss, TS.ss \\
\text{FROM} & \quad \text{tblfunc('R', 'name')} TR, \\
& \quad \text{tblfunc('S', 'name')} TS \\
\text{WHERE} & \quad \sigma \text{city} = 'SLC' \\
& \quad \text{AND} \quad \text{TS.county} = 'Salt Lake' \\
& \quad \text{AND} \quad TR.token = TS.token \\
\text{GROUP BY} & \quad TR.ss, TS.ss \\
\text{HAVING} & \quad \text{COUNT(*)} \geq k;
\end{align*}
\]
Motivation

State-of-the-Art: Table Function

Customer tables $R, S$:
- join customers with similar names
- only customers from 'SLC' and 'Salt Lake'

```
SELECT TR.ssn, TS.ssn
FROM tblfunc('R', 'name') TR,
tblfunc('S', 'name') TS
WHERE TR.city = 'SLC'
AND TS.county = 'Salt Lake'
AND TR.token = TS.token
GROUP BY TR.ssn, TS.ssn
HAVING COUNT(*) >= k;
```

- **black box**: selection and projection not pushed down
  - tokens computed for all customers
  - too many attributes replicated (name, city, county)

Table Function Sim. Join in RDBs

<table>
<thead>
<tr>
<th>Table</th>
<th>Rows</th>
<th>Est. Card.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R$</td>
<td>20,000</td>
<td>1,000</td>
</tr>
<tr>
<td>$S$</td>
<td>10,000</td>
<td>1,000</td>
</tr>
</tbody>
</table>

```
\begin{align*}
\pi_{TR.ssn, TS.ssn} \\
\sigma_{\text{olap} \geq k} \\
\gamma_{TR.ssn, TS.ssn; \text{COUNT}(\ast) \to \text{olap}} \\
\times_{\sigma_{\text{city} = \text{SLC}} [1\%] \quad \sigma_{\text{county} = \text{Salt Lake}} [5\%]} \\
\text{tblfunc/TR} : 20,000 \quad \text{tblfunc/TS} : 10,000 \\
R(\text{ssn, name, city}) \quad S(\text{ssn, name, county})
\end{align*}
```
Motivation

State-of-the-Art: Table Function

Customer tables $R$, $S$:
- join customers with similar names
- only customers from 'SLC' and 'Salt Lake'

```
SELECT TR.ssn, TS.ssn
FROM tblfunc('R', 'name') TR,
     tblfunc('S', 'name') TS
WHERE TR.city = 'SLC'
AND TS.county='Salt Lake'
AND TR.token = TS.token
GROUP BY TR.ssn, TS.ssn
HAVING COUNT(*) >= k;
```

- **black box**: selection and projection not pushed down
  - tokens computed for all customers
  - too many attributes replicated (name, city, county)
- **unknown cardinality** of table function (often assumed a constant)
State-of-the-Art: Table Function

Customer tables $R, S$:
- join customers with similar names
- only customers from 'SLC' and 'Salt Lake'

$$\text{SELECT TR.ssn, TS.ssn}$$
$$\text{FROM tblfunc('R', 'name') TR,}$$
$$\text{tblfunc('S', 'name') TS}$$
$$\text{WHERE TR.city = 'SLC'}$$
$$\text{AND TS.county='Salt Lake'}$$
$$\text{AND TR.token = TS.token}$$
$$\text{GROUP BY TR.ssn, TS.ssn}$$
$$\text{HAVING COUNT(*)} \geq k;$$

Problem: poor query plans with table functions.
Outline

1. Motivation

2. The Tokenize Operator
   - Efficient Implementation
   - Query Optimization

3. Experiments
Tokenize $\tau$ is a relational operator defined as follows:

$$\tau_{\alpha}(A)(R) = \{ r \circ tk \mid r \in R, tk \in \alpha(t.A) \}$$
Tokenize $\tau$ is a relational operator defined as follows:

$$\tau_{\alpha}(A)(R) = \{ r \circ tk \mid r \in R, tk \in \alpha(t.A) \}$$

- $R$ is a relation, $A \subseteq \text{schema}(R)$ is a sequence of attributes.
The Tokenize Operator

