String Problems in Database Theory

Antoine Amarilli

October 26, 2026

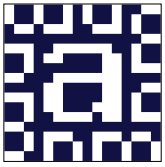
Télécom Paris, Inria Lille

Who am I?



Antoine Amarilli

Associate professor at Télécom Paris
Currently on sabbatical at Inria Lille

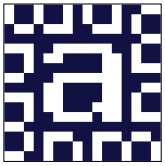


Who am I?



Antoine Amarilli

Associate professor at Télécom Paris
Currently on sabbatical at Inria Lille



- PhD in database theory: MSO on bounded-treewidth data ($\leftrightarrow$ tree automata) and probabilities



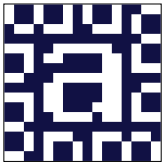


Who am I?



Antoine Amarilli

Associate professor at Télécom Paris
Currently on sabbatical at Inria Lille



- PhD in database theory: MSO on bounded-treewidth data ($\leftrightarrow$ tree automata) and probabilities
- Worked on document spanners ($\leftrightarrow$ word automata) and enumeration



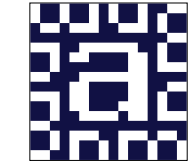


Who am I?

TELECOM Paris

Antoine Amarilli

Associate professor at Télécom Paris
Currently on sabbatical at Inria Lille



- PhD in database theory: MSO on bounded-treewidth data (↔ tree automata) and probabilities
- Worked on document spanners (↔ word automata) and enumeration
- Worked on regular languages (algebra) and incremental maintenance



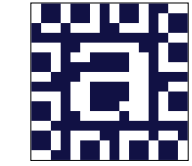


Who am I?

TELECOM Paris

Antoine Amarilli

Associate professor at Télécom Paris
Currently on sabbatical at Inria Lille





- PhD in database theory: MSO on bounded-treewidth data (↔ tree automata) and probabilities
- Worked on document spanners (↔ word automata) and enumeration
- Worked on regular languages (algebra) and incremental maintenance
- Got interested in string algorithms



Disclaimer

This is a database theory talk

Disclaimer

This is a database **theory** talk

Disclaimer

This is a database **theory** talk

Questions in database theory are motivated by practice to some extent
but often lead to theoretical questions that are interesting for their own sake

Disclaimer

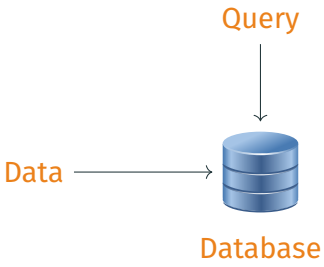
This is a database **theory** talk

Questions in database theory are motivated by practice to some extent
but often lead to theoretical questions that are interesting for their own sake

→ Possible methodological connection with stringology? ;)

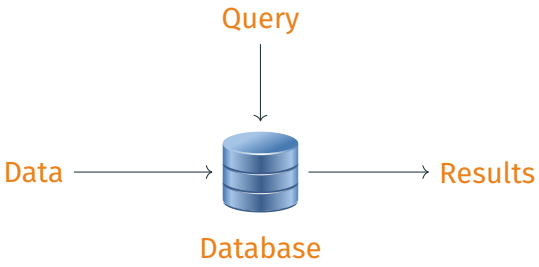
Query evaluation problem

Central (family of) problems in database theory: query evaluation



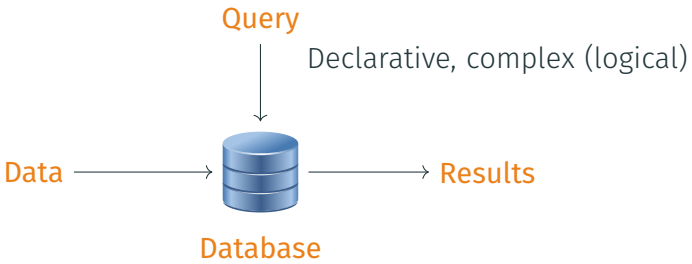
Query evaluation problem

Central (family of) problems in database theory: query evaluation



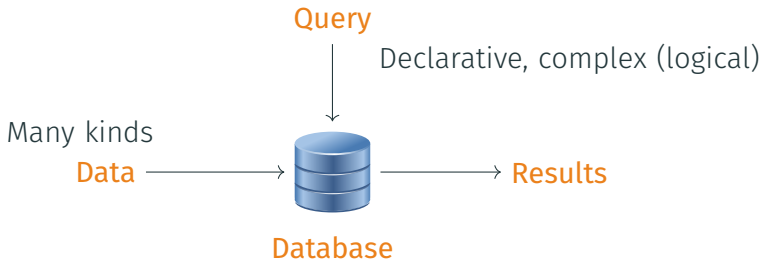
Query evaluation problem

Central (family of) problems in database theory: query evaluation



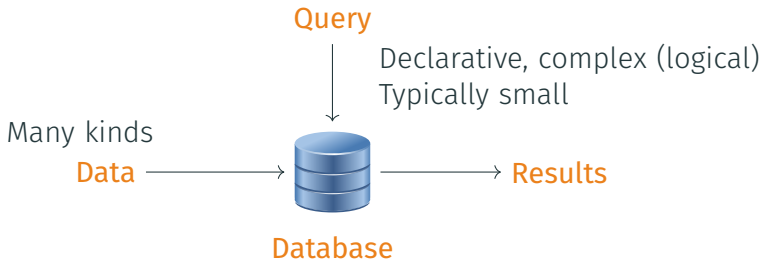
Query evaluation problem

Central (family of) problems in database theory: query evaluation



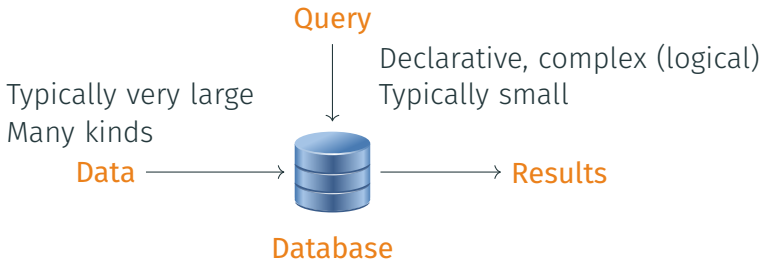
Query evaluation problem

Central (family of) problems in database theory: query evaluation



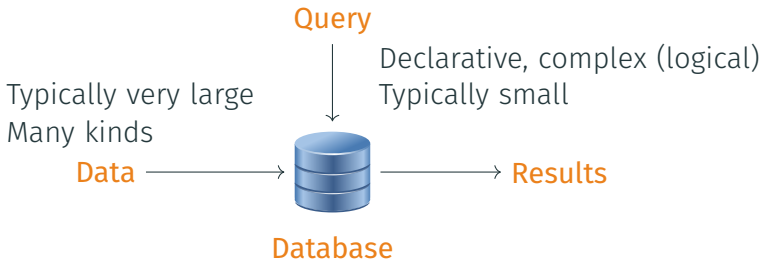
Query evaluation problem

Central (family of) problems in database theory: query evaluation



Query evaluation problem

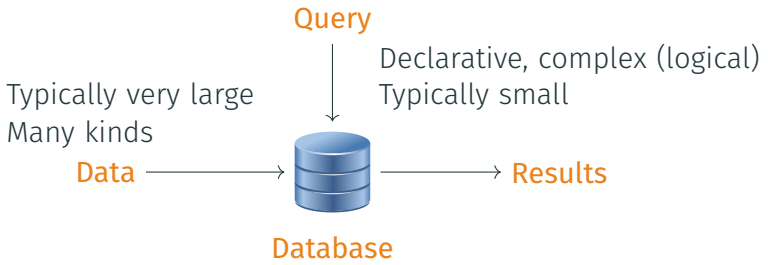
Central (family of) problems in database theory: query evaluation



- Combined complexity:
  - → both the query and data are given as input
- Data complexity for a fixed query:
  - → when only the data is given as input

Query evaluation problem

Central (family of) problems in database theory: query evaluation



- Combined complexity:
  - → both the query and data are given as input
- Data complexity for a fixed query:
  - → when only the data is given as input
- Common goal: classify fixed queries according to their data complexity

Many kinds of data

Relational data

DiseaseAlleles

Disease	Allele
X	ATG
X	GGG
Y	CTG

Sequences

SeqId	Seq
101	GCT
102	GGG
103	GGG

- Established, mature software
- Expressive but very rigid

Many kinds of data

Relational data

DiseaseAlleles

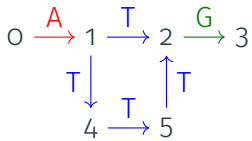
Disease	Allele
X	ATG
X	GGG
Y	CTG

Sequences

SeqId	Seq
101	GCT
102	GGG
103	GGG

- Established, mature software
- Expressive but very rigid

Graph data



- Schemaless, recursive queries
- Will be covered in Wim's talk

Many kinds of data

Relational data

DiseaseAlleles

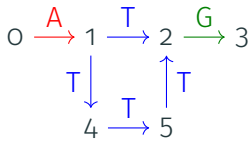
Disease	Allele
X	ATG
X	GGG
Y	CTG

Sequences

SeqId	Seq
101	GCT
102	GGG
103	GGG

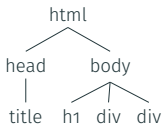
- Established, mature software
- Expressive but very rigid

Graph data



- Schemaless, recursive queries
- Will be covered in Wim's talk

Tree data



- Motivations: XML and JSON

Many kinds of data

Relational data

DiseaseAlleles

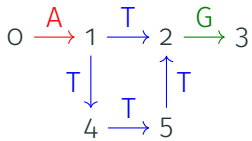
Disease	Allele
X	ATG
X	GGG
Y	CTG

Sequences

SeqId	Seq
101	GCT
102	GGG
103	GGG

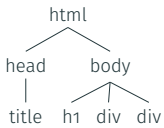
- Established, mature software
- Expressive but very rigid

Graph data



- Schemaless, recursive queries
- Will be covered in Wim's talk

Tree data



- Motivations: XML and JSON

Text data

```
Stringology is the area of computer science which focuses
on understanding the fundamental mathematical properties of
strings (or texts), as well as designing efficient processing
methods for them.  Problems occurring in this area are central
```

- Less expressive, efficient algorithms
- Main focus of the talk

Aside: Stringology questions on relational data

There are many string algorithm problems inside relational database systems!

DiseaseAlleles

Disease	Allele
X	ATG
X	GGG
Y	CTG

Sequences

SeqId	Seq
101	GCT
102	GGG
103	GGG

Aside: Stringology questions on relational data

There are many string algorithm problems inside relational database systems!

DiseaseAlleles

Disease	Allele
X	ATG
X	GGG
Y	CTG

Sequences

SeqId	Seq
101	GCT
102	GGG
103	GGG

- String operators: `SELECT * FROM Sequences WHERE Seq LIKE '%ACG%';`

Aside: Stringology questions on relational data

There are many string algorithm problems inside relational database systems!

DiseaseAlleles

Disease	Allele
X	ATG
X	GGG
Y	CTG

Sequences

SeqId	Seq
101	GCT
102	GGG
103	GGG

- String operators: `SELECT * FROM Sequences WHERE Seq LIKE '%ACG%';`
- Similarity joins: join **DiseaseAlleles** and **Sequences** allowing some errors

Aside: Stringology questions on relational data

There are many string algorithm problems inside relational database systems!

DiseaseAlleles

Disease	Allele
X	ATG
X	GGG
Y	CTG

Sequences

SeqId	Seq
101	GCT
102	GGG
103	GGG

- String operators: `SELECT * FROM Sequences WHERE Seq LIKE '%ACG%';`
- Similarity joins: join **DiseaseAlleles** and **Sequences** allowing some errors
- Embedded JSON/XML data in database fields

Aside: Stringology questions on relational data

There are many string algorithm problems inside relational database systems!

DiseaseAlleles

Disease	Allele
X	ATG
X	GGG
Y	CTG

Sequences

SeqId	Seq
101	GCT
102	GGG
103	GGG

- String operators: `SELECT * FROM Sequences WHERE Seq LIKE '%ACG%';`
- Similarity joins: join **DiseaseAlleles** and **Sequences** allowing some errors
- Embedded JSON/XML data in database fields

→ Can we have efficient plans (join optimization, string queries, indexes, etc.)?

Aside: Stringology questions on relational data

There are many string algorithm problems inside relational database systems!

DiseaseAlleles

Disease	Allele
X	ATG
X	GGG
Y	CTG

Sequences

SeqId	Seq
101	GCT
102	GGG
103	GGG

- String operators: `SELECT * FROM Sequences WHERE Seq LIKE '%ACG%';`
- Similarity joins: join **DiseaseAlleles** and **Sequences** allowing some errors
- Embedded JSON/XML data in database fields

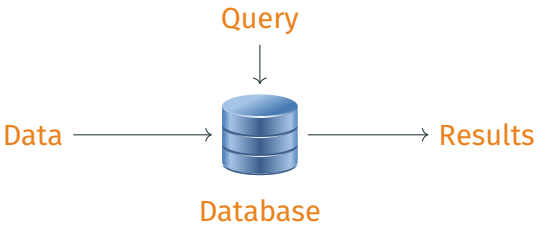
→ Can we have efficient plans (join optimization, string queries, indexes, etc.)?

→ Some practical research, but relatively little theory

→ Not the focus of this talk

Many query features

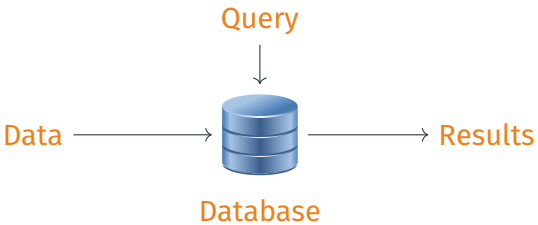
Across all kinds of data, different query features are needed



Many query features

Across all kinds of data, different query features are needed

- Enumerating query results
  - Possibly according to an order
  - Possibly supporting direct access

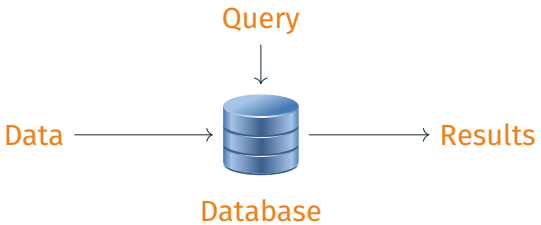


Many query features

Across all kinds of data, different query features are needed

- Incremental maintenance when the data is updated
- Read the data in streaming
- Compressed data representations

- Enumerating query results
  - Possibly according to an order
  - Possibly supporting direct access



Roadmap of this talk

- Main focus: document spanners to query textual documents
  - Efficient enumeration algorithms
  - Efficient incremental maintenance
  - Efficient algorithms on compressed text
  - Ranked enumeration and ranked direct access
- Extending to trees
- Conclusion

Document Spanners

Database motivation

- Declarative Information Extraction:
  - Specify what we want to extract from text

    “pairs of a first name and a phone number separated by whitespace”
  - Not how we want to extract it

    “when a word starts with a capital then go until the next whitespace then if ...”

