On-the-Fly Token Similarity Joins in Relational Databases

N. Augsten[1] A. Miraglia[2] T. Neumann[3] A. Kemper[3]

[1]University of Salzburg, Austria
nikolaus.augsten@sbg.ac.at

[2]VU University Amsterdam, Netherlands
a.miraglia@student.vu.nl

[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

Outline

1 Motivation

2 The Tokenize Operator
- Efficient Implementation
- Query Optimization

3 Experiments

Token Similarity Join

R

A
snowbird
canyons
...

$$\bowtie_{\mathrm{sim}(A,B)\geq 70\%}$$

$$\mathrm{sim}(A,B) = \frac{|\alpha(A) \cap \alpha(B)|}{|\alpha(A) \cup \alpha(B)|}$$

S

B
snowbasin
snowbirds
...

Token Similarity Join

R

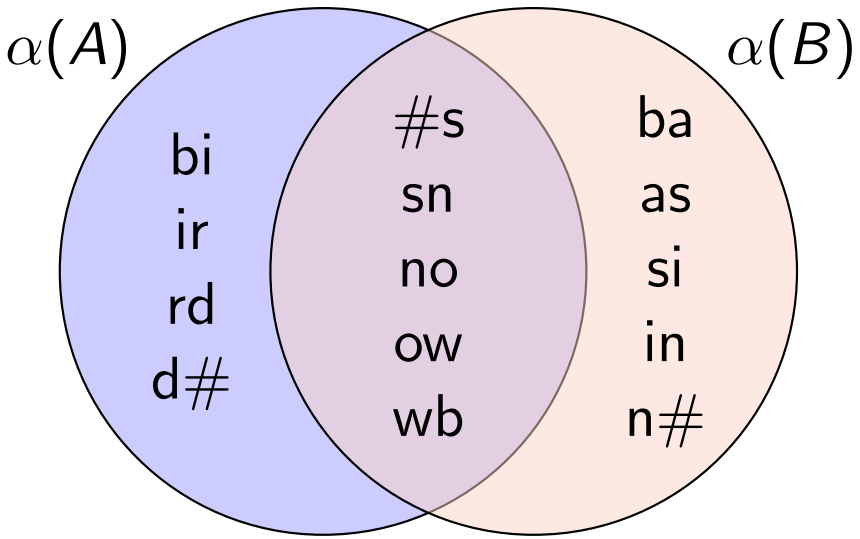
A
snowbird
canyons
...

$\bowtie_{\text{sim}(A,B)\geq 70\%}$

S

B
snowbasin
snowbirds
...

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



$$\text{sim}(A, B) = \frac{5}{14} = 36\%$$ ✗

Token Similarity Join

R

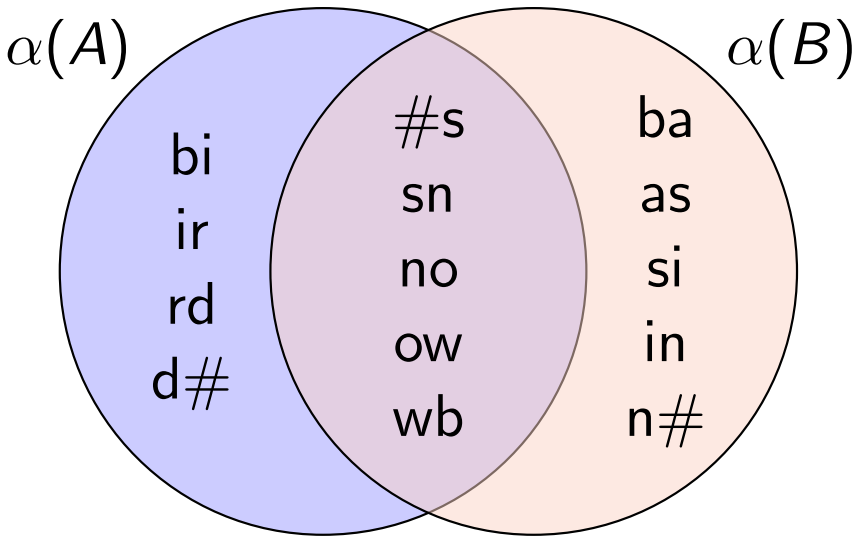
A
snowbird
canyons
...

$\bowtie_{\text{sim}(A,B)\geq 70\%}$

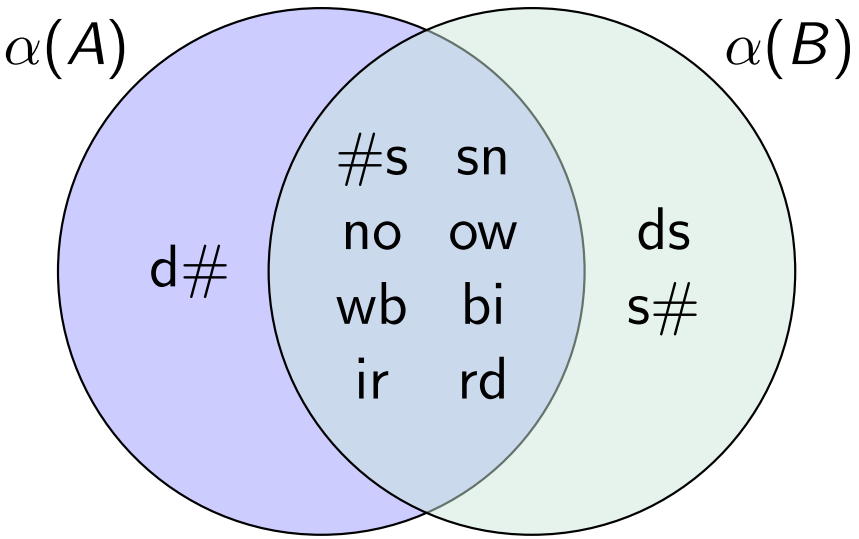
S

B
snowbasin
snowbirds
...