Tokenize $\tau$ is a relational operator defined as follows:

$$\tau_\alpha(A)(R) = \{ r \circ tk \mid r \in R, tk \in \alpha(t.A) \}$$

- $R$ is a relation, $A \subseteq \text{schema}(R)$ is a sequence of attributes.
- $\alpha$ is token function
  - computes tokens for a single value (e.g., q-grams for a string)

\[
\begin{array}{|c|}
\hline
A \\
\hline
\text{snowbird} \\
\text{canyons} \\
\ldots \\
\hline
\end{array}
\]
**Solution: Tokenize Operator**

**Tokenize** \( \tau \) is a relational operator defined as follows:

\[
\tau_\alpha(A)(R) = \{ r \circ tk \mid r \in R, tk \in \alpha(t.A) \}
\]

- \( R \) is a relation, \( A \subseteq \text{schema}(R) \) is a sequence of attributes.
- \( \alpha \) is token function
  - computes tokens for a single value (e.g., q-grams for a string)
- **Output**: relation with tokens

\[
\begin{array}{|c|c|}
\hline
A & \text{token} \\
\hline
\text{snowbird} & \#s \\
\text{canyons} & \#c \\
\text{...} & \text{...} \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
A & \text{token} \\
\hline
\text{snowbird} & \text{sn} \\
\text{snowbird} & \text{no} \\
\text{...} & \text{...} \\
\text{canyons} & \text{ca} \\
\text{canyons} & \text{an} \\
\text{...} & \text{...} \\
\hline
\end{array}
\]
**Solution: Tokenize Operator**

Tokenize $\tau$ is a relational operator defined as follows:

$$\tau_\alpha(A)(R) = \{ r \circ tk \mid r \in R, tk \in \alpha(t.A) \}$$

- $R$ is a relation, $A \subseteq \text{schema}(R)$ is a sequence of attributes.
- $\alpha$ is token function
  - computes tokens for a single value (e.g., q-grams for a string)
- Output: relation with tokens

\[
\begin{array}{|c|}
\hline
\text{A} \\
\text{snowbird} \\
\text{canyons} \\
\vdots \\
\hline
\end{array}
\quad \tau_\alpha(A) \quad \begin{array}{|c|}
\hline
\text{A} & \text{token} \\
\text{snowbird} & \#s \\
\text{snowbird} & \text{sn} \\
\text{snowbird} & \text{no} \\
\vdots & \vdots \\
\text{canyons} & \#c \\
\text{canyons} & \text{ca} \\
\text{canyons} & \text{an} \\
\vdots & \vdots \\
\hline
\end{array}
\]

- Note:
  - $\alpha$ is type specific (like MAX, AVG)
  - tokenize $\tau$ abstracts from types (like GROUP)
Outline

1 Motivation

2 The Tokenize Operator
   • Efficient Implementation
   • Query Optimization

3 Experiments
Virtual Tuple Replication/1

\[ R \]

<table>
<thead>
<tr>
<th>name</th>
<th>role</th>
<th>conference</th>
</tr>
</thead>
<tbody>
<tr>
<td>curtis dyreson</td>
<td>general chair</td>
<td>sigmod</td>
</tr>
<tr>
<td>feifei li</td>
<td>general chair</td>
<td>sigmod</td>
</tr>
<tr>
<td>tamer ozsu</td>
<td>program chair</td>
<td>sigmod</td>
</tr>
</tbody>
</table>

The Tokenize Operator

Efficient Implementation

\[ \tau \alpha(\text{name}) \rightarrow \text{token} \]

\[ \text{GID}(\text{R}) \]

physical replication is expensive

VTR avoids physical replication

1. mark group with GID
2. create VTR bit array
3. keep single bit for each replicated attribute

bit array + grouping restores original values

expensive!
### Virtual Tuple Replication

**R**

<table>
<thead>
<tr>
<th>name</th>
<th>role</th>
<th>conference</th>
</tr>
</thead>
<tbody>
<tr>
<td>curtis dyreson</td>
<td>general chair</td>
<td>sigmod</td>
</tr>
<tr>
<td>feifei li</td>
<td>general chair</td>
<td>sigmod</td>
</tr>
<tr>
<td>tamer o兹su</td>
<td>program chair</td>
<td>sigmod</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

\( \tau_{\alpha}(name) \rightarrow \text{token}(R) \)

