Efficient Enumeration of Query Answers via Circuits

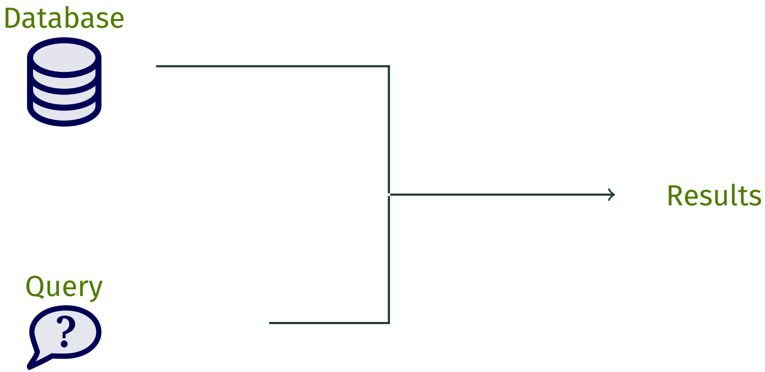
Antoine Amarilli

March 19th, 2024

Télécom Paris

Query evaluation

Central problem in database theory and practice: query evaluation



Query evaluation

Central problem in database theory and practice: query evaluation

Database

Follows

from	to
Alice	Bob
Bob	Carol
Bob	Dave
Carol	Eve

Query

→ Results

Query evaluation

Central problem in database theory and practice: query evaluation

Database

Follows

from	to
Alice	Bob
Bob	Carol
Bob	Dave
Carol	Eve

Query

"Find all pairs of users ***x*** *and* ***y*** *such that* ***x*** *follows someone who follows* ***z****"*

→ **Results**

Query evaluation

Central problem in database theory and practice: query evaluation

Database

Follows

from	**to**
Alice	Bob
Bob	Carol
Bob	Dave
Carol	Eve

→ **Results**

Query

"Find all pairs of users x *and* z *such that* x *follows someone who follows* z*"*

$$Q(x, z) : \exists y\ F(x, y) \land F(y, z)$$

Query evaluation

Central problem in database theory and practice: **query evaluation**

Database

Follows

from	to
Alice	Bob
Bob	Carol
Bob	Dave
Carol	Eve

Query

*"Find all pairs of users **x** and **z** such that **x** follows someone who follows **z**"*

$Q(x, z) : \exists y \; F(x, y) \wedge F(y, z)$

Results

x	z
Alice	Carol
Alice	Dave
Bob	Eve

Combined complexity and data complexity

How to study the computational complexity of query evaluation?

Combined complexity and data complexity

How to study the **computational complexity** of query evaluation?

- **Problem** of query evaluation (QE)

Combined complexity and data complexity

How to study the **computational complexity** of query evaluation?

- **Problem** of query evaluation (QE)
  - **Input 1:** query Q
  - **Input 2:** database D

Combined complexity and data complexity

How to study the **computational complexity** of query evaluation?

- **Problem** of query evaluation (QE)
  - **Input 1:** query Q
  - **Input 2:** database D
  - **Output:** result $Q(D)$

Combined complexity and data complexity

How to study the computational complexity of query evaluation?

- Problem of query evaluation (QE)
  - Input 1: query Q
  - Input 2: database D
  - Output: result $Q(D)$

Two ways to measure complexity:

Combined complexity and data complexity

How to study the computational complexity of query evaluation?

- Problem of query evaluation (QE)
  - Input 1: query Q
  - Input 2: database D
  - Output: result $Q(D)$

Two ways to measure complexity:

- Combined complexity: the query and database are given as input

Combined complexity and data complexity

How to study the computational complexity of query evaluation?

- Problem of query evaluation (QE)
  - Input 1: query Q
  - Input 2: database D
  - Output: result $Q(D)$

Two ways to measure complexity:

- Combined complexity: the query and database are given as input
- Data complexity: the query is fixed, the input is only the data

Combined complexity and data complexity

How to study the computational complexity of query evaluation?