Database motivation

- Declarative Information Extraction:
  - Specify what we want to extract from text
    "pairs of a first name and a phone number separated by whitespace"
  - Not how we want to extract it
    "when a word starts with a capital then go until the next whitespace then if ..."
- Create a relational table of results (mappings)
- Evaluate relational algebra queries involving such tables
  → If possible without materializing them

Formalizing document spanners

- **Document:** string over an alphabet

Formalizing document spanners

- **Document:** string over an alphabet

$T =$	J	o	h	n	␣	4	5	6	1	2	3	
	0	1	2	3	4	5	6	7	8	9	10	11

Formalizing document spanners

- **Document:** string over an alphabet

$T =$	J	o	h	n	␣	4	5	6	1	2	3	
	0	1	2	3	4	5	6	7	8	9	10	11

- **Span:** interval of positions
  → ex: $[0, 4\rangle$, $[5, 11\rangle$

Formalizing document spanners

- **Document:** string over an alphabet

$T =$	J	o	h	n	␣	4	5	6	1	2	3
	0	1	2	3	4	5	6	7	8	9	10	11

- **Span:** interval of positions
  - → ex: $[0, 4\rangle$, $[5, 11\rangle$
- **Mapping** over a set of variables X: function from X to spans
  - → ex: for $X = \{x, y\}$, map x to $[0, 4\rangle$ and map y to $[5, 11\rangle$
  - → *intuitively: each mapping is one extraction result*

Formalizing document spanners

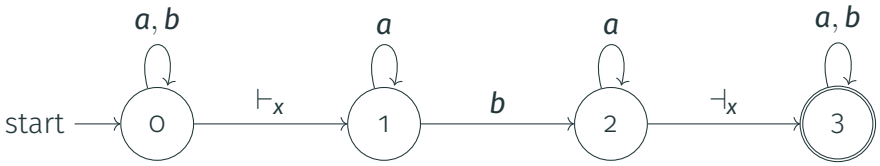
- **Document:** string over an alphabet

$T =$	J	o	h	n	␣	4	5	6	1	2	3
	0	1	2	3	4	5	6	7	8	9	10	11

- **Span:** interval of positions
  - → ex: $[0, 4\rangle$, $[5, 11\rangle$
- **Mapping** over a set of variables X: function from X to spans
  - → ex: for $X = \{x, y\}$, map x to $[0, 4\rangle$ and map y to $[5, 11\rangle$
  - → *intuitively: each mapping is one extraction result*
- **Spanner**: function that maps each document to a set of mappings
  - → ex: for $X = \{x, y\}$, each document is mapped to a **relational table**
    - → The columns are x, y
    - → Each row is one mapping over X (= one result)
    - → *intuitively: the table is the entire result on the input document*

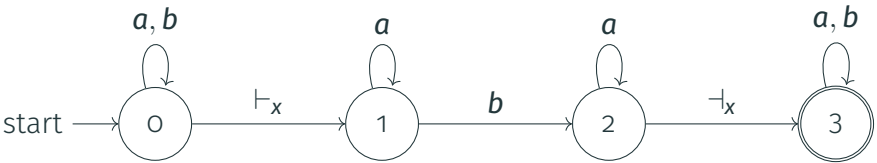
Regular document spanners

Simplest formalism: regular spanners, representable by variable-set automata



Regular document spanners

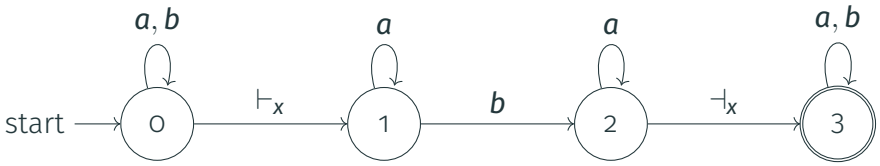
Simplest formalism: regular spanners, representable by variable-set automata



*"Extract all substrings containing exactly one **b**"*

Regular document spanners

Simplest formalism: regular spanners, representable by variable-set automata

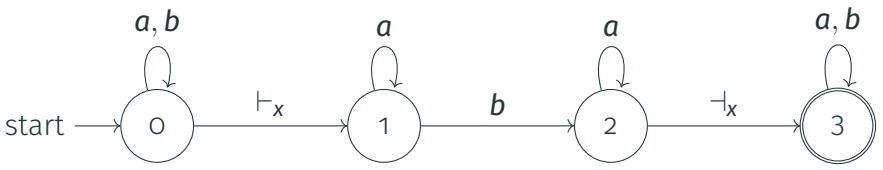


"Extract all substrings containing exactly one **b***"*

$$T = \begin{array}{cccccc} \texttt{b} & \texttt{b} & \texttt{a} & \texttt{b} \\ 0 & 1 & 2 & 3 & 4 \end{array}$$

Regular document spanners

Simplest formalism: regular spanners, representable by variable-set automata



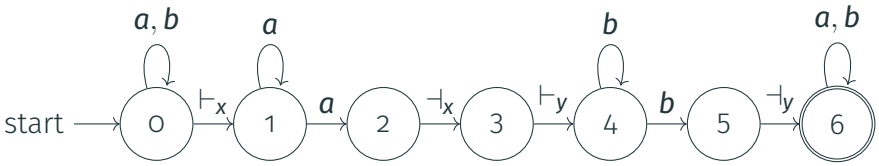
x
[0, 1⟩
[1, 2⟩
[1, 3⟩
[2, 4⟩
[3, 4⟩

"Extract all substrings containing exactly one **b***"*

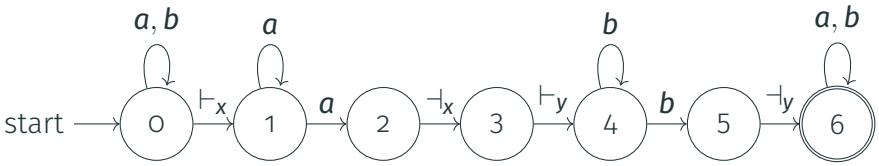
$T =$ b b a b

0 1 2 3 4

Regular document spanners (second example)

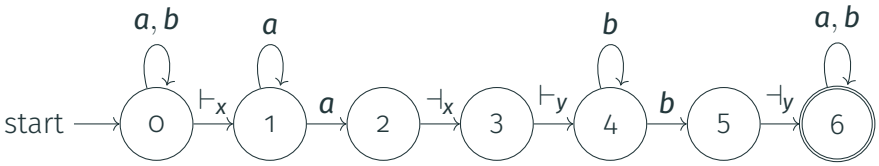


Regular document spanners (second example)



*"Extract all nonempty contiguous substrings of **a**'s followed by nonempty contiguous substrings of **b**'s"*

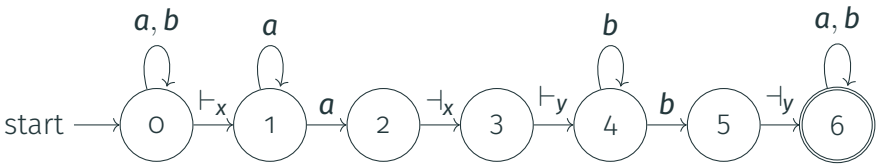
Regular document spanners (second example)



"Extract all nonempty contiguous substrings of **a***'s followed by nonempty contiguous substrings of* **b***'s"*

$$T = \begin{array}{cccccc} \texttt{a} & \texttt{b} & \texttt{a} & \texttt{b} & \texttt{b} & \\ 0 & 1 & 2 & 3 & 4 & 5 \end{array}$$

Regular document spanners (second example)



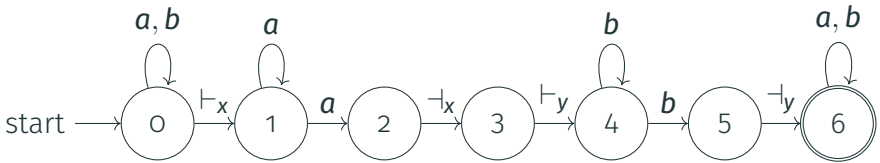
x	y
$[0, 1\rangle$	$[1, 2\rangle$
$[2, 3\rangle$	$[3, 4\rangle$
$[2, 3\rangle$	$[3, 5\rangle$

*"Extract all nonempty contiguous substrings of **a**'s followed by nonempty contiguous substrings of **b**'s"*

$$T = \begin{array}{cccccc} \texttt{a} & \texttt{b} & \texttt{a} & \texttt{b} & \texttt{b} & \\ 0 & 1 & 2 & 3 & 4 & 5 \end{array}$$

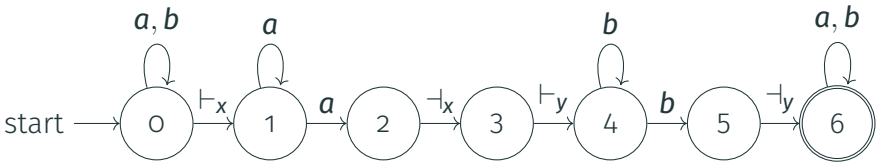
Writing regular document spanners

Variable-set automata are not always concise to write:



Writing regular document spanners

Variable-set automata are not always concise to write:

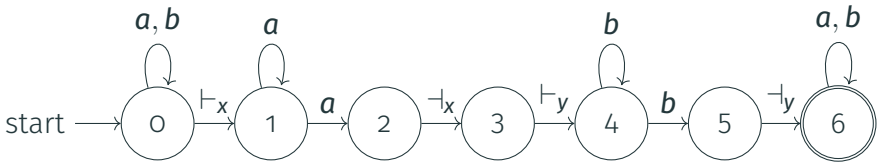


In practice, often more convenient to write as regex-formulas:

$$\Sigma^* \; \mathtt{x\{\; a^+ \;\}\; y\{\; b^+ \;\}}\; \Sigma^*$$

Writing regular document spanners

Variable-set automata are not always concise to write:



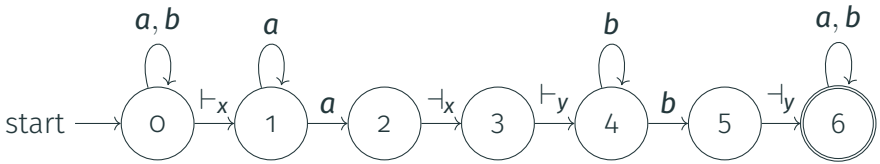
In practice, often more convenient to write as regex-formulas:

$$\Sigma^* \; \texttt{x\{ a}^+ \texttt{ \} y\{ b}^+ \texttt{ \}} \; \Sigma^*$$

These are just regular expressions with named capture variables!

Writing regular document spanners

Variable-set automata are not always concise to write:



In practice, often more convenient to write as regex-formulas:

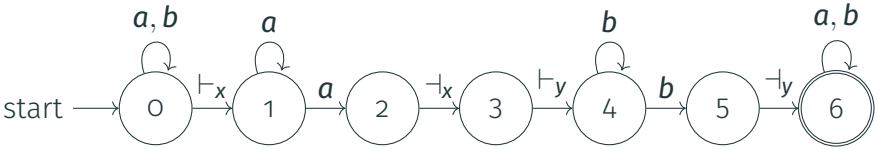
$$\Sigma^* \ \mathtt{x\{}\ \mathtt{a}^+\ \mathtt{\}}\ \mathtt{y\{}\ \mathtt{b}^+\ \mathtt{\}}\ \Sigma^*$$

These are just regular expressions with named capture variables!

Small expressiveness difference with vset-automata: hierarchicality

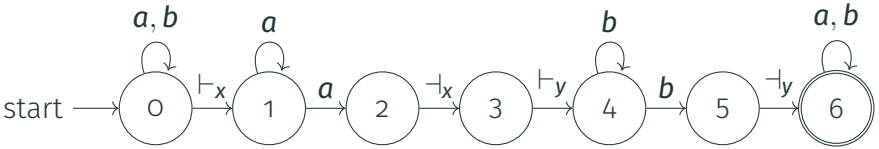
→ Their mappings can never have two incomparable spans that overlap

Some annoying issues



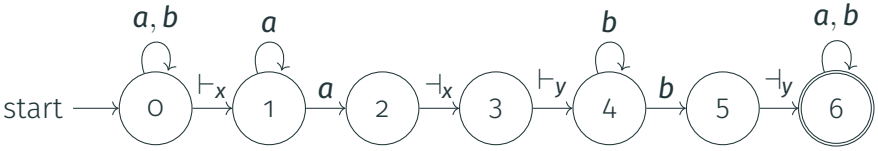
- We assume throughout that vset-automata are functional
  - Every variable opened and then closed exactly once

Some annoying issues



- We assume throughout that vset-automata are functional
  - Every variable opened and then closed exactly once
- There can be multiple runs defining the same mapping
  - With same order of the marker symbols $\vdash_x / \dashv_x$ or without
    - We may silently assume if necessary that the order on markers has been fixed
  - Unambiguous vset-automaton: at most one run for each mapping

Some annoying issues

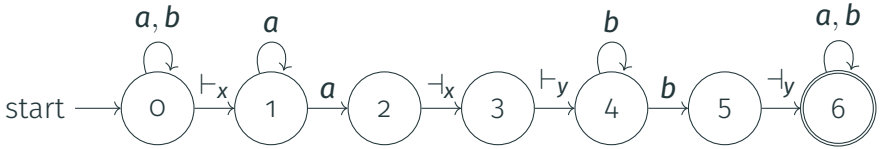


- We assume throughout that vset-automata are functional
  - Every variable opened and then closed exactly once
- There can be multiple runs defining the same mapping
  - With same order of the marker symbols $\vdash_x / \dashv_x$ or without
    - We may silently assume if necessary that the order on markers has been fixed
  - Unambiguous vset-automaton: at most one run for each mapping

New formalism of extractors that avoids most problems [Schmid, 2026]

- Each transition reads a letter and guesses a set of annotation variables
- Corresponds to a logic, i.e., monadic second-order logic (MSO)

Some annoying issues



- We assume throughout that vset-automata are functional
  - Every variable opened and then closed exactly once
- There can be multiple runs defining the same mapping
  - With same order of the marker symbols $\vdash_x / \dashv_x$ or without
    - We may silently assume if necessary that the order on markers has been fixed
  - Unambiguous vset-automaton: at most one run for each mapping

New formalism of extractors that avoids most problems [Schmid, 2026]

- Each transition reads a letter and guesses a set of annotation variables
- Corresponds to a logic, i.e., monadic second-order logic (MSO)
- However: blurs the correspondence with substrings / string equality
- Also, extractors are not always unambiguous

What can you do with spanners?