<table>
<thead>
<tr>
<th>name</th>
<th>role</th>
<th>conference</th>
<th>token</th>
</tr>
</thead>
<tbody>
<tr>
<td>curtis dyreson</td>
<td>general chair</td>
<td>sigmod</td>
<td>#c</td>
</tr>
<tr>
<td>curtis dyreson</td>
<td>general chair</td>
<td>sigmod</td>
<td>cu</td>
</tr>
<tr>
<td>curtis dyreson</td>
<td>general chair</td>
<td>sigmod</td>
<td>ur</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>feifei li</td>
<td>general chair</td>
<td>sigmod</td>
<td>#f</td>
</tr>
<tr>
<td>feifei li</td>
<td>general chair</td>
<td>sigmod</td>
<td>fe</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>tamer o兹su</td>
<td>program chair</td>
<td>sigmod</td>
<td>#t</td>
</tr>
<tr>
<td>tamer o兹su</td>
<td>program chair</td>
<td>sigmod</td>
<td>ta</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
---

## Virtual Tuple Replication

### Table: Relation R

<table>
<thead>
<tr>
<th>name</th>
<th>role</th>
<th>conference</th>
</tr>
</thead>
<tbody>
<tr>
<td>curtis dyreson</td>
<td>general chair</td>
<td>sigmod</td>
</tr>
<tr>
<td>feifei li</td>
<td>general chair</td>
<td>sigmod</td>
</tr>
<tr>
<td>tamer ozsu</td>
<td>program chair</td>
<td>sigmod</td>
</tr>
</tbody>
</table>

### Table: Tokenization $\tau_{\alpha}(name) \rightarrow \text{token}(R)$

<table>
<thead>
<tr>
<th>name</th>
<th>role</th>
<th>conference</th>
<th>token</th>
</tr>
</thead>
<tbody>
<tr>
<td>curtis dyreson</td>
<td>general chair</td>
<td>sigmod</td>
<td>#c</td>
</tr>
<tr>
<td>curtis dyreson</td>
<td>general chair</td>
<td>sigmod</td>
<td>cu</td>
</tr>
<tr>
<td>curtis dyreson</td>
<td>general chair</td>
<td>sigmod</td>
<td>ur</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>feifei li</td>
<td>general chair</td>
<td>sigmod</td>
<td>#f</td>
</tr>
<tr>
<td>feifei li</td>
<td>general chair</td>
<td>sigmod</td>
<td>fe</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>tamer ozsu</td>
<td>program chair</td>
<td>sigmod</td>
<td>#t</td>
</tr>
<tr>
<td>tamer ozsu</td>
<td>program chair</td>
<td>sigmod</td>
<td>ta</td>
</tr>
</tbody>
</table>

---

**Efficient Implementation**

- **Virtual Tuple Replication (VTR)**
  - Mark group with GID
  - Create VTR bit array
  - Keep single bit for each replicated attribute
  - Bit array + grouping restores original values

- **Physical Replication is Expensive**

---

Nikolaus Augsten (Salzburg, Austria)  
On-the-Fly Token Sim. Join in RDBs  
SIGMOD 14 – Snowbird, UT
### Virtual Tuple Replication

<table>
<thead>
<tr>
<th>name</th>
<th>role</th>
<th>conference</th>
</tr>
</thead>
<tbody>
<tr>
<td>curtis dyreson</td>
<td>general chair</td>
<td>sigmod</td>
</tr>
<tr>
<td>feifei li</td>
<td>general chair</td>
<td>sigmod</td>
</tr>
<tr>
<td>tamer ozsu</td>
<td>program chair</td>
<td>sigmod</td>
</tr>
</tbody>
</table>

\[ \tau_{\alpha}(\text{name} \rightarrow \text{token})(R) \]

<table>
<thead>
<tr>
<th>name</th>
<th>role</th>
<th>conference</th>
<th>token</th>
</tr>
</thead>
<tbody>
<tr>
<td>curtis dyreson</td>
<td>general chair</td>
<td>sigmod</td>
<td>#c</td>
</tr>
<tr>
<td>curtis dyreson</td>
<td>general chair</td>
<td>sigmod</td>
<td>cu</td>
</tr>
<tr>
<td>curtis dyreson</td>
<td>general chair</td>
<td>sigmod</td>
<td>ur</td>
</tr>
<tr>
<td>feifei li</td>
<td>general chair</td>
<td>sigmod</td>
<td>#f</td>
</tr>
<tr>
<td>feifei li</td>
<td>general chair</td>
<td>sigmod</td>
<td>fe</td>
</tr>
<tr>
<td>tamer ozsu</td>
<td>program chair</td>
<td>sigmod</td>
<td>#t</td>
</tr>
<tr>
<td>tamer ozsu</td>
<td>program chair</td>
<td>sigmod</td>
<td>ta</td>
</tr>
</tbody>
</table>