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

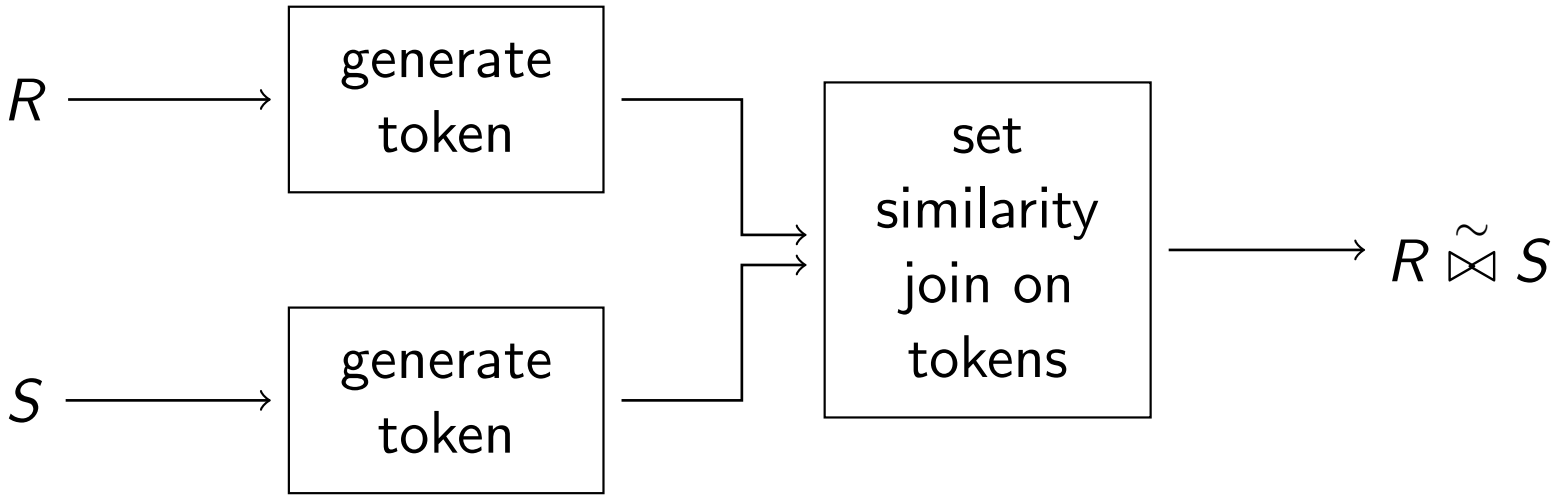


$$\text{sim}(A, B) = \frac{5}{14} = 36\%$$ ✗

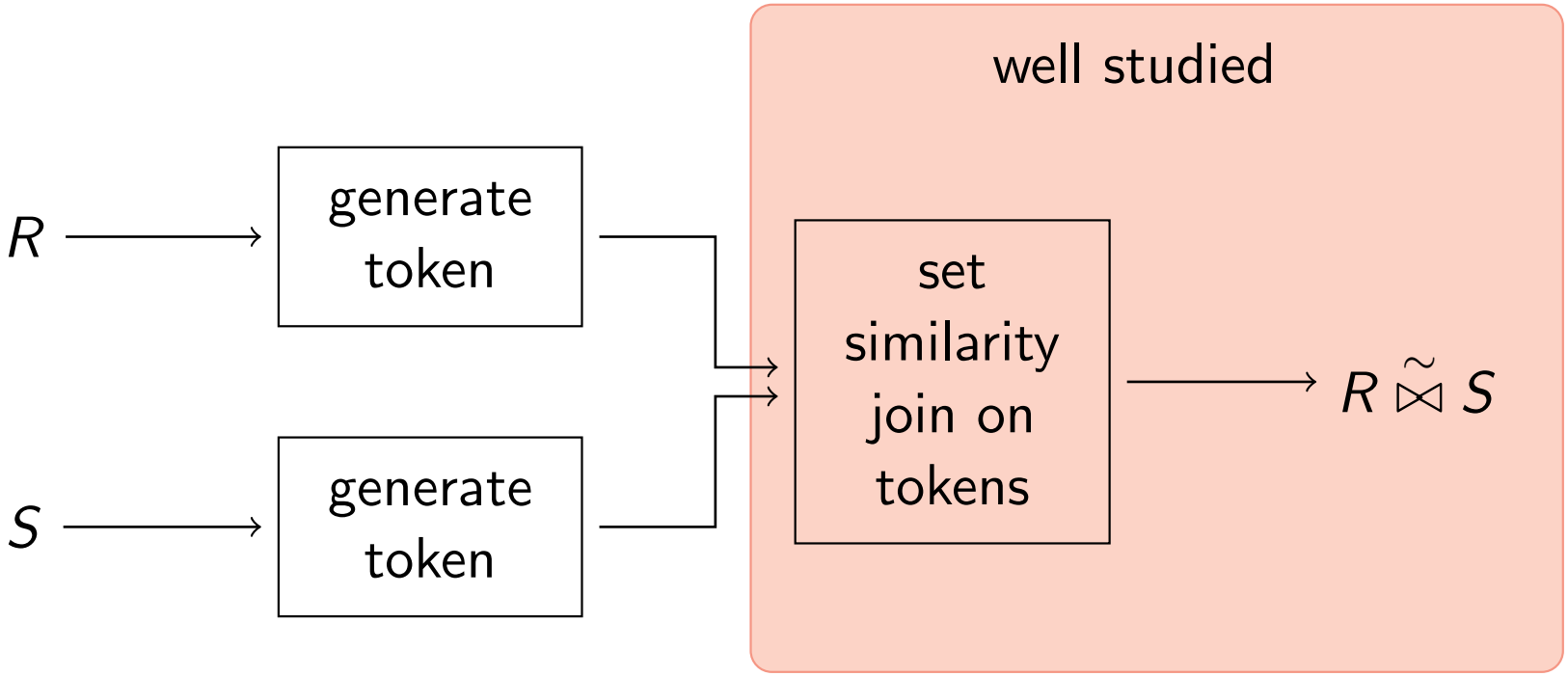


$$\text{sim}(A, B) = \frac{8}{11} = 73\%$$ ✓

Token Generation in Similarity Joins

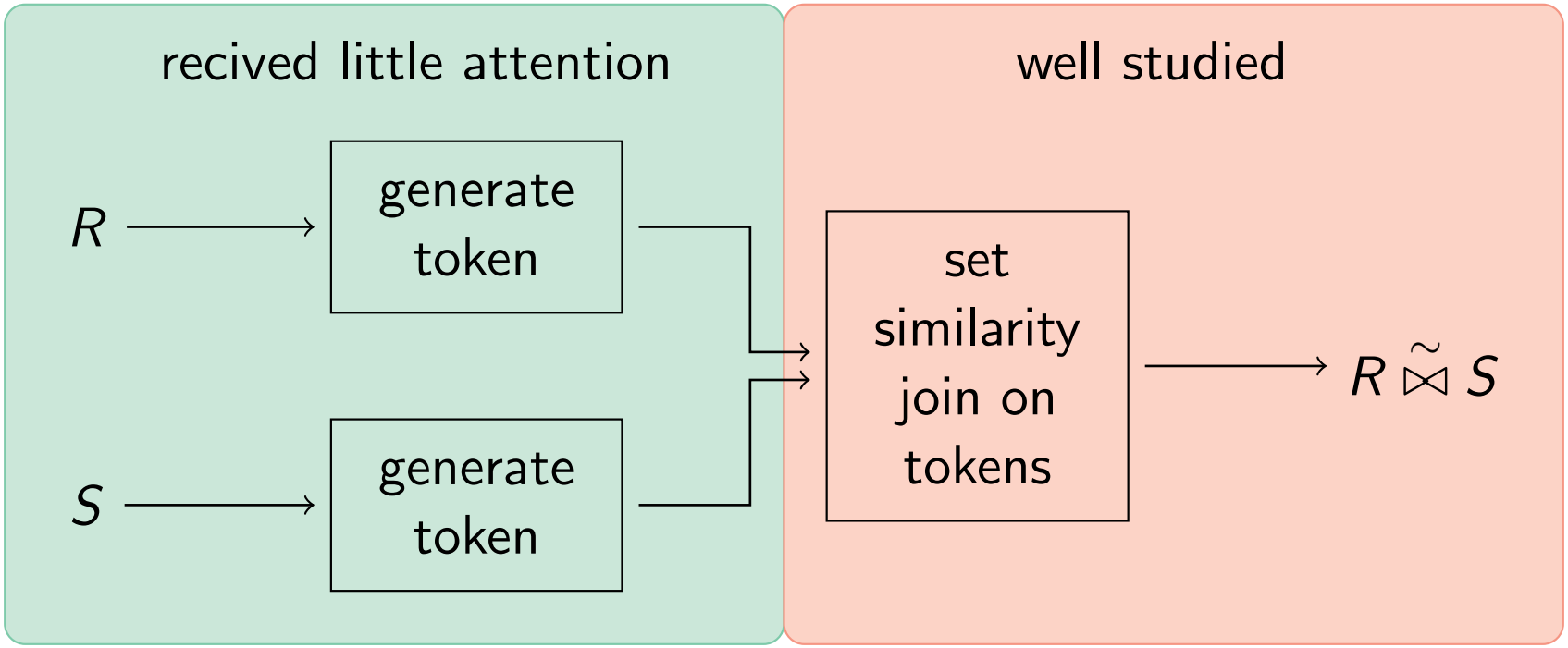


Token Generation in Similarity Joins



- Part-Enum (VLDB'06)
- All-Pairs (WWW'07)
- PP-Join (WWW'08)
- MP-Join (Inf. Syst.'11)
- Adapt-Join (SIGMOD'12)
- ...

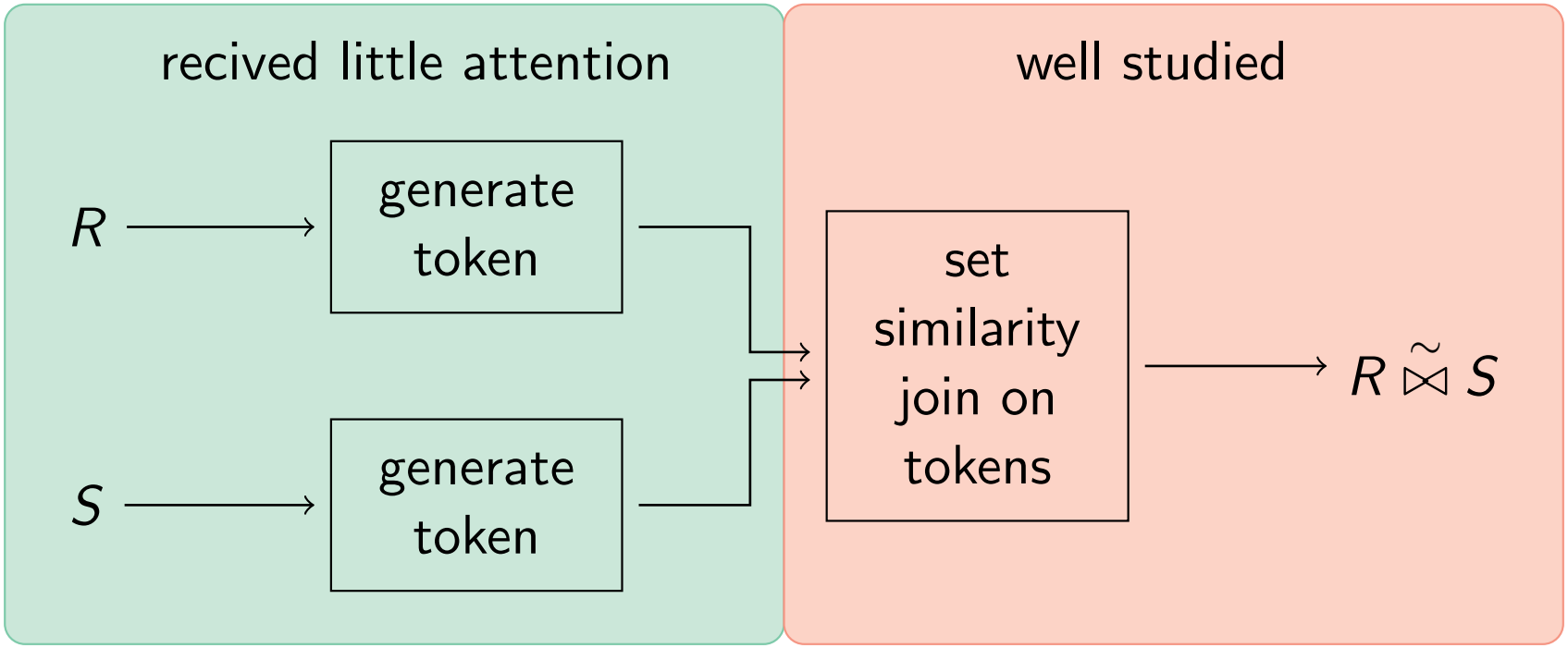
Token Generation in Similarity Joins



- 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



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

State-of-the-Art: Table Function

Customer tables R, S:

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

State-of-the-Art: Table Function

Customer tables R, S:

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

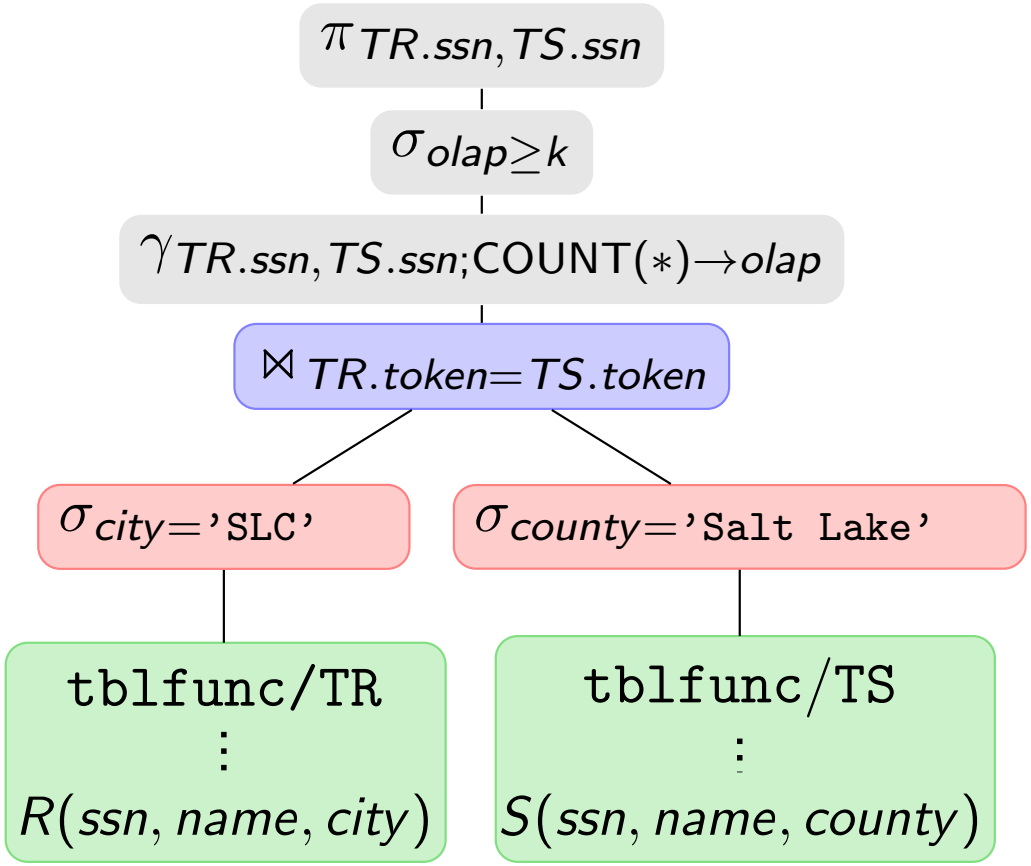
```
SELECT   TR.ssn, TS.ssn
  FROM   tblfunc('R', 'name') TR,
         tblfunc('S', 'name') TS
 WHERE   TR.city = 'SLC'
   AND   TS.county='Salt Lake'
   AND   TR.token = TS.token
 GROUP   BY TR.ssn, TS.ssn
HAVING   COUNT(*) >= k;
```

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;
```

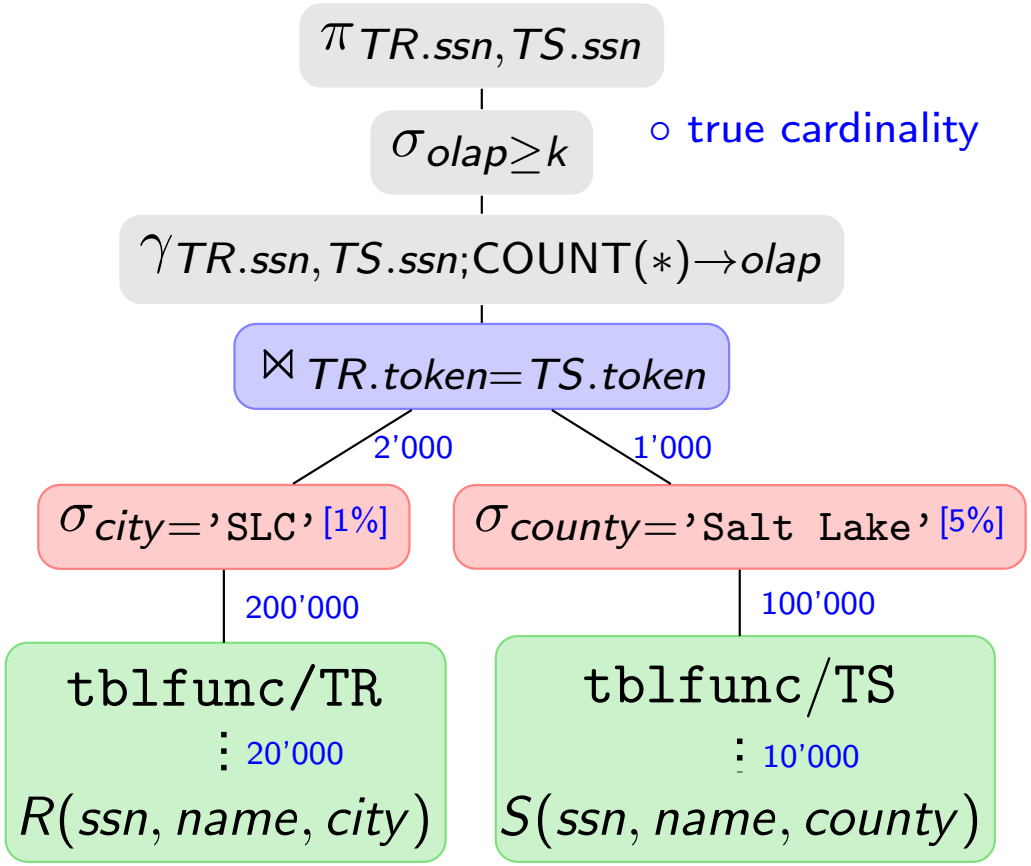


State-of-the-Art: Table Function

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;
```



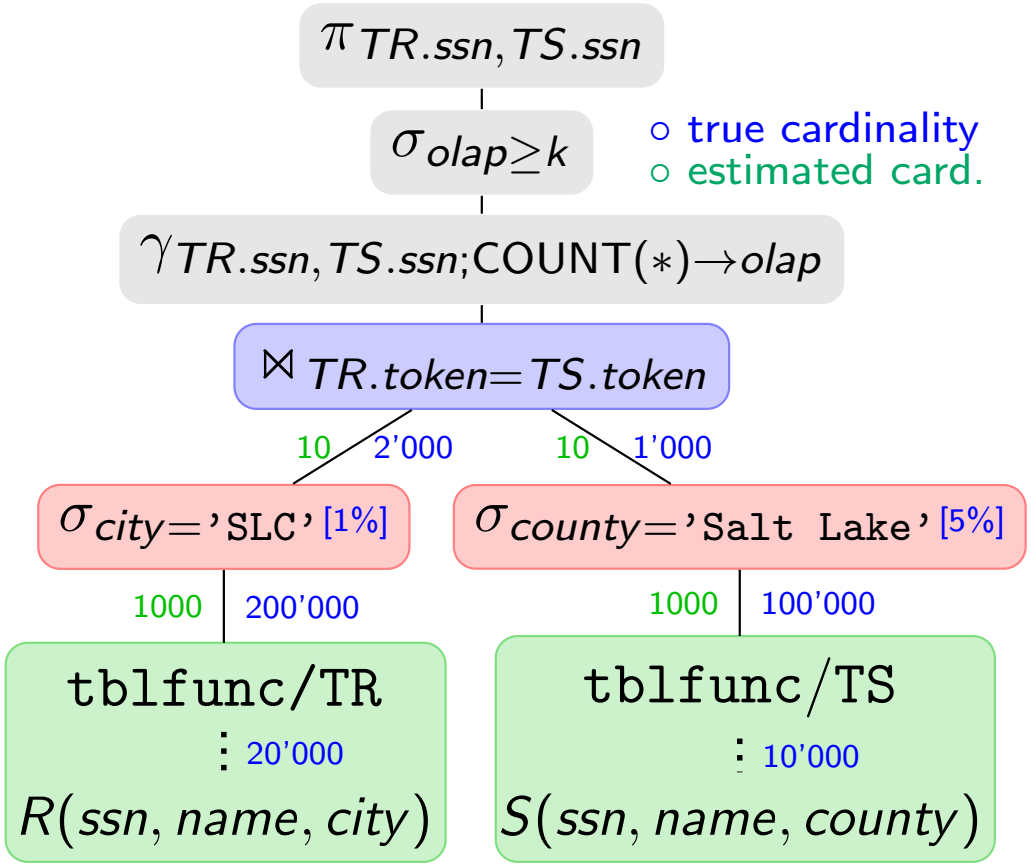
- black box: selection and projection not pushed down
  - tokens computed for *all* customers
  - too many attributes replicated (name, city, county)