- Problem of query evaluation for Q ($\mathrm{QE}(Q)$)
  - ~~Input 1: query Q~~
  - Input: database D
  - Output: result $Q(D)$

Two ways to measure complexity:

- Combined complexity: the query and database are given as input
- Data complexity: the query is fixed, the input is only the data

Combined complexity and data complexity

How to study the computational complexity of query evaluation?

- Problem of query evaluation for Q (QE(Q))
  - ~~Input 1: query Q~~
  - Input: database D
  - Output: result $Q(D)$

Two ways to measure complexity:

- Combined complexity: the query and database are given as input
- Data complexity: the query is fixed, the input is only the data

  $\rightarrow$ Motivation: the data is usually much larger than the query

Data complexity for large output size

- Consider the query Q: *"Find all users* x*,* y*, and* z *such that* x *follows* y *and* y *follows* z*"*

$$Q(x, y, z) : F(x, y) \wedge F(y, z)$$

Data complexity for large output size

- Consider the query Q: *"Find all users* x*,* y*, and* z *such that* x *follows* y *and* y *follows* z*"*
  $$Q(x, y, z) : F(x, y) \wedge F(y, z)$$
- Assume the input database D contains n "follows" facts $\quad |D| = n$

Data complexity for large output size

- Consider the query Q: *"Find all users* x*,* y*, and* z *such that* x *follows* y *and* y *follows* z*"*

$$Q(x, y, z) : F(x, y) \wedge F(y, z)$$

- Assume the input database D contains n "follows" facts $\quad |D| = n$
- What is the data complexity of Q as a function of n?

Data complexity for large output size

- Consider the query Q: *"Find all users* x, y, *and* z *such that* x *follows* y *and* y *follows* z*"* $Q(x, y, z) : F(x, y) \wedge F(y, z)$
- Assume the input database D contains **n "follows" facts** $|D| = n$
- What is the **data complexity** of Q as a function of n?
    - **Trivial algorithm:** check every pair always $\Theta(n^2)$

Data complexity for large output size

- Consider the query Q: *"Find all users* x, y, *and* z *such that* x *follows* y *and* y *follows* z*"* $Q(x, y, z) : F(x, y) \wedge F(y, z)$
- Assume the input database D contains n "follows" facts $|D| = n$
- What is the **data complexity** of Q as a function of n?
  - **Trivial algorithm:** check every pair always $\Theta(n^2)$
  - **Better algorithm:**
    - Check which y have a follower x and followee z

Data complexity for large output size

- Consider the query Q: *"Find all users* x, y, *and* z *such that* x *follows* y *and* y *follows* z*"*
  $$Q(x, y, z) : F(x, y) \wedge F(y, z)$$
- Assume the input database D contains n "follows" facts $\quad |D| = n$
- What is the data complexity of Q as a function of n?
  - Trivial algorithm: check every pair $\quad$ always $\Theta(n^2)$
  - Better algorithm:
    - Check which y have a follower x and followee z
    - For each such y, output all matching pairs of x and z

Data complexity for large output size

- Consider the query Q: *"Find all users* x*,* y*, and* z *such that* x *follows* y *and* y *follows* z*"* $Q(x, y, z) : F(x, y) \wedge F(y, z)$
- Assume the input database D contains n **"follows" facts** $|D| = n$
- What is the **data complexity** of Q as a function of n?
  - **Trivial algorithm:** check every pair — always $\Theta(n^2)$
  - **Better algorithm:** — also $\Theta(n^2)$!
    - Check which y have a follower x and followee z
    - For each such y, output all matching pairs of x and z
  - **Problem:** we can't beat the **result size** which is $\Omega(n^2)$ in general

Data complexity for large output size

- Consider the query Q: *"Find all users x, y, and z such that x follows y and y follows z"* $Q(x, y, z) : F(x, y) \wedge F(y, z)$
- Assume the input database D contains n **"follows" facts** $|D| = n$
- What is the **data complexity** of Q as a function of n?
  - **Trivial algorithm:** check every pair — always $\Theta(n^2)$
  - **Better algorithm:** — also $\Theta(n^2)$!
    - Check which y have a follower x and followee z
    - For each such y, output all matching pairs of x and z
  - **Problem:** we can't beat the **result size** which is $\Omega(n^2)$ in general