- Physical replication is expensive
- VTR avoids physical replication
### Virtual Tuple Replication

#### $R$

<table>
<thead>
<tr>
<th>name</th>
<th>role</th>
<th>conference</th>
</tr>
</thead>
<tbody>
<tr>
<td>curtis dyreson</td>
<td>general chair</td>
<td>sigmod</td>
</tr>
<tr>
<td>feifei li</td>
<td>general chair</td>
<td>sigmod</td>
</tr>
<tr>
<td>tamer ozsu</td>
<td>program chair</td>
<td>sigmod</td>
</tr>
</tbody>
</table>

#### $\tau_\alpha(name) \rightarrow \text{token, } GID(R)$

<table>
<thead>
<tr>
<th>name</th>
<th>role</th>
<th>conference</th>
<th>token</th>
<th>GID</th>
</tr>
</thead>
<tbody>
<tr>
<td>curtis dyreson</td>
<td>general chair</td>
<td>sigmod</td>
<td>#c</td>
<td>1</td>
</tr>
<tr>
<td>curtis dyreson</td>
<td>general chair</td>
<td>sigmod</td>
<td>cu</td>
<td>1</td>
</tr>
<tr>
<td>curtis dyreson</td>
<td>general chair</td>
<td>sigmod</td>
<td>ur</td>
<td>1</td>
</tr>
<tr>
<td>feifei li</td>
<td>general chair</td>
<td>sigmod</td>
<td>#f</td>
<td>2</td>
</tr>
<tr>
<td>feifei li</td>
<td>general chair</td>
<td>sigmod</td>
<td>fe</td>
<td>2</td>
</tr>
<tr>
<td>tamer ozsu</td>
<td>program chair</td>
<td>sigmod</td>
<td>#t</td>
<td>3</td>
</tr>
<tr>
<td>tamer ozsu</td>
<td>program chair</td>
<td>sigmod</td>
<td>ta</td>
<td>3</td>
</tr>
</tbody>
</table>

- Physical replication is expensive
- VTR avoids physical replication
  1. mark group with GID

VTR Bit Array:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

On-the-Fly Token Sim. Join in RDBs

SIGMOD 14 – Snowbird, UT
### Virtual Tuple Replication

The **Tokenize Operator** (VTR) is an efficient method for implementing tuple replication in relational databases. VTR avoids physical tuple replication, which is expensive, by using a bit array and grouping to keep track of replicated attributes. Here’s how it works:

1. **Mark Group with GID**
2. **Create VTR Bit Array**
3. **Keep Single Bit for Each Replicated Attribute**

#### Example

Consider the relation `R` with attributes `name`, `role`, and `conference`:

<table>
<thead>
<tr>
<th>name</th>
<th>role</th>
<th>conference</th>
</tr>
</thead>
<tbody>
<tr>
<td>curtis dyreson</td>
<td>general chair</td>
<td>sigmod</td>
</tr>
<tr>
<td>feifei li</td>
<td>general chair</td>
<td>sigmod</td>
</tr>
<tr>
<td>tamer ozu</td>
<td>program chair</td>
<td>sigmod</td>
</tr>
</tbody>
</table>

When applying the `τα(name)→token; GID(R)` operation, the relation is transformed as follows:

<table>
<thead>
<tr>
<th>name</th>
<th>role</th>
<th>conference</th>
<th>token</th>
<th>GID</th>
</tr>
</thead>
<tbody>
<tr>
<td>curtis dyreson</td>
<td>general chair</td>
<td>sigmod</td>
<td>#c</td>
<td>1</td>
</tr>
<tr>
<td>curtis dyreson</td>
<td>general chair</td>
<td>sigmod</td>
<td>cu</td>
<td>1</td>
</tr>
<tr>
<td>curtis dyreson</td>
<td>general chair</td>
<td>sigmod</td>
<td>ur</td>
<td>1</td>
</tr>
<tr>
<td>feifei li</td>
<td>general chair</td>
<td>sigmod</td>
<td>#f</td>
<td>2</td>
</tr>
<tr>
<td>feifei li</td>
<td>general chair</td>
<td>sigmod</td>
<td>fe</td>
<td>2</td>
</tr>
<tr>
<td>tamer ozu</td>
<td>program chair</td>
<td>sigmod</td>
<td>#t</td>
<td>3</td>
</tr>
<tr>
<td>tamer ozu</td>
<td>program chair</td>
<td>sigmod</td>
<td>ta</td>
<td>3</td>
</tr>
</tbody>
</table>