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;
```



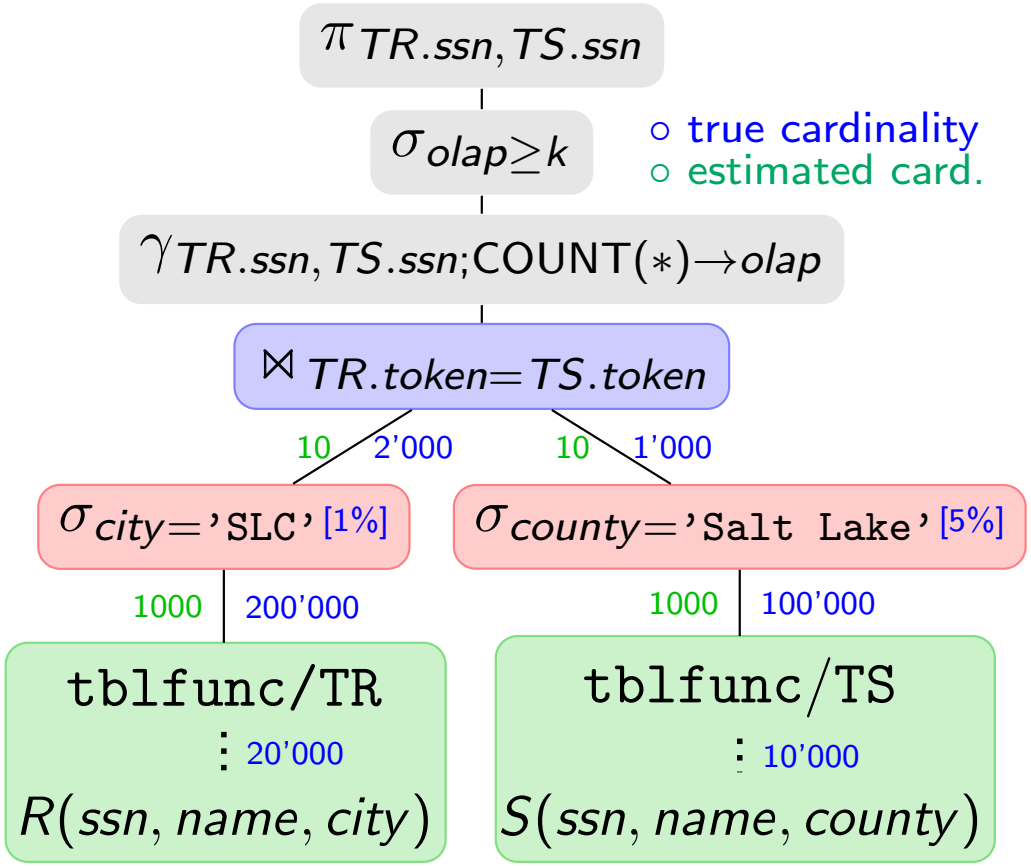
- black box: selection and projection not pushed down
  - ➥ tokens computed for *all* customers
  - ➥ too many attributes replicated (name, city, county)
- unknown cardinality of table function (often assumed a constant)

State-of-the-Art: Table Function

Customer tables R, S:

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