- **Pattern matching:** Extract all occurrences of word “stringology”:

$$\Sigma^* \; \mathtt{x\{\; stringology \;\}} \; \Sigma^*$$

What can you do with spanners?

- Pattern matching: Extract all occurrences of word “stringology”:

$$\Sigma^* \; \mathtt{x}\{ \; \mathtt{stringology} \; \} \; \Sigma^*$$

- Extract all occurrences of substrings satisfying a regular expression e

$$\Sigma^* \; \mathtt{x}\{e\} \; \Sigma^*$$

What can you do with spanners?

- Pattern matching: Extract all occurrences of word “stringology”:

$$\Sigma^* \; \mathtt{x\{\; stringology \;\}} \; \Sigma^*$$

- Extract all occurrences of substrings satisfying a regular expression e

$$\Sigma^* \; \mathtt{x}\{e\} \; \Sigma^*$$

- Test if the entire word satisfies a regular expression e

Is there a result for $\mathtt{x}\{e\}$?

What can you do with spanners?

- Pattern matching: Extract all occurrences of word “stringology”:

$$\Sigma^* \; \mathtt{x}\{ \; \mathtt{stringology} \; \} \; \Sigma^*$$

- Extract all occurrences of substrings satisfying a regular expression e

$$\Sigma^* \; \mathtt{x}\{e\} \; \Sigma^*$$

- Test if the entire word satisfies a regular expression e

Is there a result for $\mathtt{x}\{e\}$?

- Find all matches of pattern with variables xax, using string equalities (i.e., a spanner but not a regular spanner)

$\Sigma^* \; \mathtt{x}\{\Sigma^*\} \; a \; \mathtt{x'}\{\Sigma^*\} \; \Sigma^*$ with string equality $\mathtt{x} = \mathtt{x'}$

Classes of spanners

Pattern matching

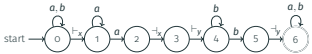
$\Sigma^*\ \mathtt{x\{\ stringology\ \}}\ \Sigma^*$

↓

Regular expressions

$\Sigma^*\ \mathtt{x}\{e\}\ \Sigma^*$

Regular spanners



Classes of spanners

Pattern matching

$\Sigma^*\ \mathtt{x\{\ stringology\ \}}\ \Sigma^*$

↓

Regular expressions

$\Sigma^*\ \mathtt{x}\{e\}\ \Sigma^*$

Regular spanners



↓

Refl-spanners

adds back-references

↓

Core spanners

adds string equality

↓

Generalized core spanners

adds complementation (set difference)

Classes of spanners

Pattern matching

$\Sigma^* \; x\{ \texttt{stringology} \} \; \Sigma^*$

↓

Regular expressions

$\Sigma^* \; x\{e\} \; \Sigma^*$

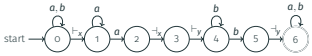
→

Regular patterns

$x_1\{\Sigma^*\} \; w_1 \ldots x_k\{\Sigma^*\} \; w_k \; x_{k+1}\{\Sigma^*\}$

↓

Regular spanners



↓

Refl-spanners

adds back-references

↓

Core spanners

adds string equality

↓

Generalized core spanners

adds complementation (set difference)

Classes of spanners

Pattern matching

$\Sigma^* \, x\{ \texttt{stringology} \} \, \Sigma^*$

↓

Regular expressions

$\Sigma^* \, x\{e\} \, \Sigma^*$

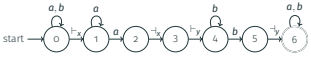
→

Regular patterns

$x_1\{\Sigma^*\} \, w_1 \ldots x_k\{\Sigma^*\} \, w_k \, x_{k+1}\{\Sigma^*\}$

↓

Regular spanners



→

Patterns with variables

like regular patterns
but with string equality

↓

Refl-spanners

adds back-references

↓

Core spanners

adds string equality

↓

Generalized core spanners

adds complementation (set difference)

Classes of spanners

Pattern matching

$\Sigma^* \; x\{ \texttt{stringology} \} \; \Sigma^*$

↓

Regular expressions

$\Sigma^* \; x\{e\} \; \Sigma^*$

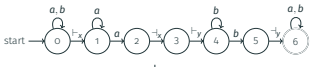
Regular patterns

$x_1\{\Sigma^*\} \; w_1 \ldots x_k\{\Sigma^*\} \; w_k \; x_{k+1}\{\Sigma^*\}$

Patterns with variables

like regular patterns
but with string equality

Regular spanners



Extraction grammars

[Peterfreund, 2021]

Annotated grammars

[A., Jachiet, Muñoz, Riveros, 2022]

spanners via context-free grammars

Refl-spanners

adds back-references

↓

Core spanners

adds string equality

↓

Generalized core spanners

adds complementation (set difference)

Core spanners and refl-spanners

Core spanners: extend regular spanners with

- The string equality operator $x = y$
  - $\rightarrow$ x and y must be mapped to spans corresponding to the same substring
- The ability to project away (forget) variables
- Example: $\Sigma^* \, \mathtt{x}\{ \, \mathtt{y}\{\Sigma^*\} \, \mathtt{z}\{\Sigma^*\} \, \} \, \Sigma^*$ with $\mathtt{y} = \mathtt{z}$ and projecting away $\mathtt{y}$ and $\mathtt{z}$
  - $\rightarrow$ each mapping assigns x to a span which contains a square (i.e., ww for $w \in \Sigma^*$)

Core spanners and refl-spanners

Core spanners: extend regular spanners with

- The string equality operator $x = y$
  - $\rightarrow$ x and y must be mapped to spans corresponding to the same substring
- The ability to project away (forget) variables
- Example: $\Sigma^*\ \mathtt{x}\{\ \mathtt{y}\{\Sigma^*\}\ \mathtt{z}\{\Sigma^*\}\ \}\ \Sigma^*$ with $\mathtt{y} = \mathtt{z}$ and projecting away $\mathtt{y}$ and $\mathtt{z}$
  - $\rightarrow$ each mapping assigns x to a span which contains a square (i.e., ww for $w \in \Sigma^*$)

Refl-spanners [Schmid and Schweikardt, 2024]:

- Simply allow back-references
- Example: $\Sigma^*\ \mathtt{x}\{\ \mathtt{y}\{\Sigma^*\}\ \backslash\mathtt{y}\ \}\ \Sigma^*$
- Less expressive but better computational properties

Tasks on spanners

I will focus on query evaluation tasks for document spanners:

- Computing and enumerating the output mappings on an input document
- Supporting updates to the input document and streaming
- Supporting compressed representations of the input
- Supporting ranked enumeration and direct access

Tasks on spanners

I will focus on query evaluation tasks for document spanners:

- Computing and enumerating the output mappings on an input document
- Supporting updates to the input document and streaming
- Supporting compressed representations of the input
- Supporting ranked enumeration and direct access

But there are other tasks (maybe less relevant to stringology):

- Static analysis tasks, like:
  - Satisfiability: given a spanner, is there a document giving a non-empty result?
  - Inclusion: given two spanners, does one always return a superset of mappings?
  - Equivalence: given two spanners, do they always return the same mappings?
- Studying expressiveness of spanner classes
  - Alternative: state complexity bounds
- Logical view: the logic FC [Freydenberger and Peterfreund, 2021]
- Interplay with relational algebra

Enumeration

Task: Enumerating results of spanners

- Problem description:

Task: Enumerating results of spanners

- Problem description:
  - Input:
    - A string T — “Organizers <seminar@dagstuhl.de>”
    - A spanner P — Σ^* < x{ [a-z]$^+$ @ [a-z]$^+$. [a-z]$^+$ } > Σ^*

Task: Enumerating results of spanners

- Problem description:
  - Input:
    - A string T — "Organizers <seminar@dagstuhl.de>"
    - A spanner P — $\Sigma^* \texttt{ < x\{ [a-z]}^+ \texttt{ @ [a-z]}^+ \texttt{ . [a-z]}^+ \texttt{ \} > } \Sigma^*$
  - Output: the list of results (mappings) of P on T

$$\{x : [12, 31\rangle\}, \ \ldots$$

Task: Enumerating results of spanners

- Problem description:
  - Input:
    - A string T — “Organizers <seminar@dagstuhl.de>”
    - A spanner P — $\Sigma^* \texttt{ < x\{ [a-z]}^+ \texttt{ @ [a-z]}^+ \texttt{ . [a-z]}^+ \texttt{ \} > } \Sigma^*$
  - Output: the list of results (mappings) of P on T

$$\{x : [12, 31\rangle\}, \ \ldots$$

- Goal:
  - Data complexity: Efficiency in T for fixed P
  - Combined complexity: Efficiency in T and P

Task: Enumerating results of spanners

- Problem description:
  - Input:
    - A string T — “Organizers <seminar@dagstuhl.de>”
    - A spanner P — $\Sigma^* \texttt{ < x\{ [a-z]}^+ \texttt{ @ [a-z]}^+ \texttt{ . [a-z]}^+ \texttt{ \} > } \Sigma^*$
  - Output: the list of results (mappings) of P on T

$$\{x : [12, 31\rangle\}, \ \ldots$$

- Goal:
  - Data complexity: Efficiency in T for fixed P
  - Combined complexity: Efficiency in T and P

Problem: the set of mappings may be very large, i.e., $\Theta(|T|^{2k})$ where $k := |X|$

Task: Enumerating results of spanners

- Problem description:
  - Input:
    - A string T — "Organizers <seminar@dagstuhl.de>"
    - A spanner P — $\Sigma^* \langle x\{ \texttt{[a-z]}^+ \texttt{@} \texttt{[a-z]}^+ \texttt{.} \texttt{[a-z]}^+ \} \rangle \Sigma^*$
  - Output: the list of results (mappings) of P on T

$$\{x : [12, 31\rangle\}, \ \ldots$$

- Goal:
  - Data complexity: Efficiency in T for fixed P
  - Combined complexity: Efficiency in T and P

Problem: the set of mappings may be very large, i.e., $\Theta(|T|^{2k})$ where $k := |X|$

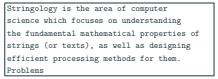
→ Measure complexity as a function of the input and output

→ Stringology: output-linear complexity in $O(\ldots + |\text{occ}|)$ (e.g., Aho-Corasick)

→ Database theory: enumeration algorithms with small delay

- Studied for conjunctive queries, e.g., [Segoufin, 2013]

Formalizing enumeration algorithms

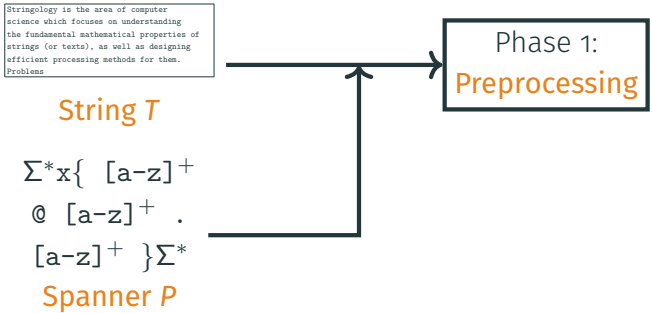


String T

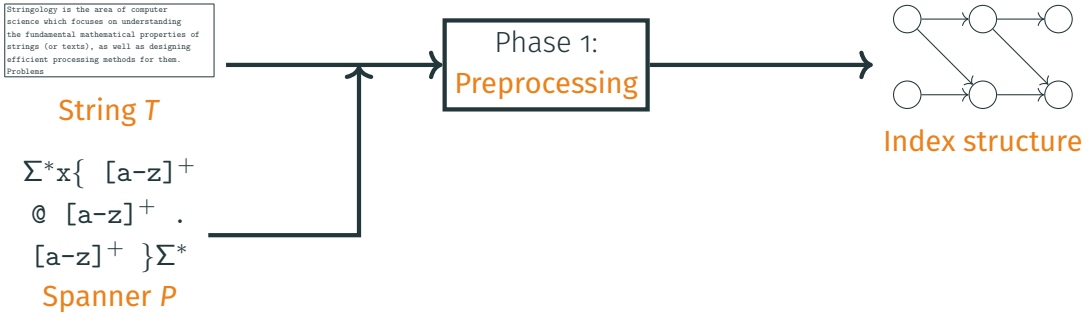
$\Sigma^* x\{\ [a\text{-}z]^+$
$@\ [a\text{-}z]^+\ .$
$[a\text{-}z]^+\ \}\Sigma^*$