The **VTR Bit Array** shows the replication status:

<table>
<thead>
<tr>
<th>VTR Bit Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1</td>
</tr>
<tr>
<td>1 1 1 1 1</td>
</tr>
<tr>
<td>1 1 1 1 1</td>
</tr>
</tbody>
</table>

This method allows for efficient tuple replication without the overhead of physical duplication.
Virtual Tuple Replication

### $R$

<table>
<thead>
<tr>
<th>name</th>
<th>role</th>
<th>conference</th>
</tr>
</thead>
<tbody>
<tr>
<td>curtis dyreson</td>
<td>general chair</td>
<td>sigmod</td>
</tr>
<tr>
<td>feifei li</td>
<td>general chair</td>
<td>sigmod</td>
</tr>
<tr>
<td>tamer ozsu</td>
<td>program chair</td>
<td>sigmod</td>
</tr>
</tbody>
</table>

- physical replication is expensive
- VTR avoids physical replication
  1. mark group with GID
  2. create VTR bit array
  3. keep single bit for each replicated attribute

\[ \tau \alpha (name) \rightarrow \text{token}; \text{GID} (R) \]

<table>
<thead>
<tr>
<th>name</th>
<th>role</th>
<th>conference</th>
<th>token</th>
<th>GID</th>
</tr>
</thead>
<tbody>
<tr>
<td>curtis dyreson</td>
<td>general chair</td>
<td>sigmod</td>
<td>#c</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cu</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ur</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>feifei li</td>
<td>general chair</td>
<td>sigmod</td>
<td>#f</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>fe</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tamer ozsu</td>
<td>program chair</td>
<td>sigmod</td>
<td>#t</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ta</td>
<td>3</td>
</tr>
</tbody>
</table>

VTR Bit Array

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**On-the-Fly Token Sim. Join in RDBs**
### Virtual Tuple Replication

#### R

<table>
<thead>
<tr>
<th>name</th>
<th>role</th>
<th>conference</th>
</tr>
</thead>
<tbody>
<tr>
<td>curtis dyreson</td>
<td>general chair</td>
<td>sigmod</td>
</tr>
<tr>
<td>feifei li</td>
<td>general chair</td>
<td>sigmod</td>
</tr>
<tr>
<td>tamer ozsu</td>
<td>program chair</td>
<td>sigmod</td>
</tr>
</tbody>
</table>

- Physical replication is expensive
- VTR avoids physical replication
  1. mark group with GID
  2. create VTR bit array
  3. keep single bit for each replicated attribute
- Bit array + grouping restores original values

#### $\tau_{\alpha(name)\rightarrow token;GID}(R)$

<table>
<thead>
<tr>
<th>name</th>
<th>role</th>
<th>conference</th>
<th>token</th>
<th>GID</th>
</tr>
</thead>
<tbody>
<tr>
<td>curtis dyreson</td>
<td>general chair</td>
<td>sigmod</td>
<td>#c</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>↓</td>
<td>↓</td>
<td>cu</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>↓</td>
<td>↓</td>
<td>ur</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>↓</td>
<td>↓</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>feifei li</td>
<td>general chair</td>
<td>sigmod</td>
<td>#f</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>↓</td>
<td>↓</td>
<td>fe</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>↓</td>
<td>↓</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>tamer ozsu</td>
<td>program chair</td>
<td>sigmod</td>
<td>#t</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>↓</td>
<td>↓</td>
<td>ta</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>↓</td>
<td>↓</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VTR Bit Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1</td>
</tr>
<tr>
<td>0 0 0 1 1</td>
</tr>
<tr>
<td>0 0 0 1 1</td>
</tr>
<tr>
<td>1 1 1 1 1</td>
</tr>
<tr>
<td>0 0 0 1 1</td>
</tr>
<tr>
<td>1 1 1 1 1</td>
</tr>
<tr>
<td>0 0 0 1 1</td>
</tr>
</tbody>
</table>