```
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;
```



Problem: poor query plans with table functions.

Outline

1 Motivation

2 The Tokenize Operator
- Efficient Implementation
- Query Optimization

3 Experiments

Solution: Tokenize Operator

Tokenize τ is a relational operator defined as follows:

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

Solution: Tokenize Operator

Tokenize τ 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.

R

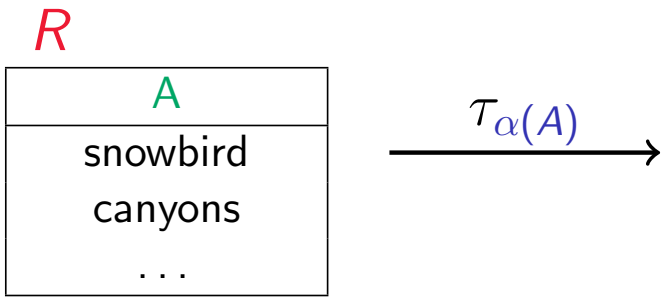
A
snowbird
canyons
...

Solution: Tokenize Operator

Tokenize τ 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.
- α is token function
  - ➥ computes tokens for a single value (e.g., q-grams for a string)



Solution: Tokenize Operator

Tokenize τ 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.
- α is token function
  - ➥ computes tokens for a single value (e.g., q-grams for a string)
- Output: relation with tokens

R

A
snowbird
canyons
...

$\xrightarrow{\tau_{\alpha(A)}}$

$\tau_{\alpha(A)\rightarrow token}(R)$

A	token
snowbird	#s
snowbird	sn
snowbird	no
...	...
canyons	#c
canyons	ca
canyons	an
...	...

Solution: Tokenize Operator

Tokenize τ 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.
- α is token function
  - ➥ computes tokens for a single value (e.g., q-grams for a string)
- Output: relation with tokens

R

A
snowbird
canyons
...

$\xrightarrow{\tau_{\alpha(A)}}$

$\tau_{\alpha(A)\rightarrow token}(R)$

A	token
snowbird	#s
snowbird	sn
snowbird	no
...	...
canyons	#c
canyons	ca
canyons	an
...	...

- Note:
  - ➥ α is type specific (like MAX, AVG)
  - ➥ tokenize τ abstracts from types (like GROUP)

Outline

1 Motivation

2 The Tokenize Operator
- Efficient Implementation
- Query Optimization

3 Experiments

Virtual Tuple Replication/1

R

name	role	conference
curtis dyreson	general chair	sigmod
feifei li	general chair	sigmod
tamer ozsu	program chair	sigmod
...	...	...

Virtual Tuple Replication/1

R

name	role	conference
curtis dyreson	general chair	sigmod
feifei li	general chair	sigmod
tamer ozsu	program chair	sigmod
...	...	...

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

name	role	conference	token
curtis dyreson	general chair	sigmod	#c
curtis dyreson	general chair	sigmod	cu
curtis dyreson	general chair	sigmod	ur
...	...	...	...
feifei li	general chair	sigmod	#f
feifei li	general chair	sigmod	fe
...	...	...	...
tamer ozsu	program chair	sigmod	#t
tamer ozsu	program chair	sigmod	ta
...	...	...	...

Virtual Tuple Replication/1

R

name	role	conference
curtis dyreson	general chair	sigmod
feifei li	general chair	sigmod
tamer ozsu	program chair	sigmod
...	...	...

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

name	role	conference	token
curtis dyreson	general chair	sigmod	#c
curtis dyreson	general chair	sigmod	cu
curtis dyreson	general chair	sigmod	ur
...	...	...	...
feifei li	general chair	sigmod	#f
feifei li	general chair	sigmod	fe
...	...	...	...
tamer ozsu	program chair	sigmod	#t
tamer ozsu	program chair	sigmod	ta
...	...	...	...

expensive!

- physical replication is expensive

Virtual Tuple Replication/1

R

name	role	conference
curtis dyreson	general chair	sigmod
feifei li	general chair	sigmod
tamer ozsu	program chair	sigmod
...	...	...

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

name	role	conference	token
curtis dyreson	general chair	sigmod	#c
curtis dyreson	general chair	sigmod	cu
curtis dyreson	general chair	sigmod	ur
...	...	...	...
feifei li	general chair	sigmod	#f
feifei li	general chair	sigmod	fe
...	...	...	...
tamer ozsu	program chair	sigmod	#t
tamer ozsu	program chair	sigmod	ta
...	...	...	...

- physical replication is expensive
- VTR avoids physical replication

Virtual Tuple Replication/1

R

name	role	conference
curtis dyreson	general chair	sigmod
feifei li	general chair	sigmod
tamer ozsu	program chair	sigmod
...	...	...

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

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

name	role	conference	token	GID
curtis dyreson	general chair	sigmod	#c	1
curtis dyreson	general chair	sigmod	cu	1
curtis dyreson	general chair	sigmod	ur	1
...	...	...	...	...
feifei li	general chair	sigmod	#f	2
feifei li	general chair	sigmod	fe	2
...	...	...	...	...
tamer ozsu	program chair	sigmod	#t	3
tamer ozsu	program chair	sigmod	ta	3
...	...	...	...	...

Virtual Tuple Replication/1

R

name	role	conference
curtis dyreson	general chair	sigmod
feifei li	general chair	sigmod
tamer ozsu	program chair	sigmod
...	...	...

- physical replication is expensive
- VTR avoids physical replication
  1. mark group with GID
  2. create VTR bit array

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

name	role	conference	token	GID
curtis dyreson	general chair	sigmod	#c	1
curtis dyreson	general chair	sigmod	cu	1
curtis dyreson	general chair	sigmod	ur	1
...	...	...	...	...
feifei li	general chair	sigmod	#f	2
feifei li	general chair	sigmod	fe	2
...	...	...	...	...
tamer ozsu	program chair	sigmod	#t	3
tamer ozsu	program chair	sigmod	ta	3
...	...	...	...	...

VTR Bit Array

1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
...	...	...	...	...
1	1	1	1	1
1	1	1	1	1
...	...	...	...	...
1	1	1	1	1
1	1	1	1	1
...	...	...	...	...

Virtual Tuple Replication/1

R

name	role	conference
curtis dyreson	general chair	sigmod
feifei li	general chair	sigmod
tamer ozsu	program chair	sigmod
...	...	...

- 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

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

name	role	conference	token	GID
curtis dyreson	general chair	sigmod	#c	1
$\perp$	$\perp$	$\perp$	cu	1
$\perp$	$\perp$	$\perp$	ur	1
...	...	...	...	...
feifei li	general chair	sigmod	#f	2
$\perp$	$\perp$	$\perp$	fe	2
...	...	...	...	...
tamer ozsu	program chair	sigmod	#t	3
$\perp$	$\perp$	$\perp$	ta	3
...	...	...	...	...

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
...	...	...	...	...

Virtual Tuple Replication/1

R

name	role	conference
curtis dyreson	general chair	sigmod
feifei li	general chair	sigmod
tamer ozsu	program chair	sigmod
...	...	...

- 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

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

name	role	conference	token	GID
curtis dyreson	general chair	sigmod	#c	1
$\perp$	$\perp$	$\perp$	cu	1
$\perp$	$\perp$	$\perp$	ur	1
...	...	...	...	...
feifei li	general chair	sigmod	#f	2
$\perp$	$\perp$	$\perp$	fe	2
...	...	...	...	...
tamer ozsu	program chair	sigmod	#t	3
$\perp$	$\perp$	$\perp$	ta	3
...	...	...	...	...

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
...	...	...	...	...

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

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:

$$\gamma \qquad \text{COUNT}(*)\rightarrow\text{cnt}(\tau_{\alpha(name)\rightarrow tk} \qquad (R))$$

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:

$$\gamma_{GID;\quad\text{COUNT}(*)\rightarrow\text{cnt}}(\tau_{\alpha(name)\rightarrow tk;GID}(R))$$

  - ➥ generate and group by *GID*

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:

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

  - ➥ generate and group by *GID*
  - ➥ restore *name* attribute

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:

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

  - ➥ generate and group by *GID*
  - ➥ restore *name* attribute
  - ➥ remove *GID*

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:

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

  - ➥ 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

Logical Query Plan

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: $A \subseteq B$

$$\pi_{BA'}(\tau_{\alpha(A)\to A'}(R)) = \tau_{\alpha(A)\to A'}(\pi_B(R))$$

- 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:

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

- Most token functions α produce linear number of tokens

  $\Rightarrow$ accurate cardinality estimates

Cardinality Estimation

- Cardinality estimation for tokenize:

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

- Most token functions α produce linear number of tokens

  $\Rightarrow$ accurate cardinality estimates

Token	Type	Statistics	Cardinality
q-grams	string	avg. string length $\bar{s}$	$\lvert R\rvert(\bar{s}+q-1)$
binar branchs	tree	avg. node number $\bar{t}$	$\lvert R\rvert\bar{t}$
pq-grams	tree	avg. node number $\bar{t}$	$\lvert R\rvert q\bar{t}$

Cardinality Estimation

- Cardinality estimation for tokenize:

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

- Most token functions α produce linear number of tokens

  $\Rightarrow$ accurate cardinality estimates

Token	Type	Statistics	Cardinality
q-grams	string	avg. string length $\bar{s}$	$\lvert R\rvert(\bar{s}+q-1)$
binar branchs	tree	avg. node number $\bar{t}$	$\lvert R\rvert\bar{t}$
pq-grams	tree	avg. node number $\bar{t}$	$\lvert R\rvert q\bar{t}$

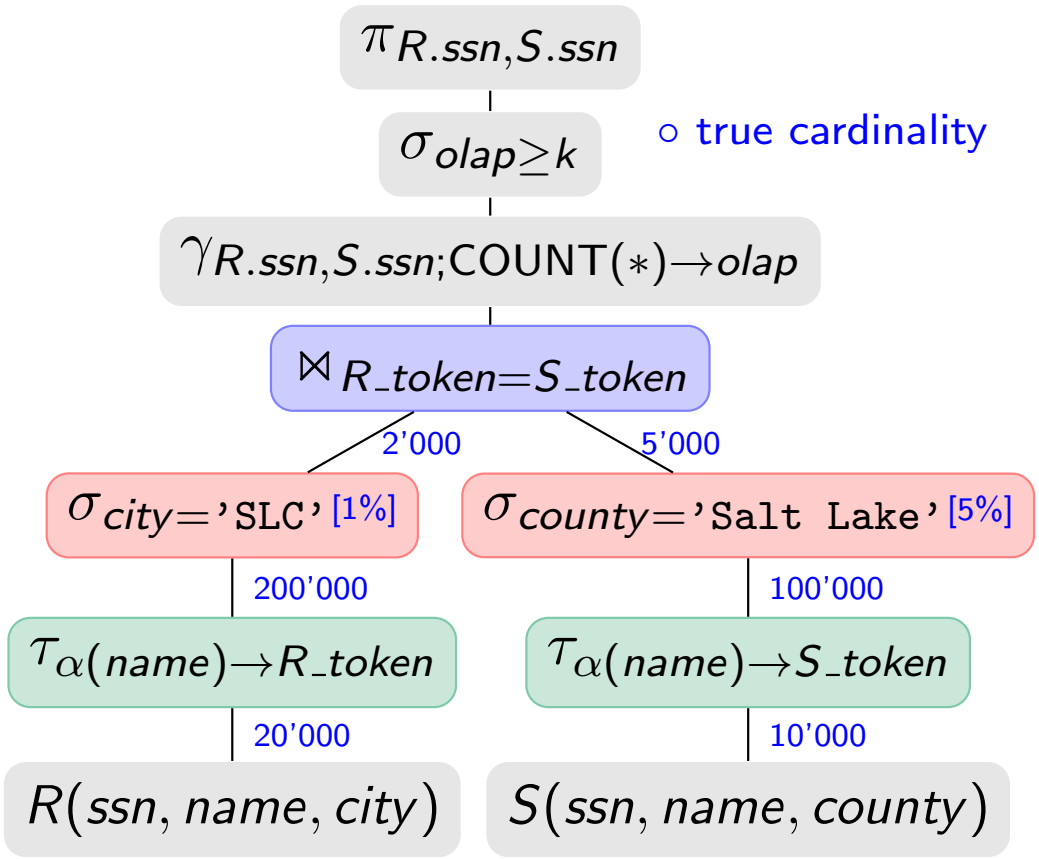
Simple and accurate cardinality estimates for tokenize.

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;
```