→ **In which sense is the second algorithm preferable?**

Data complexity for large output size

- Consider the query Q: *"Find all users* x*,* y*, and* z *such that* x *follows* y *and* y *follows* z*"*
  $$Q(x, y, z) : F(x, y) \wedge F(y, z)$$
- Assume the input database D contains **n "follows" facts** $|D| = n$
- What is the **data complexity** of Q as a function of n?
  - **Trivial algorithm:** check every pair **always $\Theta(n^2)$**
  - **Better algorithm:** **also $\Theta(n^2)$!**
    - Check which y have a follower x and followee z
    - For each such y, output all matching pairs of x and z
  - **Problem:** we can't beat the **result size** which is $\Omega(n^2)$ in general

→ **In which sense is the second algorithm preferable?**

→ We need a **better measure of complexity**

Enumeration algorithms

How to measure the running time of algorithms producing a large collection of answers?

Enumeration algorithms

How to measure the running time of algorithms producing a large collection of answers?

- Idea 1: make the complexity depend on the result size

Enumeration algorithms

How to measure the running time of algorithms producing a large collection of answers?

- Idea 1: make the complexity depend on the result size
- Idea 2: make the algorithm produce results in streaming

Enumeration algorithms

How to measure the running time of algorithms producing a large collection of answers?

- Idea 1: make the complexity depend on the result size
- Idea 2: make the algorithm produce results in streaming

Database D

Enumeration algorithms

How to measure the **running time** of algorithms producing a **large collection** of answers?

- Idea 1: make the complexity depend on the **result size**
- Idea 2: make the algorithm produce results **in streaming**

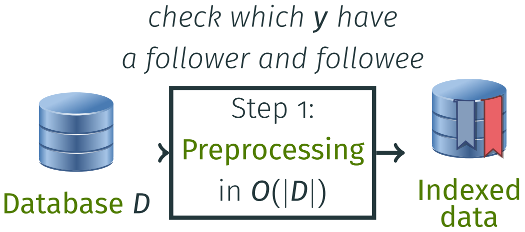
check which $\boldsymbol{y}$ *have*
a follower and followee

Step 1:
Preprocessing
in $O(|D|)$

Enumeration algorithms

How to measure the **running time** of algorithms producing a **large collection** of answers?

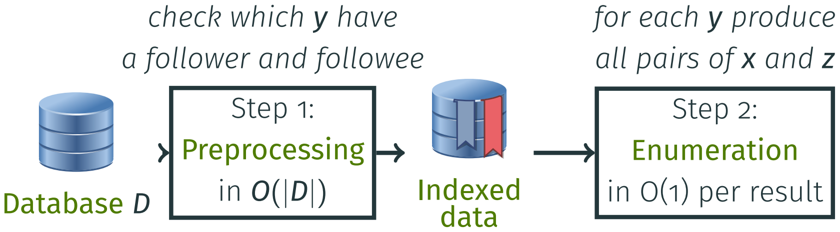
- Idea 1: make the complexity depend on the **result size**
- Idea 2: make the algorithm produce results **in streaming**



Enumeration algorithms

How to measure the **running time** of algorithms producing a **large collection** of answers?

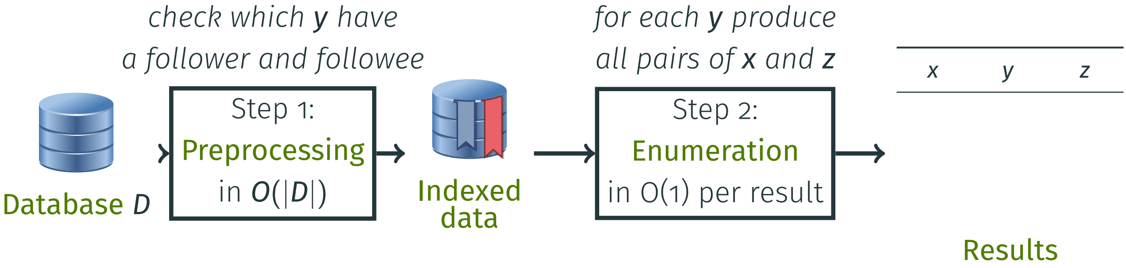
- Idea 1: make the complexity depend on the **result size**
- Idea 2: make the algorithm produce results **in streaming**



Enumeration algorithms

How to measure the **running time** of algorithms producing a **large collection** of answers?