#### VTR Bit Array

| 1 1 1 1 1 |
| 0 0 0 1 1 |
| 0 0 0 1 1 |
| 1 1 1 1 1 |
| 0 0 0 1 1 |
| 1 1 1 1 1 |
| 0 0 0 1 1 |
Example: count number of tokens for each name

\[ \gamma_{\text{name}}; \text{COUNT}(\star) \rightarrow \text{cnt}(\tau_{\alpha}(\text{name})(R)) \]

- name value replicated for each token
- replicated values removed by grouping operator
Example: count number of tokens for each name

\[ \gamma_{name}; \text{COUNT}(\tau_{\alpha}(name)) \rightarrow \text{cnt}(R) \]

- name value replicated for each token
- replicated values removed by grouping operator

VTR version of query:

\[ \gamma \text{ COUNT}(\tau_{\alpha}(name) \rightarrow tk (R)) \]
Example: count number of tokens for each name

\[ \gamma_{name}; \text{COUNT}(\ast) \rightarrow c\text{nt}(\tau_{\alpha}(\text{name})(R)) \]

- name value replicated for each token
- replicated values removed by grouping operator

VTR version of query:

\[ \gamma_{GID}; \text{COUNT}(\ast) \rightarrow c\text{nt}(\tau_{\alpha}(\text{name} \rightarrow tk; GID)(R)) \]

- generate and group by GID
Example: count number of tokens for each `name`

\[ \gamma_{name}; \text{COUNT}(\ast) \rightarrow \text{cnt}(\tau_{\alpha}(name)(R)) \]

- `name` value replicated for each token
- Replicated values removed by grouping operator

VTR version of query:

\[ \gamma_{GID}; \text{REST}(name), \text{COUNT}(\ast) \rightarrow \text{cnt}(\tau_{\alpha}(name) \rightarrow tk; GID(R)) \]

- Generate and group by `GID`
- Restore `name` attribute
Example: count number of tokens for each name

\[ \gamma_{name}; \text{COUNT}(\ast) \rightarrow 
\text{cnt}\left(\tau_{\alpha}(name)(R)\right) \]

- name value replicated for each token
- replicated values removed by grouping operator

VTR version of query:

\[ \pi_{name, cnt}\left( \gamma_{GID}; \text{REST(name)}, \text{COUNT}(\ast) \rightarrow 
\text{cnt}\left(\tau_{\alpha}(name) \rightarrow \tau_{tk}; GID(R)\right) \right) \]

- generate and group by GID
- restore name attribute
- remove GID
The Tokenize Operator

Virtual Tuple Replication/2

- **Example**: count number of tokens for each \(name\)

\[ \gamma_{name}; \text{COUNT}(*) \rightarrow \text{cnt}(\tau_{\alpha}(name)(R)) \]

- \(name\) value replicated for each token
- replicated values removed by grouping operator

- **VTR version** of query:

\[ \pi_{name, cnt}(\gamma_{GID}; \text{REST}(name), \text{COUNT}(*) \rightarrow \text{cnt}(\tau_{\alpha}(name) \rightarrow \text{tk}; GID(R))) \]

- generate and group by \(GID\)
- restore \(name\) attribute
- remove \(GID\)

VTR: efficient implementation of tokenize.
Outline

1 Motivation

2 The Tokenize Operator
   - Efficient Implementation
   - Query Optimization

3 Experiments
Equivalence transformations with tokenize:

- **push down selection**: \( \text{attr}(\theta) \subseteq \text{schema}(R) \)

\[
\sigma_\theta(\tau_\alpha(A)(R)) = \tau_\alpha(A)(\sigma_\theta(R))
\]

- **push down projection**: \( A \subseteq B \)

\[
\pi_{BA'}(\tau_\alpha(A) \rightarrow A'(R)) = \tau_\alpha(A) \rightarrow A'(\pi_B(R))
\]

- **reorder with join**: \( A \subseteq \text{schema}(R) \)

\[
\tau_\alpha(A)(R \bowtie_\theta S) = \tau_\alpha(A)(R) \bowtie_\theta S
\]

- **reorder tokenize operators**: \( A, B \subseteq \text{schema}(R) \)

\[
\tau_\alpha(A)\tau_\alpha(B)(R) = \tau_\alpha(B)\tau_\alpha(A)(R)
\]
Cardinality Estimation

- **Cardinality estimation** for tokenize:

\[ |\tau_{\alpha}(A)(R)| = |R| \times |\alpha(A)|_{\text{avg}} \]