Spanner P

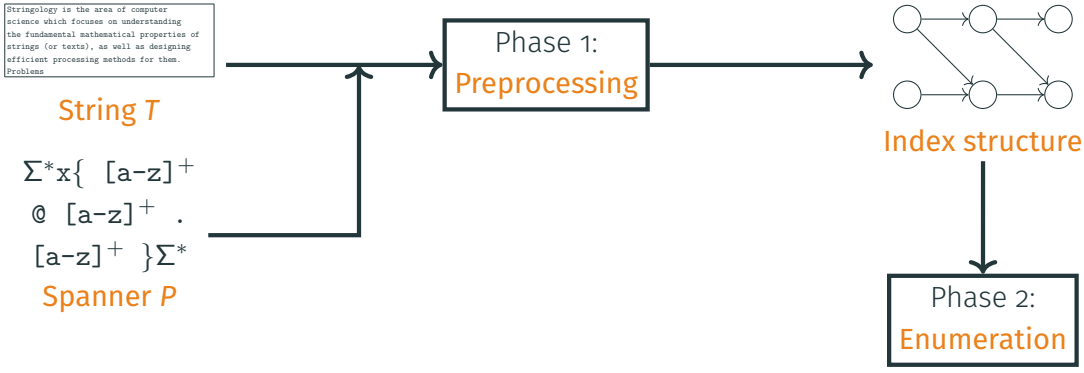
Formalizing enumeration algorithms



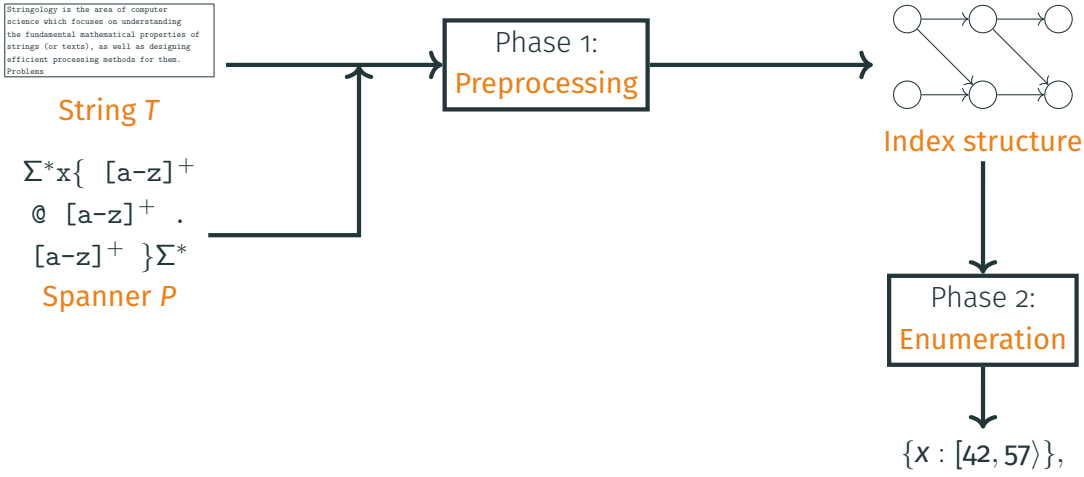
Formalizing enumeration algorithms



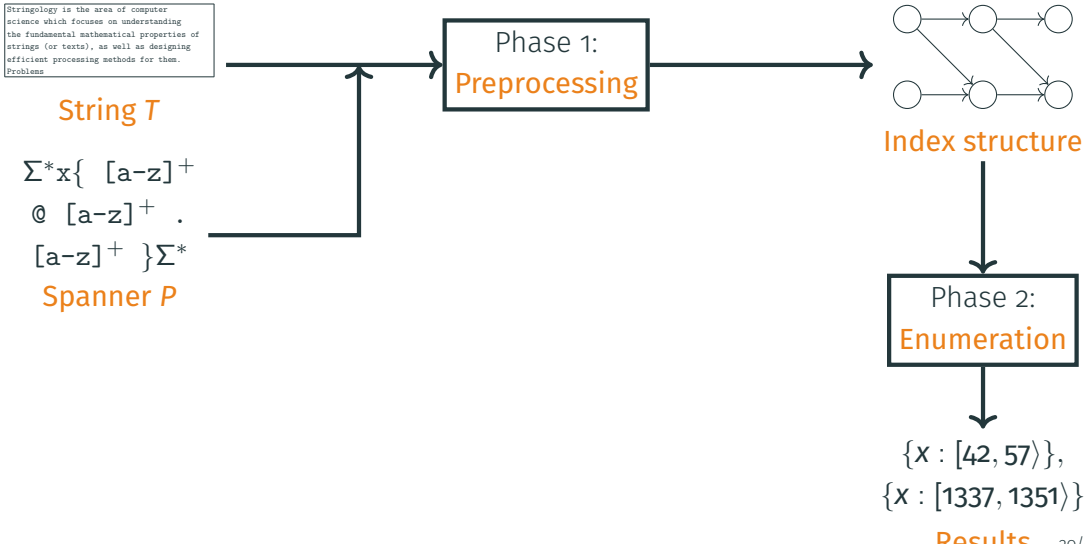
Formalizing enumeration algorithms



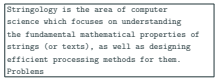
Formalizing enumeration algorithms



Formalizing enumeration algorithms



Formalizing enumeration algorithms

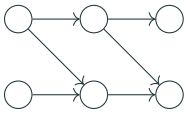


String T

$\Sigma^*x\{\ [a\text{-}z]^+$
$@\ [a\text{-}z]^+\ .$
$[a\text{-}z]^+\ \}\Sigma^*$

Spanner P

Phase 1:
Preprocessing



Index structure

Phase 2:
Enumeration

$\{x : [42, 57\rangle\},$
$\{x : [1337, 1351\rangle\}$

Results

Two ways to measure performance in **data** and **combined** complexity

- **Preprocessing time:** total time for phase 1
- **Delay:** worst-case time between two results in phase 2

... as a function of the string T and spanner P

Enumeration for regular spanners

Linear preprocessing and output-linear delay with determinism:

Theorem ([Florenzano, Riveros, Ugarte, Vansummeren, Vrgoč, 2018])

Given a string T and a deterministic vset-automaton A, we can enumerate the mappings of A on T with preprocessing $O(|T| \times |A|)$ and output-linear delay (proportional in the number of variables)

Enumeration for regular spanners

Linear preprocessing and output-linear delay with determinism:

Theorem ([Florenzano, Riveros, Ugarte, Vansummeren, Vrgoč, 2018])

Given a string T and a deterministic vset-automaton A, we can enumerate the mappings of A on T with preprocessing $O(|T|\times|A|)$ and output-linear delay (proportional in the number of variables)

Without determinism:

Theorem ([A., Bourhis, Mengel, Niewerth, 2019])

Given a string T and a nondeterministic vset-automaton A, we can enumerate the mappings of A on T with preprocessing $O(|T|\times Poly(|A|))$ and delay $O(Poly(|A|))$

Enumeration for regular spanners

Linear preprocessing and output-linear delay with determinism:

Theorem ([Florenzano, Riveros, Ugarte, Vansummeren, Vrgoč, 2018])

Given a string T and a deterministic vset-automaton A, we can enumerate the mappings of A on T with preprocessing $O(|T|\times|A|)$ and output-linear delay (proportional in the number of variables)

Without determinism:

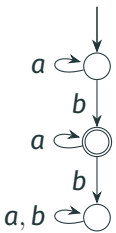
Theorem ([A., Bourhis, Mengel, Niewerth, 2019])

Given a string T and a nondeterministic vset-automaton A, we can enumerate the mappings of A on T with preprocessing $O(|T|\times Poly(|A|))$ and delay $O(Poly(|A|))$

- Already interesting for the spanner $\Sigma^* \, \mathtt{x}\{e\} \, \Sigma^*$ for a regular expression e
  $\rightarrow$ Enumerating all substrings matching e
- Less efficient than Knuth-Morris-Pratt / Aho-Corasick if the spanner is just extracting a fixed string or fixed dictionary

Proof: Enumerating all substrings for $\Sigma^* \, \mathrm{x}\{e\} \, \Sigma^*$ with a DFA

Efficiently enumerate all spans matching the DFA A "exactly one b" on string T

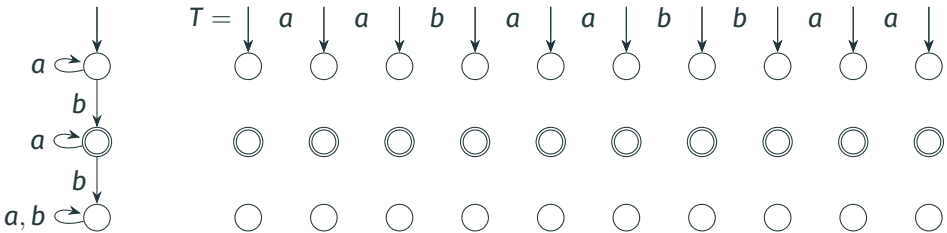


$T =$ a a b a a b b a a

Proof: Enumerating all substrings for $\Sigma^* \, x\{e\} \, \Sigma^*$ with a DFA

Efficiently enumerate all spans matching the DFA A "exactly one b" on string T

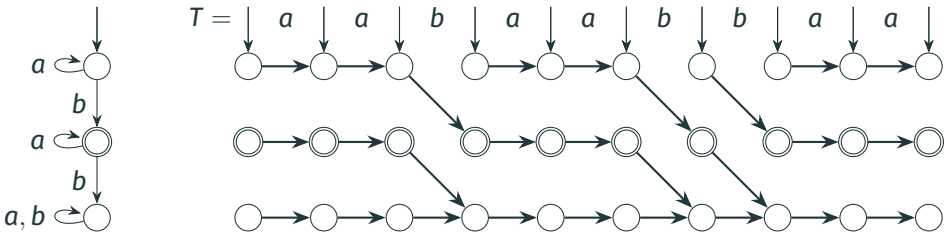
Build the product automaton in $O(|A| \times |T|)$



Proof: Enumerating all substrings for $\Sigma^* \, x\{e\} \, \Sigma^*$ with a DFA

Efficiently enumerate all spans matching the DFA A "exactly one b" on string T

Build the product automaton in $O(|A| \times |T|)$

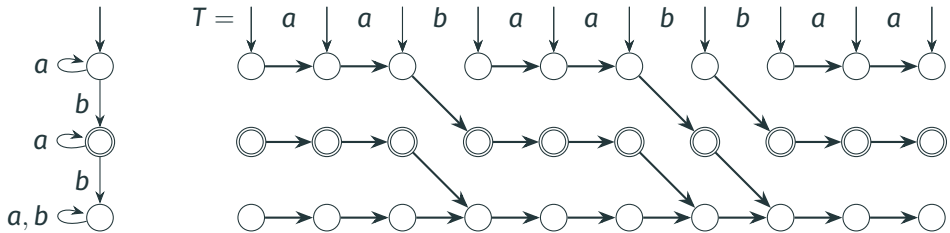


Proof: Enumerating all substrings for $\Sigma^* \, x\{e\} \, \Sigma^*$ with a DFA

Efficiently enumerate all spans matching the DFA A "exactly one b" on string T

Build the **product automaton** in $O(|A| \times |T|)$

$O(|T|)$ delay: For every starting position i, explore from (q_0, i) and produce $[i, j\rangle$ for all j where we reach (q_f, j) (with q_f final)



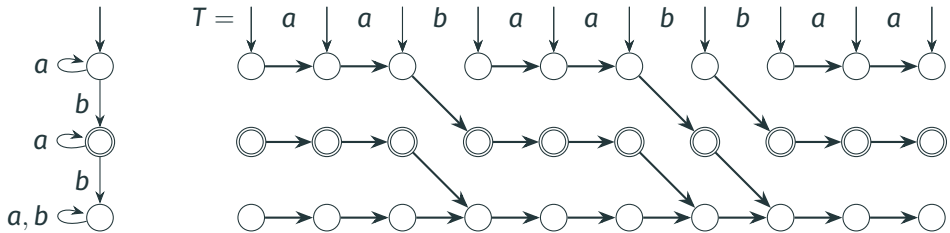
Proof: Enumerating all substrings for $\Sigma^* \, x\{e\} \, \Sigma^*$ with a DFA

Efficiently enumerate all spans matching the DFA A "exactly one b" on string T

Build the product automaton in $O(|A| \times |T|)$

$O(|T|)$ delay: For every starting position i, explore from (q_0, i) and produce $[i, j\rangle$ for all j where we reach (q_f, j) (with q_f final)

$O(1)$ delay: compute from right to left, for each (q, i), a pointer to the leftmost reachable (q_f, j) (with q_f final, and minimal $j > i$)



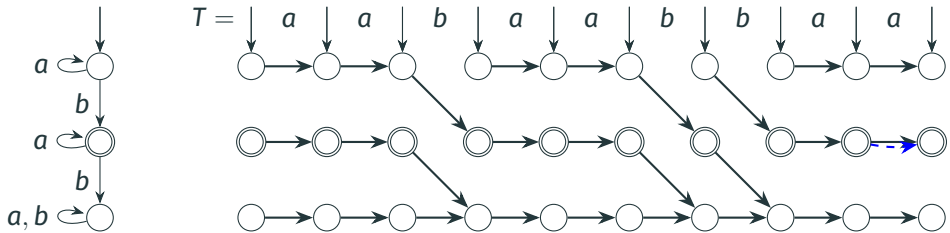
Proof: Enumerating all substrings for $\Sigma^* \, x\{e\} \, \Sigma^*$ with a DFA

Efficiently enumerate all spans matching the DFA A "exactly one b" on string T

Build the product automaton in $O(|A| \times |T|)$

$O(|T|)$ delay: For every starting position i, explore from (q_0, i) and produce $[i, j\rangle$ for all j where we reach (q_f, j) (with q_f final)

$O(1)$ delay: compute from right to left, for each (q, i), a pointer to the leftmost reachable (q_f, j) (with q_f final, and minimal $j > i$)



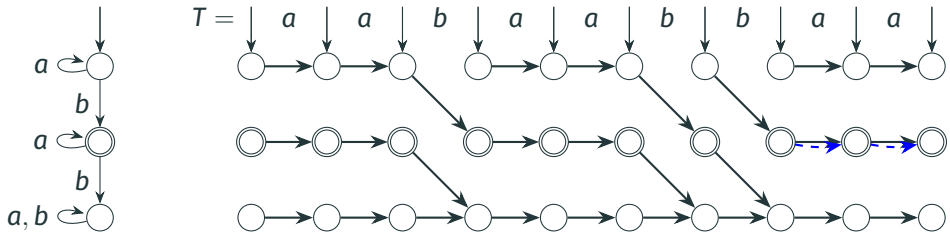
Proof: Enumerating all substrings for $\Sigma^* \, x\{e\} \, \Sigma^*$ with a DFA

Efficiently enumerate all spans matching the DFA A "exactly one b" on string T

Build the product automaton in $O(|A| \times |T|)$

$O(|T|)$ delay: For every starting position i, explore from (q_0, i) and produce $[i, j\rangle$ for all j where we reach (q_f, j) (with q_f final)

$O(1)$ delay: compute from right to left, for each (q, i), a pointer to the leftmost reachable (q_f, j) (with q_f final, and minimal $j > i$)



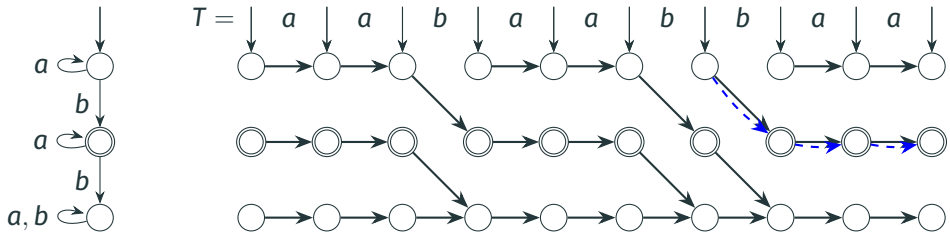
Proof: Enumerating all substrings for $\Sigma^* \, x\{e\} \, \Sigma^*$ with a DFA

Efficiently enumerate all spans matching the DFA A "exactly one b" on string T

Build the product automaton in $O(|A| \times |T|)$

$O(|T|)$ delay: For every starting position i, explore from (q_0, i) and produce $[i, j\rangle$ for all j where we reach (q_f, j) (with q_f final)

$O(1)$ delay: compute from right to left, for each (q, i), a pointer to the leftmost reachable (q_f, j) (with q_f final, and minimal $j > i$)