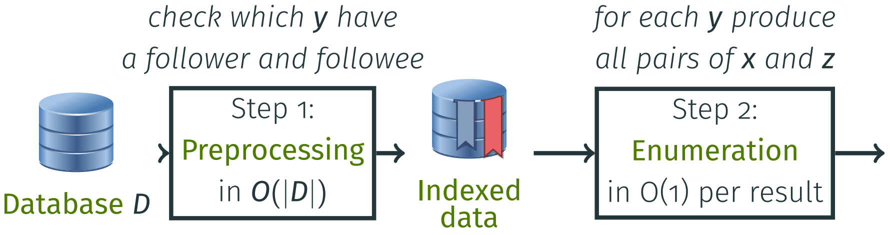
- Idea 1: make the complexity depend on the **result size**
- Idea 2: make the algorithm produce results **in streaming**



Enumeration algorithms

How to measure the **running time** of algorithms producing a **large collection** of answers?

- Idea 1: make the complexity depend on the **result size**
- Idea 2: make the algorithm produce results **in streaming**

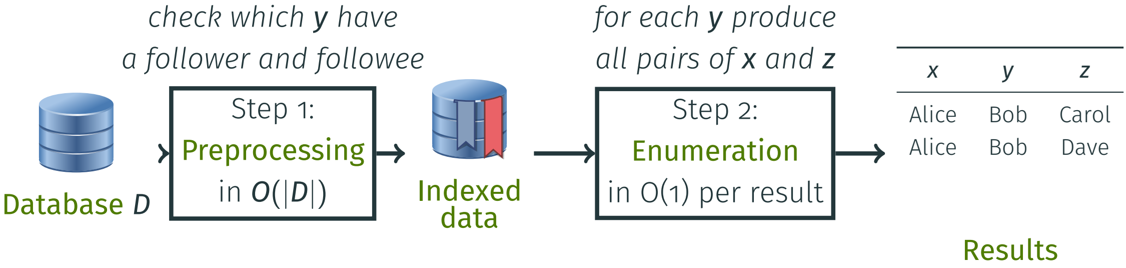


x	y	z
Alice	Bob	Carol

Enumeration algorithms

How to measure the **running time** of algorithms producing a **large collection** of answers?

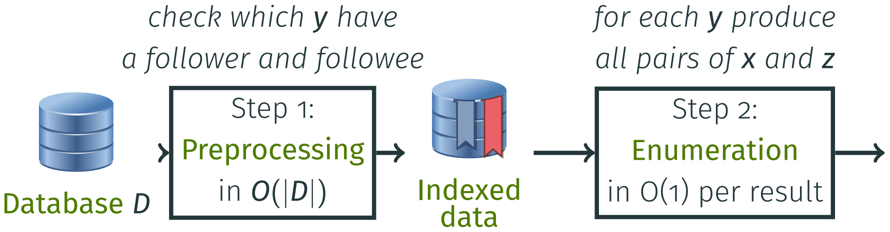
- Idea 1: make the complexity depend on the **result size**
- Idea 2: make the algorithm produce results **in streaming**



Enumeration algorithms

How to measure the **running time** of algorithms producing a **large collection** of answers?

- Idea 1: make the complexity depend on the **result size**
- Idea 2: make the algorithm produce results **in streaming**



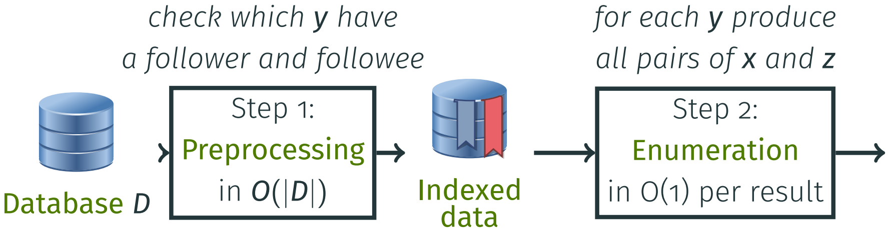
x	y	z
Alice	Bob	Carol
Alice	Bob	Dave
Bob	Carol	Eve

Results

Enumeration algorithms

How to measure the **running time** of algorithms producing a **large collection** of answers?