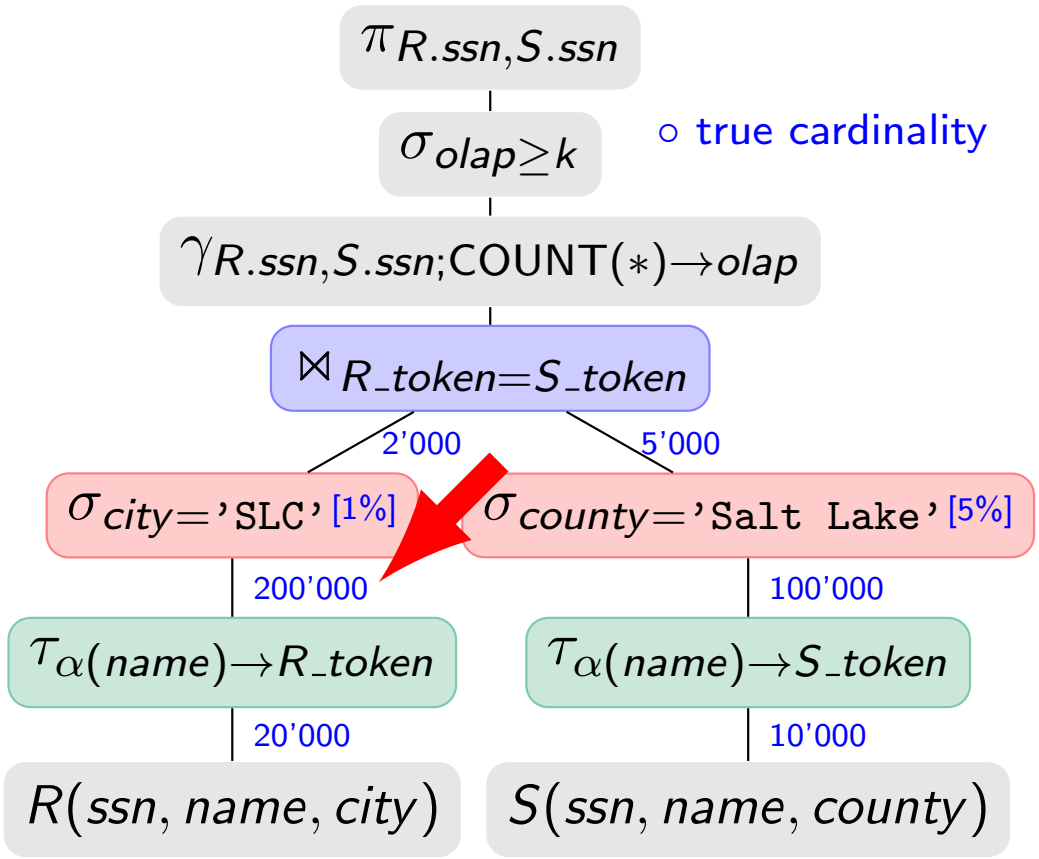
Query Plans with Tokenize

```
    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;
```



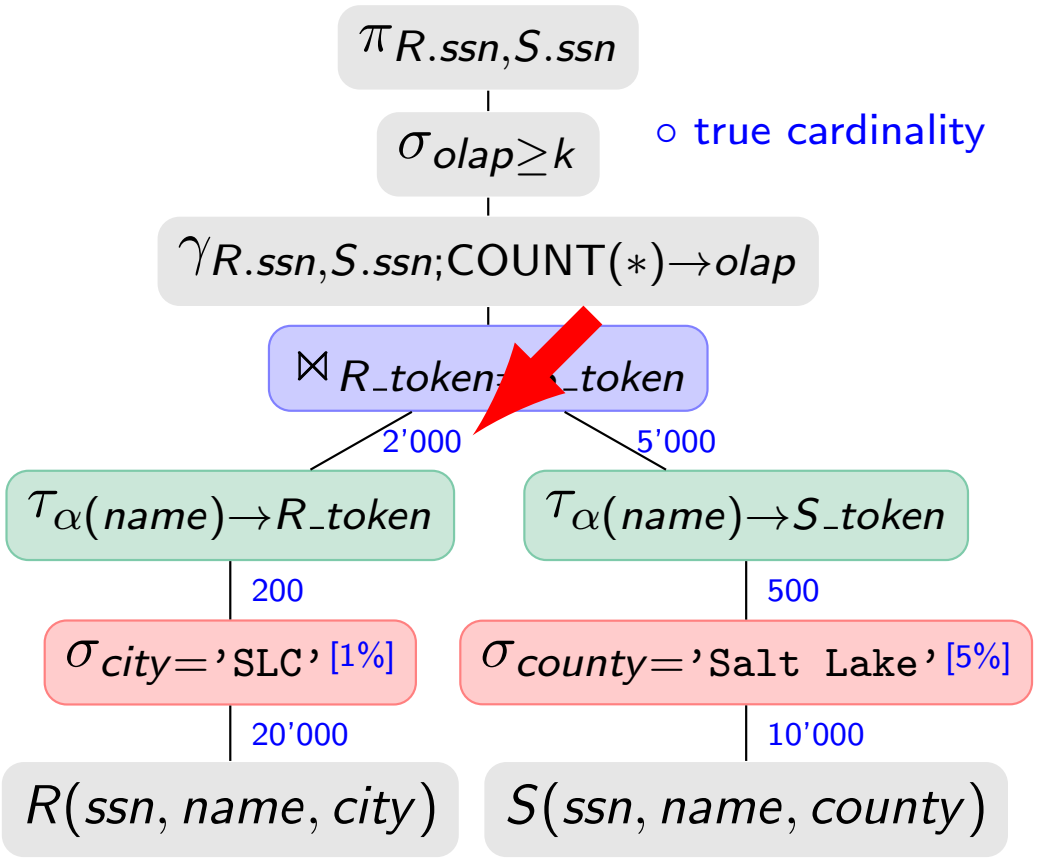
Query Plans with Tokenize

```
    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;
```