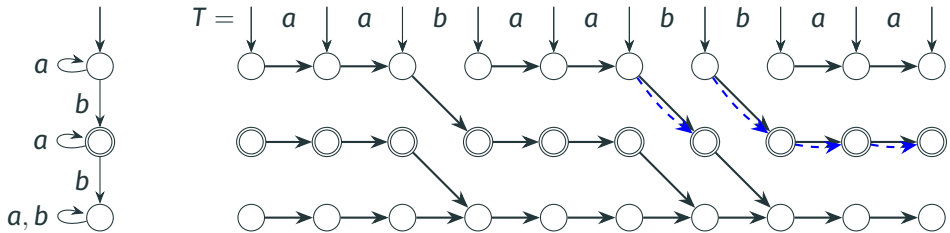
Proof: Enumerating all substrings for $\Sigma^* \, x\{e\} \, \Sigma^*$ with a DFA

Efficiently enumerate all spans matching the DFA A "exactly one b" on string T

Build the **product automaton** in $O(|A| \times |T|)$

$O(|T|)$ delay: For every starting position i, explore from (q_0, i) and produce $[i, j\rangle$ for all j where we reach (q_f, j) (with q_f final)

$O(1)$ delay: compute from right to left, for each (q, i), a **pointer** to the leftmost reachable (q_f, j) (with q_f final, and minimal $j > i$)



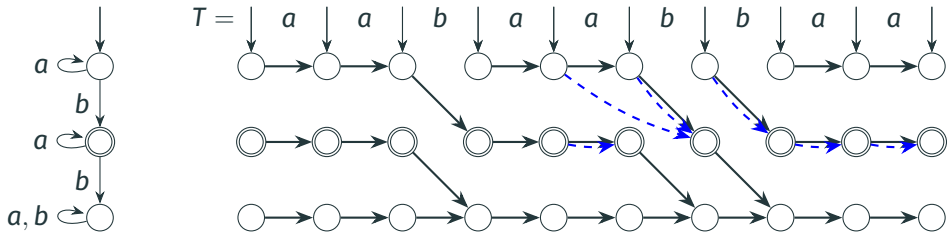
Proof: Enumerating all substrings for $\Sigma^* \, x\{e\} \, \Sigma^*$ with a DFA

Efficiently enumerate all spans matching the DFA A "exactly one b" on string T

Build the **product automaton** in $O(|A| \times |T|)$

$O(|T|)$ delay: For every starting position i, explore from (q_0, i) and produce $[i, j\rangle$ for all j where we reach (q_f, j) (with q_f final)

$O(1)$ delay: compute from right to left, for each (q, i), a **pointer** to the leftmost reachable (q_f, j) (with q_f final, and minimal $j > i$)



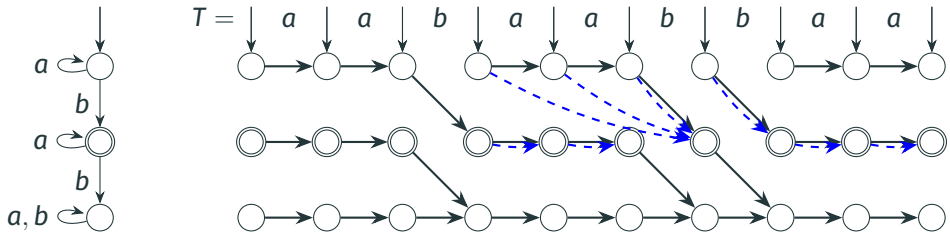
Proof: Enumerating all substrings for $\Sigma^* \, x\{e\} \, \Sigma^*$ with a DFA

Efficiently enumerate all spans matching the DFA A "exactly one b" on string T

Build the **product automaton** in $O(|A| \times |T|)$

$O(|T|)$ delay: For every starting position i, explore from (q_0, i) and produce $[i, j\rangle$ for all j where we reach (q_f, j) (with q_f final)

$O(1)$ delay: compute from right to left, for each (q, i), a **pointer** to the leftmost reachable (q_f, j) (with q_f final, and minimal $j > i$)



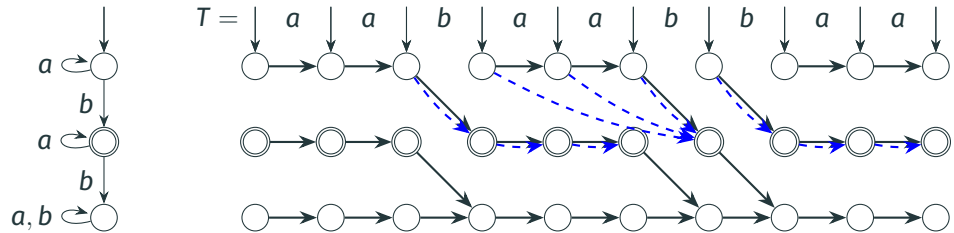
Proof: Enumerating all substrings for $\Sigma^* \, x\{e\} \, \Sigma^*$ with a DFA

Efficiently enumerate all spans matching the DFA A "exactly one b" on string T

Build the **product automaton** in $O(|A| \times |T|)$

$O(|T|)$ delay: For every starting position i, explore from (q_0, i) and produce $[i, j\rangle$ for all j where we reach (q_f, j) (with q_f final)

$O(1)$ delay: compute from right to left, for each (q, i), a **pointer** to the leftmost reachable (q_f, j) (with q_f final, and minimal $j > i$)



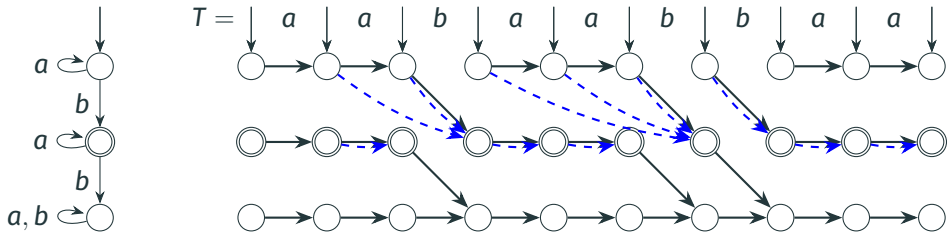
Proof: Enumerating all substrings for $\Sigma^* \, x\{e\} \, \Sigma^*$ with a DFA

Efficiently enumerate all spans matching the DFA A "exactly one b" on string T

Build the **product automaton** in $O(|A| \times |T|)$

$O(|T|)$ delay: For every starting position i, explore from (q_0, i) and produce $[i, j\rangle$ for all j where we reach (q_f, j) (with q_f final)

$O(1)$ delay: compute from right to left, for each (q, i), a **pointer** to the leftmost reachable (q_f, j) (with q_f final, and minimal $j > i$)



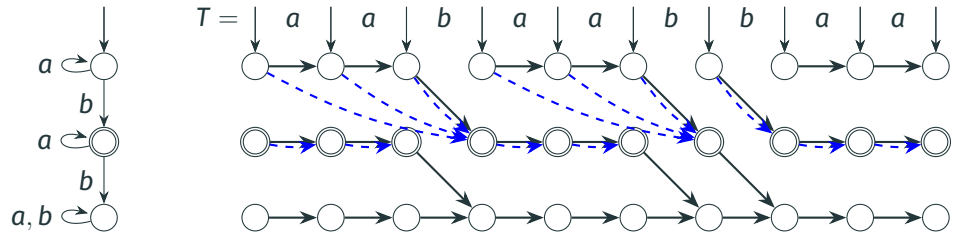
Proof: Enumerating all substrings for $\Sigma^* \, x\{e\} \, \Sigma^*$ with a DFA

Efficiently enumerate all spans matching the DFA A "exactly one b" on string T

Build the **product automaton** in $O(|A| \times |T|)$

$O(|T|)$ delay: For every starting position i, explore from (q_0, i) and produce $[i, j\rangle$ for all j where we reach (q_f, j) (with q_f final)

$O(1)$ delay: compute from right to left, for each (q, i), a **pointer** to the leftmost reachable (q_f, j) (with q_f final, and minimal $j > i$)



Proof: Enumerating all substrings for $\Sigma^* \, x\{e\} \, \Sigma^*$ with a DFA

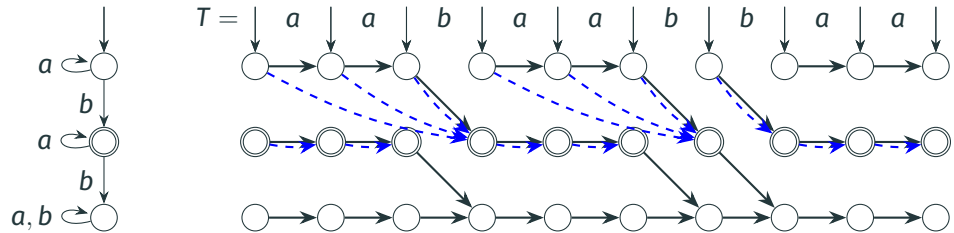
Efficiently enumerate all spans matching the DFA A "exactly one b" on string T

Build the product automaton in $O(|A| \times |T|)$

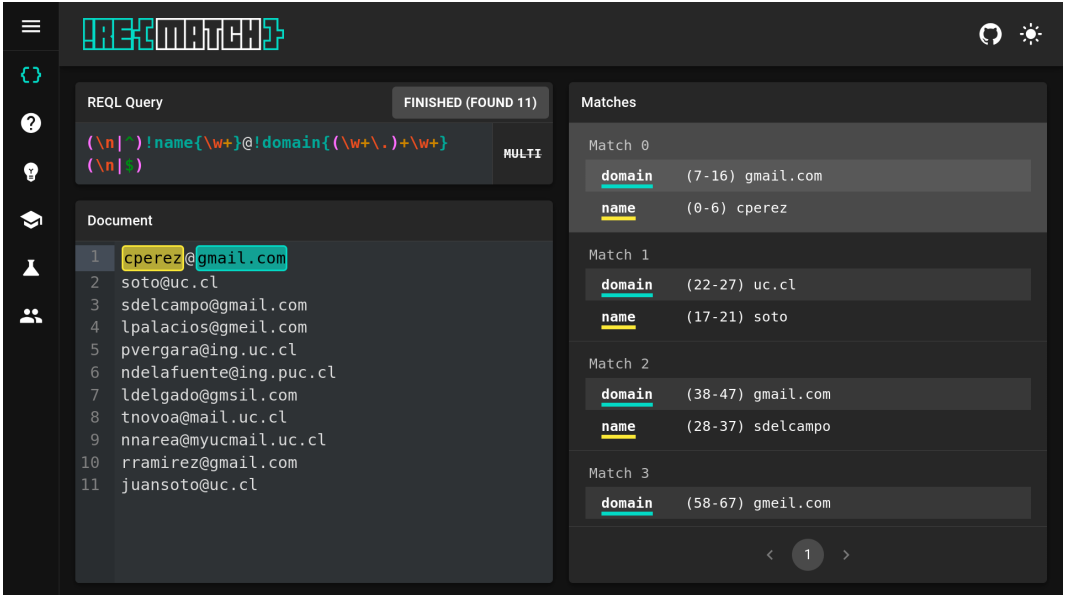
$O(|T|)$ delay: For every starting position i, explore from (q_0, i) and produce $[i, j\rangle$ for all j where we reach (q_f, j) (with q_f final)

$O(1)$ delay: compute from right to left, for each (q, i), a pointer to the leftmost reachable (q_f, j) (with q_f final, and minimal $j > i$)

For an NFA A, more complicated: state-set simulation with some tricks



REmatch [Bossonney, Calisto, Riveros, Toro, Van Sint Jan, Vrgoč, 2024]



Faster enumeration for regular patterns

Regular patterns: intersection of patterns with variables and regular spanners

$$x_1\{\Sigma^*\}\ w_1 \ldots x_k\{\Sigma^*\}\ w_k\ x_{k+1}\{\Sigma^*\}$$

Theorem ([Gawrychowski, Manea, Sarnighausen-Cahn, Siemer, 2026])
Given a string T and a regular pattern P, we can enumerate the mappings of P on T with preprocessing $O(|T|)$ and constant delay (in combined complexity)

Faster enumeration for regular patterns

Regular patterns: intersection of patterns with variables and regular spanners

$$x_1\{\Sigma^*\}\ w_1 \dots x_k\{\Sigma^*\}\ w_k\ x_{k+1}\{\Sigma^*\}$$

Theorem ([Gawrychowski, Manea, Sarnighausen-Cahn, Siemer, 2026])
Given a string T and a regular pattern P, we can enumerate the mappings of P on T with preprocessing $O(|T|)$ and constant delay (in combined complexity)

Wait: but the mappings have size $k + 1$ which is not constant??

Faster enumeration for regular patterns

Regular patterns: intersection of patterns with variables and regular spanners

$$x_1\{\Sigma^*\}\ w_1 \ldots x_k\{\Sigma^*\}\ w_k\ x_{k+1}\{\Sigma^*\}$$

Theorem ([Gawrychowski, Manea, Sarnighausen-Cahn, Siemer, 2026])
Given a string T and a regular pattern P, we can enumerate the mappings of P on T with preprocessing $O(|T|)$ and constant delay (in combined complexity)

Wait: but the mappings have size $k + 1$ which is not constant??

- Hence the model is enumeration via diffs:
  - Each mapping is produced by editing the previous mapping in place
  - Avoids accounting for the time to write all mappings

Cf. Stefan's talk this afternoon

Open question: Which other spanner algorithms can do enumeration via diffs?

Enumeration for refl-spanners and core spanners

Unfortunately intractable in combined complexity, because:

- Given a core spanner or refl-spanner P and a string T, it is NP-hard to know if some result exists

Enumeration for refl-spanners and core spanners

Unfortunately intractable in combined complexity, because:

- Given a core spanner or refl-spanner P and a string T, it is NP-hard to know if some result exists

So what can we do? (already for matching, without delay bounds)

Enumeration for refl-spanners and core spanners

Unfortunately intractable in combined complexity, because:

- Given a core spanner or refl-spanner P and a string T, it is NP-hard to know if some result exists

So what can we do? (already for matching, without delay bounds)

- Bound the treewidth (using results from patterns with variables)
  - cf. [Freydenberger and Peterfreund, 2021, Reidenbach and Schmid, 2014]
  - cf. Martín's talk this afternoon

Enumeration for refl-spanners and core spanners

Unfortunately intractable in combined complexity, because:

- Given a core spanner or refl-spanner P and a string T, it is NP-hard to know if some result exists

So what can we do? (already for matching, without delay bounds)

- Bound the treewidth (using results from patterns with variables)
  - cf. [Freydenberger and Peterfreund, 2021, Reidenbach and Schmid, 2014]
  - cf. Martín's talk this afternoon
- Very recent algorithms and lower bounds for related formalisms, based on the number of variables and of variable reuses:
  - [Kumabe and Uezato, 2026] for regular expressions with backreferences *(↔ refl-spanners)*
  - [Uezato, 2026] for regular-typed pattern languages *(subclass of refl-spanners)* with improved bounds in [Bille, Gørtz, Jessen, 2026]

Incremental Maintenance

Motivation

- Textual documents are often edited (or exist in multiple versions)
  - Also studied in database theory for conjunctive queries: [Berkholz et al., 2017]
- Can we maintain query results without recomputing them from scratch?