- Idea 1: make the complexity depend on the **result size**
- Idea 2: make the algorithm produce results **in streaming**



x	y	z
Alice	Bob	Carol
Alice	Bob	Dave
Bob	Carol	Eve

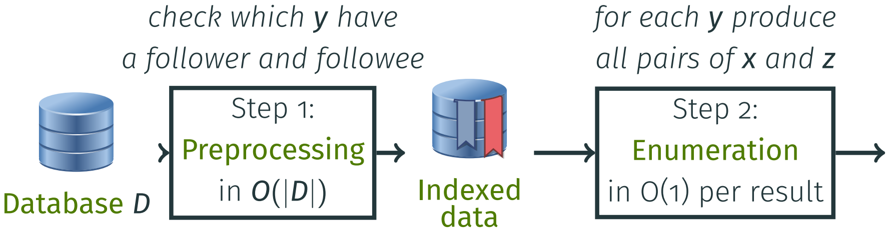
Results

→ Tests **if there is an answer** in time $O(|D|)$

Enumeration algorithms

How to measure the running time of algorithms producing a large collection of answers?

- Idea 1: make the complexity depend on the result size
- Idea 2: make the algorithm produce results in streaming



x	y	z
Alice	Bob	Carol
Alice	Bob	Dave
Bob	Carol	Eve

Results

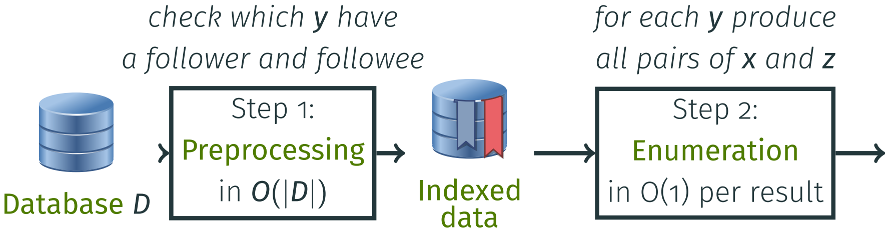
$\rightarrow$ Tests if there is an answer in time $O(|D|)$

$\rightarrow$ Computes the first k answers in time $O(|D| + k)$

Enumeration algorithms

How to measure the **running time** of algorithms producing a **large collection** of answers?

- Idea 1: make the complexity depend on the **result size**
- Idea 2: make the algorithm produce results **in streaming**



x	y	z
Alice	Bob	Carol
Alice	Bob	Dave
Bob	Carol	Eve

Results

→ Tests **if there is an answer** in time $O(|D|)$

→ Computes the **first k answers** in time $O(|D| + k)$

Idea: Factorized representations (aka circuits)

- During preprocessing, compute a factorized representation of the answers
- During enumeration, decompress this factorized representation

Idea: Factorized representations (aka circuits)

- During preprocessing, compute a factorized representation of the answers
- During enumeration, decompress this factorized representation

$$Q(x, y, z) : F(x, y) \wedge F(y, z)$$

Idea: Factorized representations (aka circuits)

- During preprocessing, compute a factorized representation of the answers
- During enumeration, decompress this factorized representation

$$Q(x, y, z) : F(x, y) \wedge F(y, z)$$

Database D

Follows

from	**to**
Alice	Bob
Bob	Carol
Bob	Dave
Carol	Eve

Idea: Factorized representations (aka circuits)

- During preprocessing, compute a **factorized representation** of the answers
- During enumeration, **decompress** this factorized representation

$$Q(x, y, z) : F(x, y) \wedge F(y, z)$$

Database D

Follows

from	**to**
Alice	Bob
Bob	Carol
Bob	Dave
Carol	Eve

Output $Q(D)$

x	y	z
Alice	Bob	Carol
Alice	Bob	Dave
Bob	Carol	Eve

Idea: Factorized representations (aka circuits)