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;
```



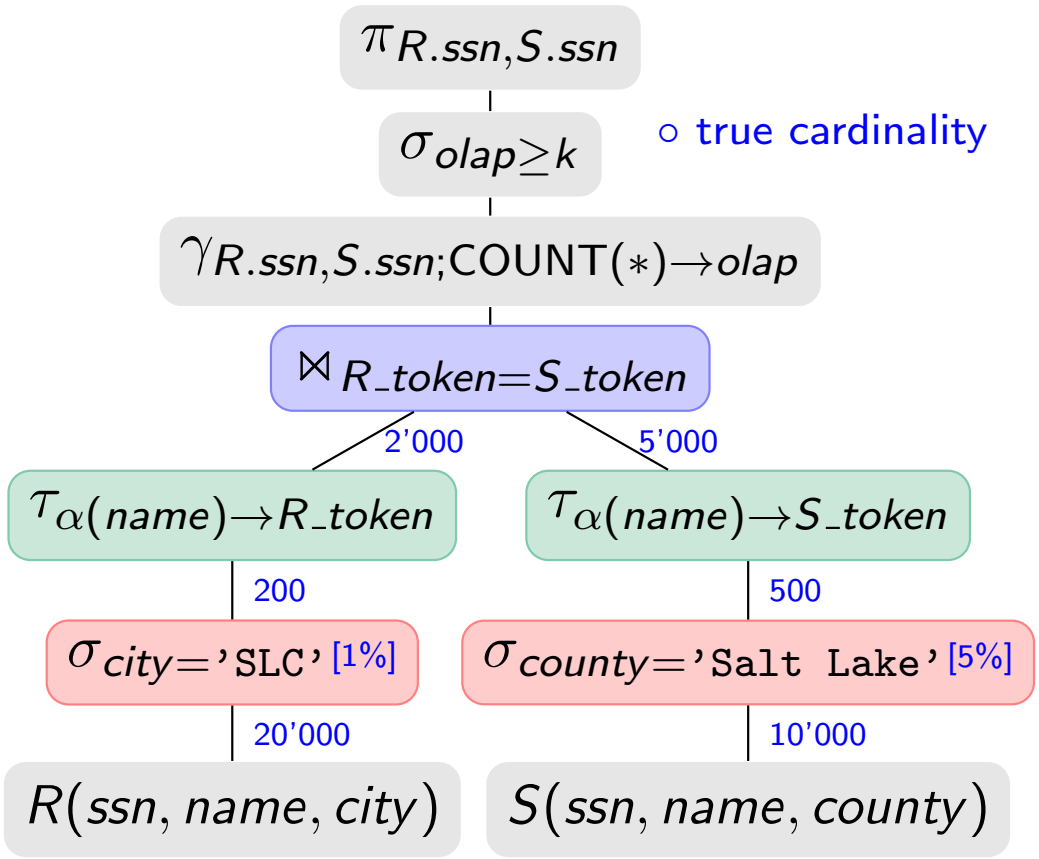
- efficient logical plans with transformation rules

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:

- $|\text{name}| = 9$ chars on avg.
- $|\alpha(\text{name})| = 10$ on avg.



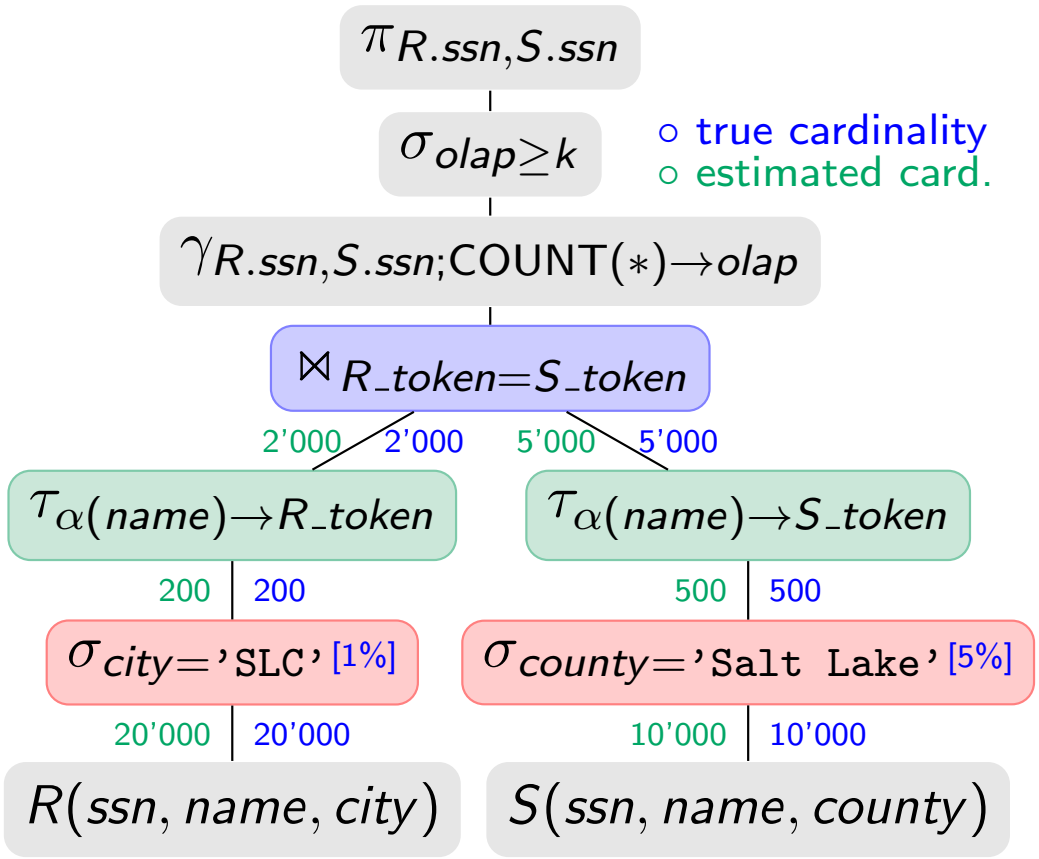
- efficient logical plans with transformation rules

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:

- $|\text{name}| = 9$ chars on avg.
- $|\alpha(\text{name})| = 10$ on avg.



- efficient logical plans with transformation rules
- accurate cardinality estimates

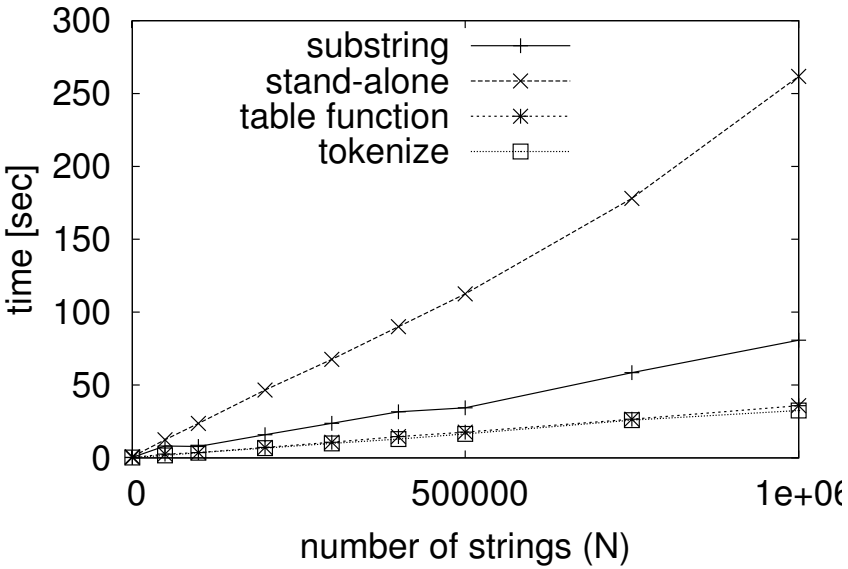
Outline

1 Motivation

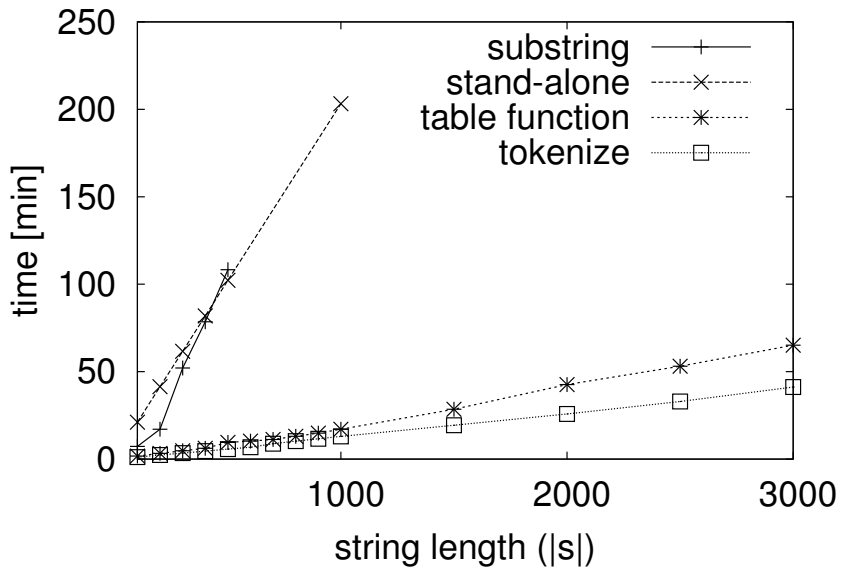
2 The Tokenize Operator
- Efficient Implementation
- Query Optimization

3 Experiments

Generating Tokens



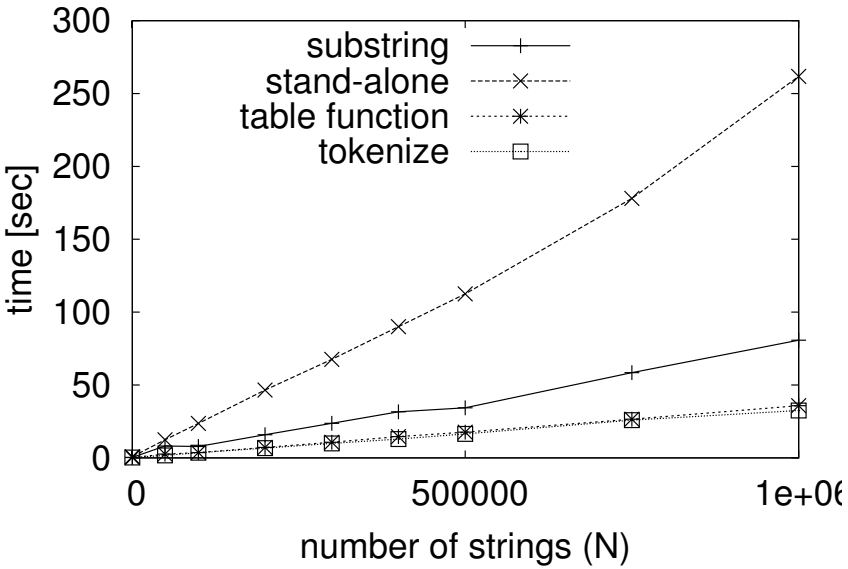
q-grams, $q = 3$, $|s| = 10$



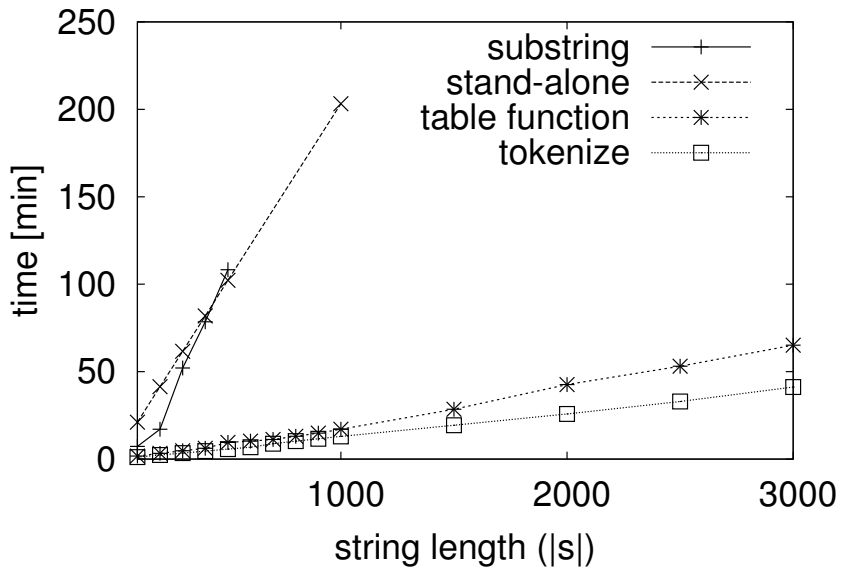
q-grams, $q = 5$, 1M tuples

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