Motivation

- Textual documents are often edited (or exist in multiple versions)
  - Also studied in database theory for conjunctive queries: [Berkholz et al., 2017]
- Can we maintain query results without recomputing them from scratch?
- Focus on two problems:
  - Dynamic membership: maintaining membership to a regular language
  - Maintaining a structure to enumerate the mappings of a regular spanner

Motivation

- Textual documents are often edited (or exist in multiple versions)
  - Also studied in database theory for conjunctive queries: [Berkholz et al., 2017]
- Can we maintain query results without recomputing them from scratch?
- Focus on two problems:
  - Dynamic membership: maintaining membership to a regular language
  - Maintaining a structure to enumerate the mappings of a regular spanner
- Can be studied for various kinds of updates:
  - Simplest: substitution updates $w[42] := a$
  - Insertions and deletions *insert* a *before position 42 / delete at position 42*
  - Arbitrary split and join *split* w *at position 42 / concatenate* u *and* v

Dynamic membership for regular languages under substitutions

- Fix a regular language L
  - $\rightarrow$ E.g., $L = (ab)^*$

Dynamic membership for regular languages under substitutions

- Fix a regular language L
  - $\rightarrow$ E.g., $L = (ab)^*$
- Read an input string T with $n := |T|$
  - $\rightarrow$ E.g., $T = abbbab$

Dynamic membership for regular languages under substitutions

- Fix a regular language L
  - $\rightarrow$ E.g., $L = (ab)^*$
- Read an input string T with $n := |T|$
  - $\rightarrow$ E.g., $T = abbbab$
- Maintain the membership of T to L under substitution updates
  - $\rightarrow$ Initially, we have $T \notin L$
  - $\rightarrow$ Replace character at position 2 with a: we now have $T \in L$
  - $\rightarrow$ The length n never changes

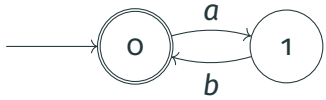
Dynamic membership for regular languages under substitutions

- Fix a regular language L
  - $\rightarrow$ E.g., $L = (ab)^*$
- Read an input string T with $n := |T|$
  - $\rightarrow$ E.g., $T = abbbab$
- Maintain the membership of T to L under substitution updates
  - $\rightarrow$ Initially, we have $T \notin L$
  - $\rightarrow$ Replace character at position 2 with a: we now have $T \in L$
  - $\rightarrow$ The length n never changes

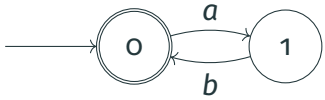
Theorem (folklore)

Given an NFA A recognizing a language L and given a string T, we can maintain dynamic membership of T to L under substitution updates in $O(Poly(|A|) \times \log |T|)$ per update

Proof of the $O(\log n)$ data complexity algorithm

Fix the language $L = (ab)^*$: 

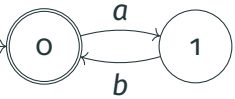
Proof of the $O(\log n)$ data complexity algorithm

Fix the language $L = (ab)^*$: 

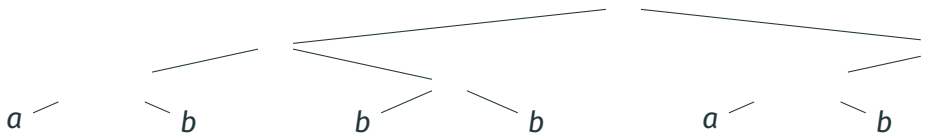
- Build a balanced binary tree on the input string $T = abbbab$

Proof of the $O(\log n)$ data complexity algorithm

Fix the language $L = (ab)^*$: start

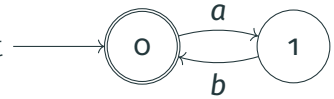


- Build a balanced binary tree on the input string $T = abbbab$

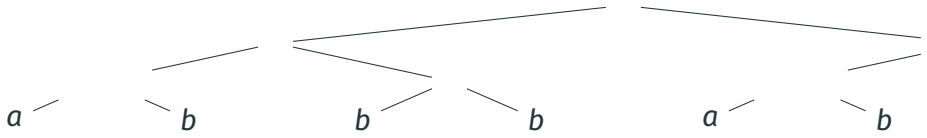


Proof of the $O(\log n)$ data complexity algorithm

Fix the language $L = (ab)^*$:

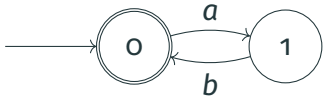


- Build a balanced binary tree on the input string $T = abbbab$
- Label each node x by the transition monoid element: all pairs $q \rightsquigarrow q'$ such that we can go from q to q' by reading the substring below x

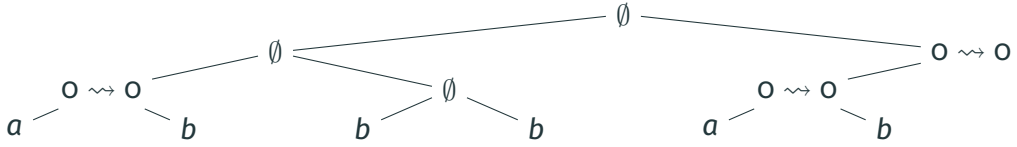


Proof of the $O(\log n)$ data complexity algorithm

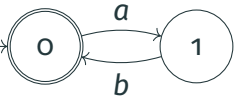
Fix the language $L = (ab)^*$:



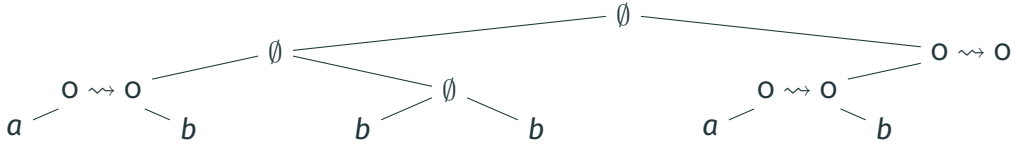
- Build a balanced binary tree on the input string $T = abbbab$
- Label each node x by the transition monoid element: all pairs $q \rightsquigarrow q'$ such that we can go from q to q' by reading the substring below x



Proof of the $O(\log n)$ data complexity algorithm

Fix the language $L = (ab)^*$: start 

- Build a balanced binary tree on the input string $T = abbbab$
- Label each node x by the transition monoid element: all pairs $q \rightsquigarrow q'$ such that we can go from q to q' by reading the substring below x



- The tree root describes if $T \in L$
- We can update the tree for each substitution in $O(\log n)$
- Can be improved to $O(\log n / \log \log n)$ data complexity with a log-ary tree
- Can be extended to support split/join if we keep the tree balanced

Remark: Improving on $O(\log n)$ data complexity for some languages

For our language $L = (ab)^*$ we can handle updates in $O(1)$ data complexity:

Remark: Improving on $O(\log n)$ data complexity for some languages

For our language $L = (ab)^*$ we can handle updates in $O(1)$ data complexity:

- Check that n is even
- Count violations: a's at odd positions and b's at even positions
- Maintain this counter in constant time
- We have $T \in L$ iff there are no violations

Remark: Improving on $O(\log n)$ data complexity for some languages

For our language $L = (ab)^*$ we can handle updates in $O(1)$ data complexity:

- Check that n is even
- Count violations: a's at odd positions and b's at even positions
- Maintain this counter in constant time
- We have $T \in L$ iff there are no violations

Theorem ([A., Jachiet, Paperman, 2021])

There is a data complexity trichotomy for substitution updates:

- *Regular languages in* $O(1)$ *per update*
- *Regular languages in* $O(\log \log n)$ *and conditionally not* $O(1)$ *per update*
- *Regular languages in* $\Theta(\log n / \log \log n)$ *per update*

Remark: Improving on $O(\log n)$ data complexity for some languages

For our language $L = (ab)^*$ we can handle updates in $O(1)$ data complexity:

- Check that n is even
- Count violations: a's at odd positions and b's at even positions
- Maintain this counter in constant time
- We have $T \in L$ iff there are no violations

Theorem ([A., Jachiet, Paperman, 2021])
There is a data complexity trichotomy for substitution updates:

- *Regular languages in* $O(1)$ *per update*
- *Regular languages in* $O(\log \log n)$ *and conditionally not* $O(1)$ *per update*
- *Regular languages in* $\Theta(\log n / \log \log n)$ *per update*

Open question: Everything is in data complexity; combined complexity is open

Dynamic enumeration for regular spanners

Let's move from dynamic membership for regular languages to the more expressive dynamic enumeration for regular spanners:

- Input:
  - A string T
  - A vset-automaton A expressing a regular spanner

Dynamic enumeration for regular spanners

Let's move from dynamic membership for regular languages to the more expressive dynamic enumeration for regular spanners:

- Input:
  - A string T
  - A vset-automaton A expressing a regular spanner
- After preprocessing, we must support two tasks:
  - Enumeration: produce the next mapping of A on T
  - Update: modify T, resetting the enumeration status

Dynamic enumeration for regular spanners

Let's move from dynamic membership for regular languages to the more expressive dynamic enumeration for regular spanners:

- Input:
  - A string T
  - A vset-automaton A expressing a regular spanner
- After preprocessing, we must support two tasks:
  - Enumeration: produce the next mapping of A on T
  - Update: modify T, resetting the enumeration status

Theorem ([Muñoz and Riveros, 2025])

Given a string T and an unambiguous vset-automaton A, we can enumerate the matches of A on T with preprocessing $O(|T| \times Poly(|A|))$, output-linear delay, and update time $O(Poly(|A|) \times \log |T|)$, for split/join updates

Dynamic enumeration for regular spanners

Let's move from dynamic membership for regular languages to the more expressive dynamic enumeration for regular spanners:

- Input:
  - A string T
  - A vset-automaton A expressing a regular spanner
- After preprocessing, we must support two tasks:
  - Enumeration: produce the next mapping of A on T
  - Update: modify T, resetting the enumeration status

Theorem ([Muñoz and Riveros, 2025])

Given a string T and an unambiguous vset-automaton A, we can enumerate the matches of A on T with preprocessing $O(|T| \times Poly(|A|))$, output-linear delay, and update time $O(Poly(|A|) \times \log |T|)$, for split/join updates

This is even true on compressed representations (presented later)

Open questions

- Can we ensure polynomial-time combined complexity without unambiguity?
  - Follows from [A., Bourhis, Mengel, Niewerth, 2019] for substitution updates
  - Should adapt to split/join updates, but not immediate from current paper
  - **Open question:** Can we improve the polynomial degree?

Open questions

- Can we ensure polynomial-time combined complexity without unambiguity?
  - Follows from [A., Bourhis, Mengel, Niewerth, 2019] for substitution updates
  - Should adapt to split/join updates, but not immediate from current paper
  - **Open question:** Can we improve the polynomial degree?
- Beat $O(\log n)$ data complexity for updates? (like dynamic membership)
  - Already non-obvious for spanners $\Sigma^* \, x\{e\} \, \Sigma^*$ with e a regular expression
  - Some limited $O(1)$-update cases in [A., Dziadek, Segoufin, 2026]
  - **Open question:** Restricted edits (e.g., can only substitute some characters)

Open questions

- Can we ensure polynomial-time combined complexity without unambiguity?
  - Follows from [A., Bourhis, Mengel, Niewerth, 2019] for substitution updates
  - Should adapt to split/join updates, but not immediate from current paper
  - **Open question:** Can we improve the polynomial degree?
- Beat $O(\log n)$ data complexity for updates? (like dynamic membership)
  - Already non-obvious for spanners $\Sigma^* \, x\{e\} \, \Sigma^*$ with e a regular expression
  - Some limited $O(1)$-update cases in [A., Dziadek, Segoufin, 2026]
  - **Open question:** Restricted edits (e.g., can only substitute some characters)
- What about more expressive spanners? (refl-spanners, grammars…)

  → Related to incremental parsing

Open questions

- Can we ensure polynomial-time combined complexity without unambiguity?
  - Follows from [A., Bourhis, Mengel, Niewerth, 2019] for substitution updates
  - Should adapt to split/join updates, but not immediate from current paper
  - **Open question:** Can we improve the polynomial degree?
- Beat $O(\log n)$ data complexity for updates? (like dynamic membership)
  - Already non-obvious for spanners $\Sigma^* \, x\{e\} \, \Sigma^*$ with e a regular expression
  - Some limited $O(1)$-update cases in [A., Dziadek, Segoufin, 2026]
  - **Open question:** Restricted edits (e.g., can only substitute some characters)
- What about more expressive spanners? (refl-spanners, grammars…)

  → Related to incremental parsing
- What about more expressive edits? (e.g., search and replace?)

Streaming

Special kind of updates: the string T is received from left to right in streaming

- Useful in areas like complex event processing
- One-pass processing is efficient in general (memory, cache locality, etc.)

→ Relates to stringology topics:

- Streaming pattern matching [Breslauer and Galil, 2014]
- Streaming regular expression matching [Dudek, Gawrychowski, Gourdel, Starikovskaya, 2022]

Streaming

Special kind of updates: the string T is received from left to right in streaming

- Useful in areas like complex event processing
- One-pass processing is efficient in general (memory, cache locality, etc.)

→ Relates to stringology topics:

- Streaming pattern matching [Breslauer and Galil, 2014]
- Streaming regular expression matching [Dudek, Gawrychowski, Gourdel, Starikovskaya, 2022]

→ Regular spanner enumeration results adapt with same data complexity

→ see [Muñoz and Riveros, 2024]

Streaming

Special kind of updates: the string T is received from left to right in streaming

- Useful in areas like complex event processing
- One-pass processing is efficient in general (memory, cache locality, etc.)

→ Relates to stringology topics:
  - Streaming pattern matching [Breslauer and Galil, 2014]
  - Streaming regular expression matching [Dudek, Gawrychowski, Gourdel, Starikovskaya, 2022]

→ Regular spanner enumeration results adapt with same data complexity
  → see [Muñoz and Riveros, 2024]

→ For tasks on a sliding window (push-right, pop-left), space complexity classification in [Ganardi, Hucke, Lohrey, Mamouras, Starikovskaya, 2025]
  - Beyond regular languages: [Ganardi, Jachiet, Lohrey, Schwentick, 2022]

Streaming

Special kind of updates: the string T is received from left to right in streaming

- Useful in areas like complex event processing
- One-pass processing is efficient in general (memory, cache locality, etc.)