- Most token functions \( \alpha \) produce **linear number of tokens**
  - \Rightarrow accurate cardinality estimates
Cardinality Estimation

- **Cardinality estimation** for tokenize:

\[ |\tau_{\alpha}(A)(R)| = |R| \times |\alpha(A)|_{\text{avg}} \]

- Most token functions \( \alpha \) produce **linear number of tokens**
  \[ \Rightarrow \] accurate cardinality estimates

<table>
<thead>
<tr>
<th>Token</th>
<th>Type</th>
<th>Statistics</th>
<th>Cardinality</th>
</tr>
</thead>
<tbody>
<tr>
<td>q-grams</td>
<td>string</td>
<td>avg. string length ( \bar{s} )</td>
<td>(</td>
</tr>
<tr>
<td>binar branches</td>
<td>tree</td>
<td>avg. node number ( \bar{t} )</td>
<td>(</td>
</tr>
<tr>
<td>pq-grams</td>
<td>tree</td>
<td>avg. node number ( \bar{t} )</td>
<td>(</td>
</tr>
</tbody>
</table>
Cardinality Estimation

- **Cardinality estimation** for tokenize:

\[ |\tau_{\alpha}(A)(R)| = |R| \times |\alpha(A)|_{avg} \]

- Most token functions \( \alpha \) produce **linear number of tokens**
  \( \Rightarrow \) accurate cardinality estimates

<table>
<thead>
<tr>
<th>Token</th>
<th>Type</th>
<th>Statistics</th>
<th>Cardinality</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q )-grams</td>
<td>string</td>
<td>avg. string length ( \bar{s} )</td>
<td>(</td>
</tr>
<tr>
<td>binar branches</td>
<td>tree</td>
<td>avg. node number ( \bar{t} )</td>
<td>(</td>
</tr>
<tr>
<td>( pq )-grams</td>
<td>tree</td>
<td>avg. node number ( \bar{t} )</td>
<td>(</td>
</tr>
</tbody>
</table>

Simple and accurate cardinality estimates for tokenize.
SELECT R.ssn, S.ssn
FROM R, S
TOKENIZE
  R ON name AS R_token,
  S ON name AS S_token
WHERE R.city = 'SLC'
  AND S.county='Salt Lake'
  AND R_token = S_token
GROUP BY R.ssn, S.ssn
HAVING COUNT(*) >= k;
The Tokenize Operator

query optimization

query plans with tokenize

SELECT R.ssn, S.ssn
FROM R, S
TOKENIZE
  R ON name AS R_token,
  S ON name AS S_token
WHERE R.city = 'SLC'
  AND S.county='Salt Lake'
  AND R_token = S_token
GROUP BY R.ssn, S.ssn
HAVING COUNT(*) >= k;

π R.ssn, S.ssn
σ olap ≥ k
γ R.ssn, S.ssn; COUNT(*) → olap
R_token = S_token
σ city = 'SLC' [1%]
σ county = 'Salt Lake' [5%]
τα(name) → R_token
20'000
R(ssn, name, city)
200'000
100'000
τα(name) → S_token
5'000
S(ssn, name, county)
500
2'000
true cardinality

Nikolaus Augsten (Salzburg, Austria)
On-the-Fly Token Sim. Join in RDBs
SIGMOD 14 – Snowbird, UT
SELECT R.ssn, S.ssn
FROM R, S
TOKENIZE
   R ON name AS R_token,
   S ON name AS S_token
WHERE R.city = 'SLC'
   AND S.county = 'Salt Lake'
   AND R_token = S_token
GROUP BY R.ssn, S.ssn
HAVING COUNT(*) \geq k;
SELECT R.ssn, S.ssn
FROM R, S
TOKENIZE
  R ON name AS R_token,
  S ON name AS S_token
WHERE R.city = 'SLC'
  AND S.county='Salt Lake'
  AND R.token = S.token
GROUP BY R.ssn, S.ssn
HAVING COUNT(*) >= k;

---

- efficient logical plans with transformation rules
Query Plans with Tokenize

SELECT R.ssn, S.ssn
FROM R, S
TOKENIZE
  R ON name AS R_token,
  S ON name AS S_token
WHERE R.city = 'SLC'
  AND S.county = 'Salt Lake'
  AND R_token = S_token
GROUP BY R.ssn, S.ssn
HAVING COUNT(*) >= k;

Customer tables \( R, S \):
- \(|name| = 9\) chars on avg.
- \(|\alpha(name)| = 10\) on avg.