Generating Tokens



q-grams, $q = 3$, $|s| = 10$

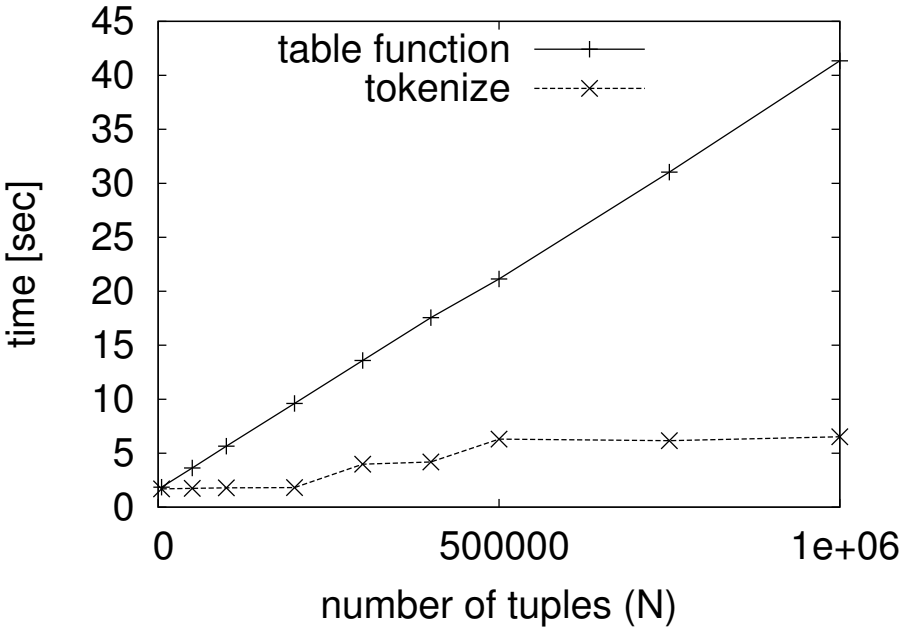


q-grams, $q = 5$, 1M tuples

- 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.

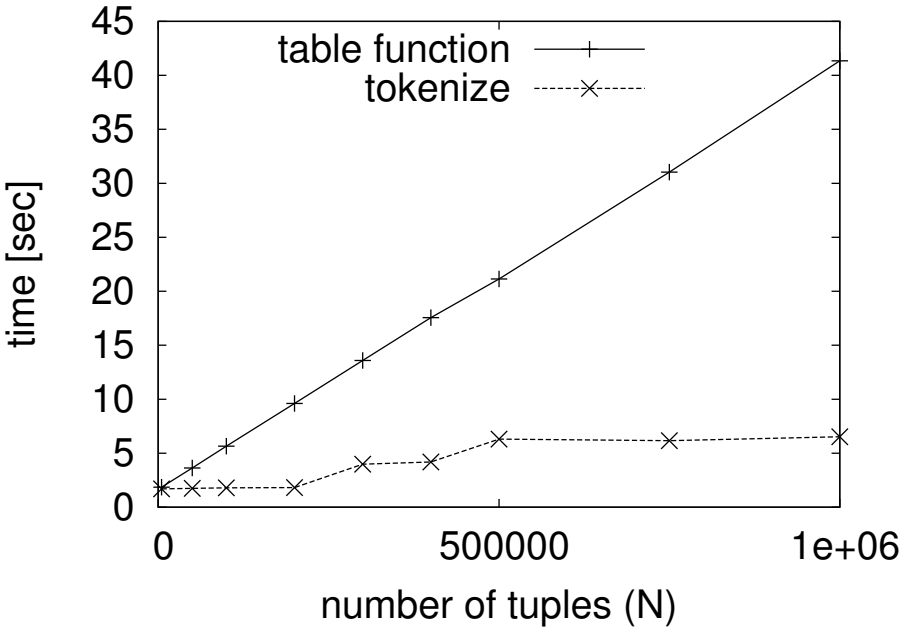
Tokenize vs. Table Function



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

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

Tokenize vs. Table Function

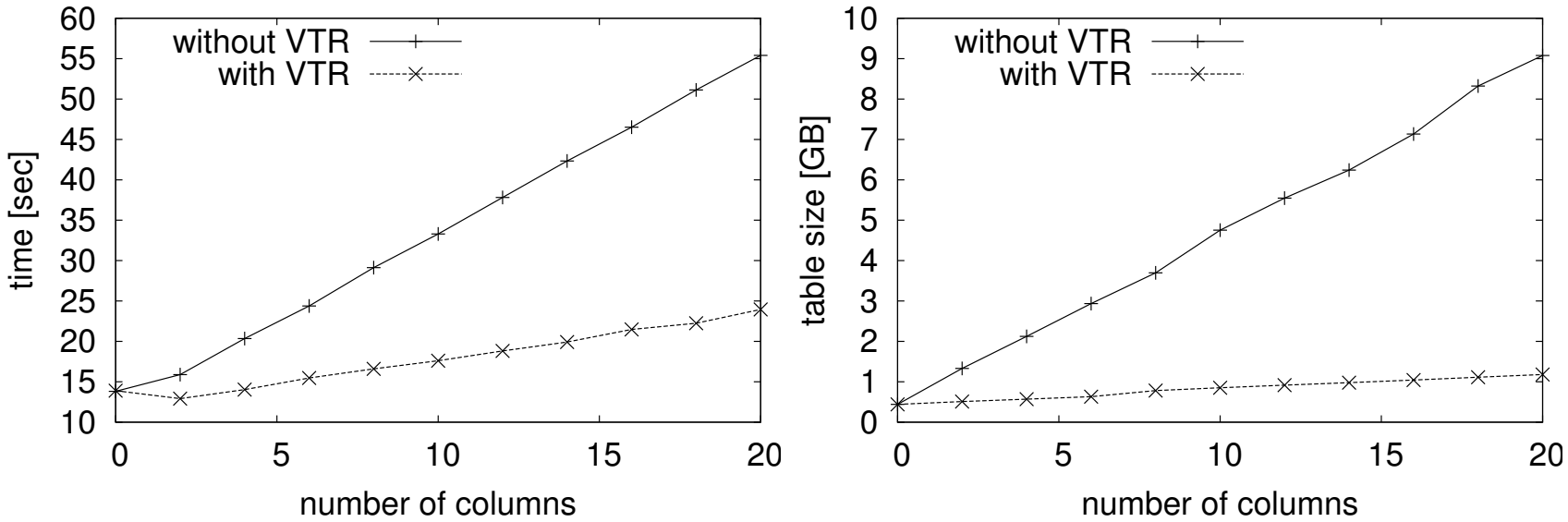


q-grams, $q = 2$, Jaccard threshold 0.9

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

Tokenize generates more efficient query plans.

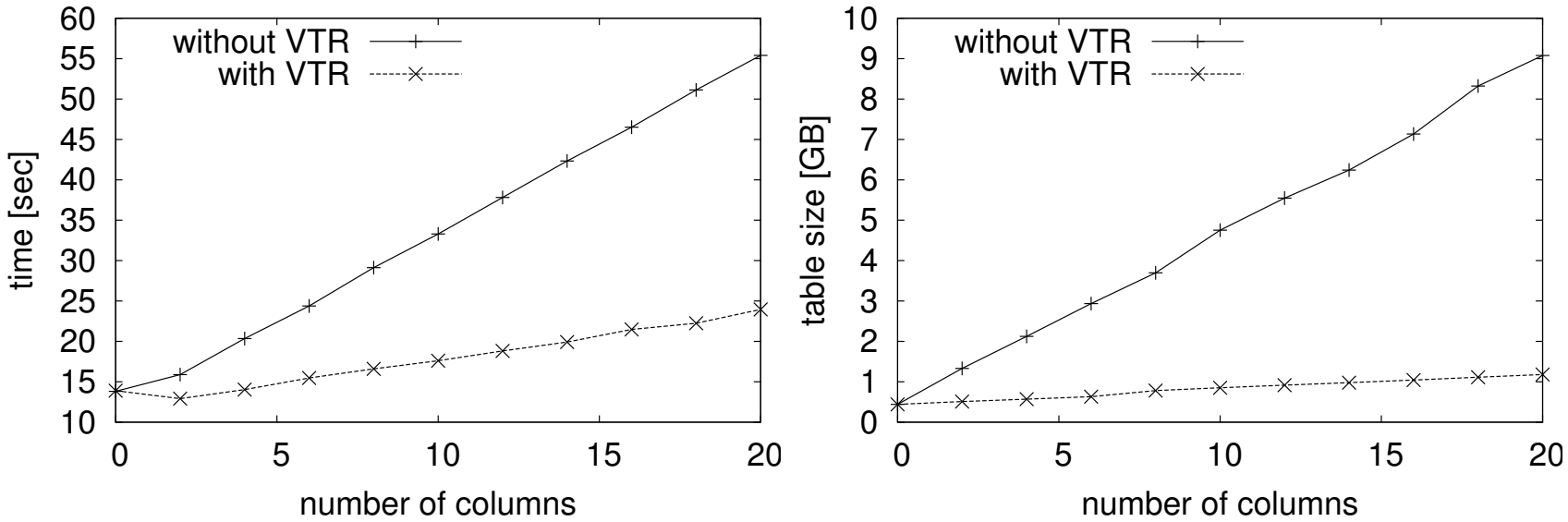
Virtual Tuple Replication



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

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

Virtual Tuple Replication



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

- 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

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

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.