→ Relates to stringology topics:
  - Streaming pattern matching [Breslauer and Galil, 2014]
  - Streaming regular expression matching [Dudek, Gawrychowski, Gourdel, Starikovskaya, 2022]

→ Regular spanner enumeration results adapt with same data complexity
  → see [Muñoz and Riveros, 2024]

→ For tasks on a sliding window (push-right, pop-left), space complexity classification in [Ganardi, Hucke, Lohrey, Mamouras, Starikovskaya, 2025]
  - Beyond regular languages: [Ganardi, Jachiet, Lohrey, Schwentick, 2022]

→ **Open question:** Precise amount of memory needed for earliest answering?

Compressed Inputs

Compressed inputs

Motivation: Sometimes textual data is highly compressible, and processing it without decompression is faster

Compressed inputs

Motivation: Sometimes textual data is highly compressible, and processing it without decompression is faster

In stringology, many works study such algorithms:

- On compressed data for different compression methods
- On collections of repetitive strings [Navarro, 2021a, Navarro, 2021b]

Compressed inputs

Motivation: Sometimes textual data is highly compressible, and processing it without decompression is faster

In stringology, many works study such algorithms:

- On compressed data for different compression methods
- On collections of repetitive strings [Navarro, 2021a, Navarro, 2021b]

In document spanners:

→ Only Straight-Line Programs (SLP) [Lohrey, 2012]

→ Question introduced by [Schmid and Schweikardt, 2021]

→ **Open question:** Is any other compression model relevant?

SLPs

An SLP is just a context-free grammar representing one single string:

- Alphabet N of non-terminals in addition to the alphabet Σ of terminals
- Distinguished axiom $S \in N$
- Each non-terminal $A \in N$ gives precisely one rule $A \to \beta$ with $\beta \in (N \cup \Sigma)^+$
- The dependency graph on non-terminals is acyclic

Example:

$$S \to \mathtt{b}A_8 \qquad A_8 \to A_4A_4 \qquad A_4 \to A_2A_2 \qquad A_2 \to \mathtt{aa}$$

represents `baaaaaaaa`

Enumerating matches of regular spanners on SLPs

Theorem ([Muñoz and Riveros, 2025])

Given an unambiguous vset-automaton A and an SLP T, we can enumerate the matches of A on the string represented by T with preprocessing $O(Poly(|A|) \times |T|)$, output-linear delay, and update time $O(Poly(|A|) \times \log |T|)$

The updates are directly on the SLP, cf. [Schmid and Schweikardt, 2022b]

$\rightarrow$ They feature in particular split/join

Enumerating matches of regular spanners on SLPs

Theorem ([Muñoz and Riveros, 2025])

Given an unambiguous vset-automaton A and an SLP T, we can enumerate the matches of A on the string represented by T with preprocessing $O(Poly(|A|) \times |T|)$, output-linear delay, and update time $O(Poly(|A|) \times \log |T|)$

The updates are directly on the SLP, cf. [Schmid and Schweikardt, 2022b]

→ They feature in particular split/join

Open question: Best combined complexity possible?

→ in particular without unambiguity?

→ some results on this in [Schmid and Schweikardt, 2021] (but not on delay)

Enumerating matches of regular spanners on SLPs

Theorem ([Muñoz and Riveros, 2025])

Given an unambiguous vset-automaton A and an SLP T, we can enumerate the matches of A on the string represented by T with preprocessing $O(Poly(|A|) \times |T|)$, output-linear delay, and update time $O(Poly(|A|) \times \log |T|)$

The updates are directly on the SLP, cf. [Schmid and Schweikardt, 2022b]

→ They feature in particular split/join

Open question: Best combined complexity possible?

→ in particular without unambiguity?

→ some results on this in [Schmid and Schweikardt, 2021] (but not on delay)

Open question: Connection to transducers? [Bojańczyk and Lohrey, 2026]

→ in particular to produce a compressed representation of the output?

Ranked Enumeration and Direct Access

Motivation

- Important to obtain query results in a relevant order
  - Studied for database queries [Carmeli et al., 2023, Tziavelis et al., 2024]
  - Stringology: access pattern occurrences in sorted order, top-k document retrieval, etc.

Motivation

- Important to obtain query results in a relevant order
  - Studied for database queries [Carmeli et al., 2023, Tziavelis et al., 2024]
  - Stringology: access pattern occurrences in sorted order, top-k document retrieval, etc.
- Supporting orders can be relevant for two problems:
  - Ranked enumeration, with enumeration as presented up to now
  - Ranked direct access: given index i, get the i-th solution according to the order

Motivation

- Important to obtain query results in a relevant order
  - Studied for database queries [Carmeli et al., 2023, Tziavelis et al., 2024]
  - Stringology: access pattern occurrences in sorted order, top-k document retrieval, etc.
- Supporting orders can be relevant for two problems:
  - Ranked enumeration, with enumeration as presented up to now
  - Ranked direct access: given index i, get the i-th solution according to the order
- At least two seemingly unrelated ways to define orders on results:
  - Logic-defined: write a formula / spanner to express the order
    - Supports in particular lexicographic order based on positions in the input string
  - Weight-defined: compute a weight along each mapping
    - cf. Pierre's talk this afternoon

Motivation

- Important to obtain query results in a relevant order
  - Studied for database queries [Carmeli et al., 2023, Tziavelis et al., 2024]
  - Stringology: access pattern occurrences in sorted order, top-k document retrieval, etc.
- Supporting orders can be relevant for two problems:
  - Ranked enumeration, with enumeration as presented up to now
  - Ranked direct access: given index i, get the i-th solution according to the order
- At least two seemingly unrelated ways to define orders on results:
  - Logic-defined: write a formula / spanner to express the order
    - Supports in particular lexicographic order based on positions in the input string
  - Weight-defined: compute a weight along each mapping
    - cf. Pierre's talk this afternoon
- Other possibility: list Pareto-optimal mappings according to a partial order
  - $\rightarrow$ e.g., if $\mu'(x)$ is a subspan of $\mu(x)$ for each x, do not produce μ'
  - Can make combined complexity intractable [A., Kimelfeld, Labbé, Mengel, 2024]

Setting 1: Logic-defined orders

For ranked enumeration (data complexity only, and no updates):

Theorem ([Lohrey, 2027], extending [Bojańczyk, 2018])

We can enumerate in an MSO-definable order the mappings of regular spanners on a string represented by an SLP T with preprocessing $O(|T|)$ and constant delay

Setting 1: Logic-defined orders

For ranked enumeration (data complexity only, and no updates):

Theorem ([Lohrey, 2027], extending [Bojańczyk, 2018])
We can enumerate in an MSO-definable order the mappings of regular spanners on a string represented by an SLP T with preprocessing $O(|T|)$ and constant delay

---

For ranked direct access:

Theorem ([Muñoz, 2026], extending [Bourhis, Capelli, Mengel, Riveros, 2025])
Given an unambiguous vset-automaton A and an SLP T, we can support direct access in lexicographic order to the results of A on the string represented by T with preprocessing $O(\mathit{Poly}(|A|)\times|T|)$, access time $O(\mathit{Poly}(|A|)\times \log|T|)$, and update time $O(\mathit{Poly}(|A|)\times \log|T|)$ (updates on the SLP, including split/join)

Setting 1: Logic-defined orders

For ranked enumeration (data complexity only, and no updates):

Theorem ([Lohrey, 2027], extending [Bojańczyk, 2018])
We can enumerate in an MSO-definable order the mappings of regular spanners on a string represented by an SLP T with preprocessing $O(|T|)$ and constant delay

---

For ranked direct access:

Theorem ([Muñoz, 2026], extending [Bourhis, Capelli, Mengel, Riveros, 2025])
Given an unambiguous vset-automaton A and an SLP T, we can support direct access in lexicographic order to the results of A on the string represented by T with preprocessing $O(Poly(|A|)\times|T|)$, access time $O(Poly(|A|)\times \log|T|)$, and update time $O(Poly(|A|)\times \log|T|)$ (updates on the SLP, including split/join)

Open question: What about ranked direct access for more expressive orders? improved combined complexity?

Remark: Counting query answers

Algorithms for direct access (ranked or not) must in particular be able to count the number of results (and maintain it under updates)

Remark: Counting query answers

Algorithms for direct access (ranked or not) must in particular be able to count the number of results (and maintain it under updates)

Note that this task is in fact intractable in combined complexity when we do not restrict to unambiguous vset-automata, because:

Theorem (See, e.g., [Irwin, 2022])

Given an NFA A and integer k (in unary), the problem #NFA of counting the number of distinct words of length k accepted by A is #P-hard under Turing reductions

Remark: Counting query answers

Algorithms for direct access (ranked or not) must in particular be able to count the number of results (and maintain it under updates)

Note that this task is in fact intractable in combined complexity when we do not restrict to unambiguous vset-automata, because:

Theorem (See, e.g., [Irwin, 2022])

Given an NFA A and integer k (in unary), the problem #NFA of counting the number of distinct words of length k accepted by A is #P-hard under Turing reductions

However, this admits an FPRAS [Arenas, Croquevielle, Jayaram, Riveros, 2021b]

Remark: Counting query answers

Algorithms for direct access (ranked or not) must in particular be able to count the number of results (and maintain it under updates)

Note that this task is in fact intractable in combined complexity when we do not restrict to unambiguous vset-automata, because:

Theorem (See, e.g., [Irwin, 2022])

Given an NFA A and integer k (in unary), the problem #NFA of counting the number of distinct words of length k accepted by A is #P-hard under Turing reductions

However, this admits an FPRAS [Arenas, Croquevielle, Jayaram, Riveros, 2021b]

Open question: Can we define a weaker form of direct access that cannot count?

Setting 2: Weight-defined orders

Most common approach:

- Cost transducer: Extend vset-automata to give a weight (in some ordered group) to each transition
  - → Gives a weight for each run
- Weight of a mapping = weight of the run that yields it (needs unambiguity)
  - → Other choice: semirings [Doleschal, Kimelfeld, Martens, Peterfreund, 2022]
- Relates to MSO cost functions [Bourhis, Grez, Jachiet, Riveros, 2021]

Setting 2: Weight-defined orders

Most common approach:

- Cost transducer: Extend vset-automata to give a weight (in some ordered group) to each transition
  - → Gives a weight for each run
- Weight of a mapping = weight of the run that yields it (needs unambiguity)
  - → Other choice: semirings [Doleschal, Kimelfeld, Martens, Peterfreund, 2022]
- Relates to MSO cost functions [Bourhis, Grez, Jachiet, Riveros, 2021]

Theorem ([Gawrychowski, Manea, Schmid, 2024])

Given an unambiguous cost transducer A and a string T, we can enumerate the results of A on T in order, with preprocessing $O(|T|)$ and output-linear delay (with high probability, or deterministically but with an added logarithmic term)

Setting 2: Weight-defined orders

Most common approach:

- Cost transducer: Extend vset-automata to give a weight (in some ordered group) to each transition
  - → Gives a weight for each run
- Weight of a mapping = weight of the run that yields it (needs unambiguity)
  - → Other choice: semirings [Doleschal, Kimelfeld, Martens, Peterfreund, 2022]
- Relates to MSO cost functions [Bourhis, Grez, Jachiet, Riveros, 2021]

Theorem ([Gawrychowski, Manea, Schmid, 2024])

Given an unambiguous cost transducer A and a string T, we can enumerate the results of A on T in order, with preprocessing $O(|T|)$ and output-linear delay (with high probability, or deterministically but with an added logarithmic term)

Open question: What about direct access?

Setting 2: Weight-defined orders (alternative approach)

Alternative approach:

- Along with the input string, give a weight to each assignment $x := \alpha$
- The weight of a mapping is given by combining the weights of its assignments

$\rightarrow$ Pro: Weights are specified in the data, not fixed with the spanner

$\rightarrow$ Con: Weights always combine in the same way (not affected by the spanner)

Setting 2: Weight-defined orders (alternative approach)

Alternative approach:

- Along with the input string, give a weight to each assignment $x := \alpha$
- The weight of a mapping is given by combining the weights of its assignments

$\rightarrow$ Pro: Weights are specified in the data, not fixed with the spanner

$\rightarrow$ Con: Weights always combine in the same way (not affected by the spanner)

The following is known (on trees, in data complexity):

Theorem ([A., Bourhis, Capelli, Monet, 2024])

We can enumerate the mappings of regular spanners on a string T according to subset-monotone ranking functions with preprocessing $O(|T|)$ and delay $O(\log |T|)$

Here subset-monotone means: if $\mu_1 < \mu_2$ and we extend them to μ_1' and μ_2' by assigning the same variable x in the same way, then $\mu_1' < \mu_2'$

Setting 2: Weight-defined orders (alternative approach)

Alternative approach:

- Along with the input string, give a weight to each assignment $x := \alpha$
- The weight of a mapping is given by combining the weights of its assignments

$\rightarrow$ Pro: Weights are specified in the data, not fixed with the spanner

$\rightarrow$ Con: Weights always combine in the same way (not affected by the spanner)

The following is known (on trees, in data complexity):

Theorem ([A., Bourhis, Capelli, Monet, 2024])

We can enumerate the mappings of regular spanners on a string T according to subset-monotone ranking functions with preprocessing $O(|T|)$ and delay $O(\log |T|)$

Here subset-monotone means: if $\mu_1 < \mu_2$ and we extend them to μ_1' and μ_2' by assigning the same variable x in the same way, then $\mu_1' < \mu_2'$

Open questions: Combined complexity? connection to the other approach?

General idea: Factorized representations

How can we make all these query evaluation algorithms more modular?

- Design general-purpose data structures to represent a set of solutions
  → Give efficient algorithms for them, e.g., (ranked) enumeration, counting
- Build such structures to represent the answers of your query evaluation task

General idea: Factorized representations

How can we make all these query evaluation algorithms more modular?

- Design general-purpose data structures to represent a set of solutions
  - → Give efficient algorithms for them, e.g., (ranked) enumeration, counting
- Build such structures to represent the answers of your query evaluation task

In database theory:

- Factorized databases [Olteanu and Schleich, 2016]
- Tractable circuits from knowledge compilation [A. and Capelli, 2024]

→ Connections to formal language theory [Kimelfeld, Martens, Niewerth, 2025]

General idea: Factorized representations

How can we make all these query evaluation algorithms more modular?

- Design general-purpose data structures to represent a set of solutions
  - → Give efficient algorithms for them, e.g., (ranked) enumeration, counting
- Build such structures to represent the answers of your query evaluation task

In database theory:

- Factorized databases [Olteanu and Schleich, 2016]
- Tractable circuits from knowledge compilation [A. and Capelli, 2024]

→ Connections to formal language theory [Kimelfeld, Martens, Niewerth, 2025]

Some works presented so far in fact rely on factorized representations, e.g., Enumerable Compact Sets [Muñoz and Riveros, 2024]

General idea: Factorized representations

How can we make all these query evaluation algorithms more modular?