- efficient logical plans with transformation rules

\[ \pi_{R.ssn, S.ssn} \]
\[ \sigma_{\text{olap} \geq k} \]
\[ \gamma_{R.ssn, S.ssn; \text{COUNT(*)} \rightarrow \text{olap}} \]
\[ \times_{R\_token = S\_token} \]
\[ \tau_{\alpha(name) \rightarrow R\_token} \]
\[ \tau_{\alpha(name) \rightarrow S\_token} \]
\[ \sigma_{city = 'SLC'} [1\%] \]
\[ \sigma_{county = 'Salt Lake'} [5\%] \]

\( R(ssn, name, city) \)
\( S(ssn, name, county) \)

\( R \) and \( S \) are customer tables.

\( \times \) denotes true cardinality.
\( \sigma \) denotes estimated cardinality.
\( \pi \) denotes projection.
\( \gamma \) denotes count projection.
\( \tau \) denotes selection.

Nikolaus Augsten (Salzburg, Austria)
The Tokenize Operator

Query Optimization

Query Plans with Tokenize

```
SELECT R.ssn, S.ssn
FROM R, S
TOKENIZE
  R ON name AS R_token,
  S ON name AS S_token
WHERE R.city = 'SLC'
  AND S.county = 'Salt Lake'
  AND R_token = S_token
GROUP BY R.ssn, S.ssn
HAVING COUNT(*) >= k;
```

Customer tables $R$, $S$:
- $|\text{name}| = 9$ chars on avg.
- $|\alpha(\text{name})| = 10$ on avg.

- **efficient logical plans with transformation rules**
- **accurate cardinality estimates**
1 Motivation

2 The Tokenize Operator
   - Efficient Implementation
   - Query Optimization

3 Experiments
Generating Tokens

- generate tokens: stand-alone client, substring function (VLDB’01), table function, tokenize operator
- increase number of tuples / string length
- measure runtime
Experiments

Generating Tokens

- $q$-grams, $q = 3$, $|s| = 10$
- $q$-grams, $q = 5$, 1M tuples

- generate tokens: stand-alone client, substring function (VLDB’01), table function, tokenize operator
- increase number of tuples / string length
- measure runtime

Tokenize scales with tuple size and string length.
Experiments

Tokenize vs. Table Function

- join customer tables on similar names
- select by city (1k customers)
- increase number of tuples
- runtime for tokenize vs. table function

$q$-grams, $q = 2$, Jaccard threshold 0.9
Experiments

Tokenize vs. Table Function

- join customer tables on similar names
- select by city (1k customers)
- increase number of tuples
- runtime for tokenize vs. table function

$q$-grams, $q = 2$, Jaccard threshold 0.9

Tokenize generates more efficient query plans.
Virtual Tuple Replication

Experiments

- **generate tokens for 1M tuples**
- **compare VTR vs. physical replication**
- **increase tuple size (number of 50 char columns)**
- **measure runtime and size on disk**

$q$-grams, $q = 5$, 1M tuples

Nikolaus Augsten (Salzburg, Austria)  On-the-Fly Token Sim. Join in RDBs  SIGMOD 14 – Snowbird, UT
Experiments

Virtual Tuple Replication

- generate tokens for 1M tuples
- compare VTR vs. physical replication
- increase tuple size (number of 50 char columns)
- measure runtime and size on disk

VTR is fast and reduces size of intermediate results.
Conclusion

- **Tokenize** is a logical operator that computes tokens
- **VTR** avoids replicating tuples physically
- **Efficient query plans** with tokenize:
  - flexible transformation rules
  - accurate cardinality estimates

Not shown here:
- computing prefixes for filtering
- efficient verification with tokenize

Tokenize enables efficient on-the-fly token similarity joins.
**Conclusion**

- **Tokenizer** is a logical operator that computes tokens
- **VTR** avoids replicating tuples physically
- **Efficient query plans** with tokenize:
  - flexible transformation rules
  - accurate cardinality estimates
- **Not shown here:**
  - computing prefixes for filtering
  - efficient verification with tokenize
Tokenize is a logical operator that computes tokens

VTR avoids replicating tuples physically

Efficient query plans with tokenize:
- flexible transformation rules
- accurate cardinality estimates

Not shown here:
- computing prefixes for filtering
- efficient verification with tokenize

Tokenize enables efficient on-the-fly token similarity joins.