- Design general-purpose data structures to represent a set of solutions
  - → Give efficient algorithms for them, e.g., (ranked) enumeration, counting
- Build such structures to represent the answers of your query evaluation task

In database theory:

- Factorized databases [Olteanu and Schleich, 2016]
- Tractable circuits from knowledge compilation [A. and Capelli, 2024]

→ Connections to formal language theory [Kimelfeld, Martens, Niewerth, 2025]

Some works presented so far in fact rely on factorized representations, e.g., Enumerable Compact Sets [Muñoz and Riveros, 2024]

Open question: Are there connections between compressed input representations and factorized output representations?

Trees

Motivation

Trees are a common data storage format (XML, JSON)

- How can we efficiently query them? (with XPath, JSONPath, jq, etc.)
- In particular: schema validation

Motivation

Trees are a common data storage format (XML, JSON)

- How can we efficiently query them? (with XPath, JSONPath, jq, etc.)
- In particular: schema validation

The formalism of regular spanners corresponds to word automata and MSO logic

→ Can be extended from words to trees

→ Less pleasant to write (no regex-formulas)

→ No spans: assign variables to nodes, like extractors [Schmid, 2026]

MSO queries on trees (free first-order variables)

- Atoms:
  - $a(x)$: node x has label $a \in \Sigma$
  - $[x \to y]$: node y is a child of node x
- Boolean connectives: $\wedge$, $\vee$, $\neg$, $\Rightarrow$
- First-order quantification: $\forall$, $\exists$
- Quantification over sets: $\forall X$ / $\exists X$
  - X stands for a set of nodes to be used in the formula
- Answers to a query $Q(x_1, \ldots, x_k)$: all node tuples $(n_1, \ldots, n_k)$ satisfying Q

Examples:

- Every node labeled a has a child labeled b $\qquad \forall x\ a(x) \Rightarrow (\exists y\ b(y) \wedge [x \to y])$
- Every node labeled a has a descendant labeled b

  $$\forall x\ a(x) \Rightarrow \forall X \left(X(x) \wedge (\forall yz\ X(y) \wedge [y \to z] \Rightarrow X(z)) \right) \Rightarrow \left(\exists y\ X(y) \wedge b(y) \right)$$

  "for every a-node x, every set X containing x and closed under taking children must contain a b-node y"

Results on trees

- Enumeration:
  - Can be done with linear preprocessing and constant delay **data complexity** and polynomial combined complexity given a (possibly nondeterministic) automaton [A., Bourhis, Mengel, Niewerth, 2019]
    - Substitutions: logarithmic **data complexity**, polynomial combined complexity
    - Insertions/deletions: should extend via [Kleest-Meißner, Marasus, Niewerth, 2025]
    - Dynamic membership in $O(1)$ in some cases [A., Barloy, Jachiet, Paperman, 2025]
- Counting: the FPRAS generalizes to tree automata
  - See [Arenas, Croquevielle, Jayaram, Riveros, 2021a], [Meel and de Colnet, 2026]
- Streaming:
  - For nested documents ($\leftrightarrow$ trees): see [Muñoz and Riveros, 2024]
  - Earliest answering: see [Gienieczko, Muñoz, Murlak, Paperman, 2026]
  - Memory usage: see [Barloy et al., 2021]
- Compression: MSO enumeration for SLPs [Lohrey and Schmid, 2026]
- Ranked enumeration (subset-monotone): [A., Bourhis, Capelli, Monet, 2024]

Conclusion

Database theory and stringology: The data complexity perspective

- **Database theory:** find the best possible **data complexity** for each fixed query
  - → Classify **all queries** in a query class (given by a logic)
  - → Pinpoint the **hardest queries** of each class
  - → End goal: efficient **execution plans**
- **Stringology:** find the **exact complexity** of specific **individual problems**
  - → Find the best **algorithms**
  - → Prove fine-grained complexity **lower bounds**

Database theory and stringology: The data complexity perspective

- **Database theory:** find the best possible **data complexity** for each fixed query
  - → Classify **all queries** in a query class (given by a logic)
  - → Pinpoint the **hardest queries** of each class
  - → End goal: efficient **execution plans**
- **Stringology:** find the **exact complexity** of specific **individual problems**
  - → Find the best **algorithms**
  - → Prove fine-grained complexity **lower bounds**
- **Key challenge:** Queries involving **regular languages**
  - Can we find the best **data complexity** bounds for each individual language?

Some other underexplored areas

- Optimal combined complexity bounds:
  - Tractability in the presence of unambiguity
  - Improving the polynomial exponent
  - Backurs-Indyk-type lower bounds [Backurs and Indyk, 2016]
- Parallel processing: [Doleschal, Kimelfeld, Martens, Nahshon, Neven, 2019]
- Use indexes to speed up evaluation:
  - Not query-specific, e.g., suffix trees / suffix arrays
  - Sketches, for randomized/approximate guarantees
- Using logical assumptions / quantitative information in algorithms

Some other underexplored areas

- Optimal combined complexity bounds:
  - Tractability in the presence of unambiguity
  - Improving the polynomial exponent
  - Backurs-Indyk-type lower bounds [Backurs and Indyk, 2016]
- Parallel processing: [Doleschal, Kimelfeld, Martens, Nahshon, Neven, 2019]
- Use indexes to speed up evaluation:
  - Not query-specific, e.g., suffix trees / suffix arrays
  - Sketches, for randomized/approximate guarantees
- Using logical assumptions / quantitative information in algorithms

Thanks for your attention!

Acknowledgements

Many thanks to Martín Muñoz for help in navigating the related work

The talk also benefited from many existing surveys about document spanners: [Doleschal et al., 2021, Schmid and Schweikardt, 2022a, Schmid, 2024, Schmid et al., 2024]

See also [Fagin et al., 2015] which introduced the formalism

References i

- Amarilli, A., Barloy, C., Jachiet, L., and Paperman, C. (2025).
  Dynamic membership for regular tree languages.
  In MFCS.
- Amarilli, A., Bourhis, P., Capelli, F., and Monet, M. (2024a).
  Ranked enumeration for MSO on trees via knowledge compilation.
  In ICDT.
- Amarilli, A., Bourhis, P., Mengel, S., and Niewerth, M. (2019a).
  Constant-delay enumeration for nondeterministic document spanners.
  In ICDT.

References ii

- Amarilli, A., Bourhis, P., Mengel, S., and Niewerth, M. (2019b).
  Enumeration on trees with tractable combined complexity and efficient updates.
  In PODS.
- Amarilli, A. and Capelli, F. (2024).
  Tractable circuits in database theory.
  In ACM SIGMOD Record.
- Amarilli, A., Dziadek, S., and Segoufin, L. (2026).
  Constant-time dynamic enumeration of word infixes in a regular language.
  In MFCS.

References iii

Amarilli, A., Jachiet, L., Muñoz, M., and Riveros, C. (2022).
Efficient enumeration for annotated grammars.
In PODS.

Amarilli, A., Jachiet, L., and Paperman, C. (2021).
Dynamic membership for regular languages.
In ICALP.

Amarilli, A., Kimelfeld, B., Labbé, S., and Mengel, S. (2024b).
Skyline operators for document spanners.
In ICDT.

References iv

- Arenas, M., Croquevielle, L. A., Jayaram, R., and Riveros, C. (2021a).
  When is approximate counting for conjunctive queries tractable?
  In STOC.
- Arenas, M., Croquevielle, L. A., Jayaram, R., and Riveros, C. (2021b).
  #NFA admits an FPRAS: Efficient enumeration, counting, and uniform generation for logspace classes.
  Journal of the ACM (JACM), 68(6).
- Backurs, A. and Indyk, P. (2016).
  Which regular expression patterns are hard to match?
  In FOCS.

References v

- Barloy, C., Murlak, F., and Paperman, C. (2021).
  Stackless processing of streamed trees.
  In PODS.
- Berkholz, C., Keppeler, J., and Schweikardt, N. (2017).
  Answering conjunctive queries under updates.
  In PODS.
- Bille, P., Gørtz, I. L., and Jessen, R. S. (2026).
  Improved regular expression matching with simple backreferences.
  arXiv preprint arXiv:2609.16914.

References vi

- Bojańczyk, M. (2018).
  Polyregular functions.
  arXiv preprint arXiv:1810.08760.
- Bojańczyk, M. and Lohrey, M. (2026).
  Transducers on compressed strings.
  In ICALP.
- Bossonney, K., Calisto, V., Riveros, C., Toro, G., Van Sint Jan, N., and Vrgoč, D. (2024).
  Demonstrating REmatch: A novel regex engine for finding all matches.
  In SIGMOD Conference Companion.

References vii

- Bourhis, P., Capelli, F., Mengel, S., and Riveros, C. (2025).
  Dynamic direct access of MSO query evaluation over strings.
  In ICDT.
- Bourhis, P., Grez, A., Jachiet, L., and Riveros, C. (2021).
  Ranked enumeration of MSO logic on words.
  In ICDT.
- Breslauer, D. and Galil, Z. (2014).
  Real-time streaming string-matching.
  ACM Transactions on Algorithms (TALG), 10(4).

References viii

- Carmeli, N., Tziavelis, N., Gatterbauer, W., Kimelfeld, B., and Riedewald, M. (2023).
  Tractable orders for direct access to ranked answers of conjunctive queries.
  ACM Transactions on Database Systems, 48(1).
- Doleschal, J., Kimelfeld, B., and Martens, W. (2021).
  Database principles and challenges in text analysis.
  ACM SIGMOD Record, 50(2).
- Doleschal, J., Kimelfeld, B., Martens, W., Nahshon, Y., and Neven, F. (2019).
  Split-correctness in information extraction.
  In PODS.

- Doleschal, J., Kimelfeld, B., Martens, W., and Peterfreund, L. (2022). **Weight annotation in information extraction.** Logical Methods in Computer Science, 18.
- Dudek, B., Gawrychowski, P., Gourdel, G., and Starikovskaya, T. (2022). **Streaming regular expression membership and pattern matching.** In SODA. SIAM.
- Fagin, R., Kimelfeld, B., Reiss, F., and Vansummeren, S. (2015). **Document spanners: A formal approach to information extraction.** JACM, 62(2).

References x

- Florenzano, F., Riveros, C., Ugarte, M., Vansummeren, S., and Vrgoč, D. (2018). **Constant delay algorithms for regular document spanners.** In PODS.
- Freydenberger, D. D. and Peterfreund, L. (2021). **The theory of concatenation over finite models.** In ICALP.
- Ganardi, M., Hucke, D., Lohrey, M., Mamouras, K., and Starikovskaya, T. (2025). **Regular languages in the sliding window model.** TheoretiCS, 4.

- Ganardi, M., Jachiet, L., Lohrey, M., and Schwentick, T. (2022).
  Low-latency sliding window algorithms for formal languages.
  In FSTTCS.
- Gawrychowski, P., Manea, F., Sarnighausen-Cahn, P., and Siemer, S. (2026).
  Optimal enumeration of regular pattern matches.
  Proceedings of the ACM on Management of Data, 4(2).
- Gawrychowski, P., Manea, F., and Schmid, M. L. (2024).
  Revisiting weighted information extraction: a simpler and faster algorithm for ranked enumeration.
  Proceedings of the ACM on Management of Data, 2(5).

- Gienieczko, M., Muñoz, M., Murlak, F., and Paperman, C. (2026).
  Earliest query answering over streamed trees.
  arXiv preprint arXiv:2606.07408.
- Irwin, O. (2022).
  Complexity and algorithms for counting problems.
  Master's thesis.
- Kimelfeld, B., Martens, W., and Niewerth, M. (2025).
  A formal language perspective on factorized representations.
  In ICDT.


Kleest-Meißner, S., Marasus, J., and Niewerth, M. (2025).
MSO queries on trees: Enumerating answers under updates using forest algebras.
Logical Methods in Computer Science, 21.


Kumabe, S. and Uezato, Y. (2026).
On the complexity of the matching problem of regular expressions with backreferences.
In ICALP.


Lohrey, M. (2012).
Algorithmics on SLP-compressed strings: A survey.
Groups-Complexity-Cryptology, 4(2).

References xiv

- Lohrey, M. (2027).
  Ranked MSO-enumeration over compressed words.
  In PODS.
  To appear. arXiv preprint arXiv:2606.03947.
- Lohrey, M. and Schmid, M. L. (2026).
  MSO-enumeration over SLP-compressed unranked forests.
  TheoretiCS, 5.
- Meel, K. S. and de Colnet, A. (2026).
  #CFG and #DNNF admit FPRAS.
  In SODA.

- Muñoz, M. (2026).
  Dynamic direct (ranked) access of MSO query evaluation over SLP-compressed strings.
  arXiv preprint arXiv:2603.13058.
- Muñoz, M. and Riveros, C. (2024).
  Streaming enumeration on nested documents.
  ACM Transactions on Database Systems, 49(4).
- Muñoz, M. and Riveros, C. (2025).
  Constant-delay enumeration for SLP-compressed documents.
  Logical Methods in Computer Science, 21.

- Navarro, G. (2021a).
  Indexing highly repetitive string collections, part I: Repetitiveness measures.
  ACM Computing Surveys, 54(2).
- Navarro, G. (2021b).
  Indexing highly repetitive string collections, part II: Compressed indexes.
  ACM Computing Surveys, 54(2).
- Olteanu, D. and Schleich, M. (2016).
  Factorized databases.
  ACM SIGMOD Record, 45(2).

Peterfreund, L. (2021).
Grammars for document spanners.
In ICDT.

Reidenbach, D. and Schmid, M. L. (2014).
Patterns with bounded treewidth.
Information and Computation, 239.

Schmid, M. L. (2024).
The information extraction framework of document spanners – A very informal survey.
In SOFSEM.

- Schmid, M. L. (2026).
  A general information extraction framework based on formal languages.
  Information and Computation.
- Schmid, M. L. et al. (2024).
  A short note on the mutually beneficial relationship between information extraction in database theory and classical formal language theory.
  Bulletin of EATCS, 143(1).
- Schmid, M. L. and Schweikardt, N. (2021).
  Spanner evaluation over SLP-compressed documents.
  In PODS.

Schmid, M. L. and Schweikardt, N. (2022a).
Document spanners – A brief overview of concepts, results, and recent developments.
In PODS.

Schmid, M. L. and Schweikardt, N. (2022b).
Query evaluation over slp-represented document databases with complex document editing.
In Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems.

References xx

- Schmid, M. L. and Schweikardt, N. (2024).
  Refl-spanners: A purely regular approach to non-regular core spanners.
  Logical Methods in Computer Science, 20.
- Segoufin, L. (2013).
  Enumerating with constant delay the answers to a query.
  In ICDT.
- Tziavelis, N., Gatterbauer, W., and Riedewald, M. (2024).
  Ranked enumeration for database queries.
  ACM SIGMOD Record, 53(3).


Uezato, Y. (2026).
Matching regular-typed pattern languages: Quadratic-time algorithms.
In CPM.