- During preprocessing, compute a **factorized representation** of the answers
- During enumeration, **decompress** this factorized representation

$$Q(x, y, z) : F(x, y) \land F(y, z)$$

Database D

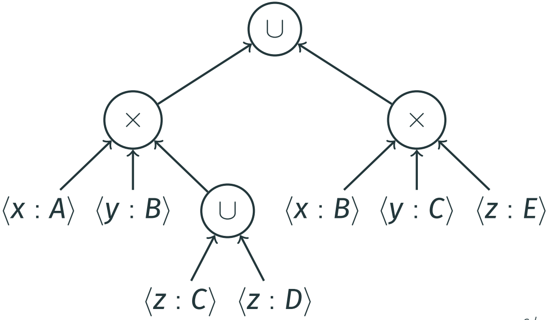
Follows

from	to
Alice	Bob
Bob	Carol
Bob	Dave
Carol	Eve

Output $Q(D)$

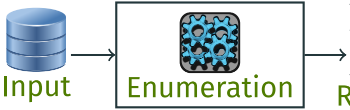
x	y	z
Alice	Bob	Carol
Alice	Bob	Dave
Bob	Carol	Eve

Factorized representation of $Q(D)$



Advantage of factorized representations: Modularity

WITHOUT factorized representations:

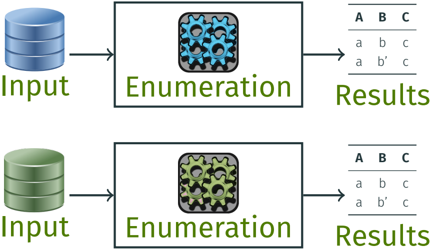


A	B	C
a	b	c
a	b’	c

Results

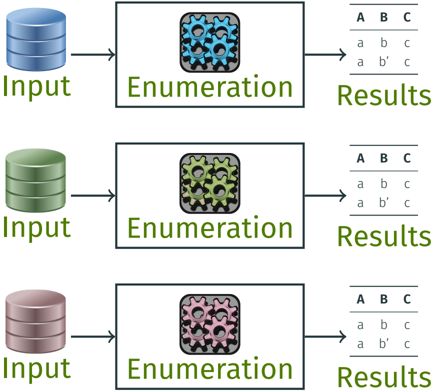
Advantage of factorized representations: Modularity

WITHOUT factorized representations:



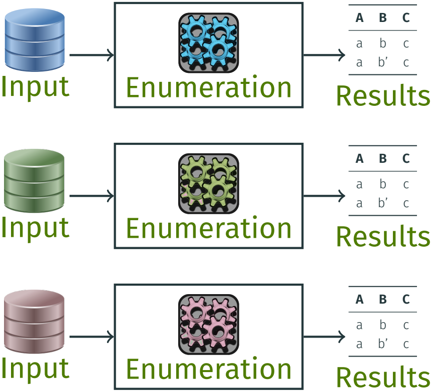
Advantage of factorized representations: Modularity

WITHOUT factorized representations:



Advantage of factorized representations: Modularity

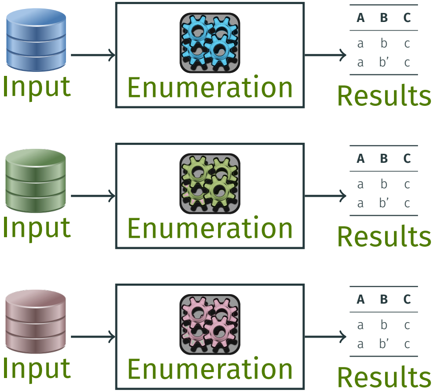
WITHOUT factorized representations:



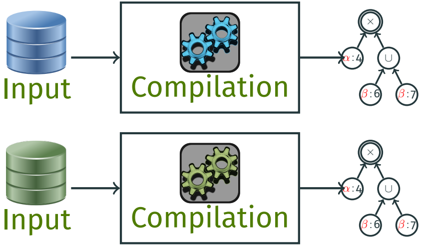
WITH factorized representations:

Advantage of factorized representations: Modularity

WITHOUT factorized representations:

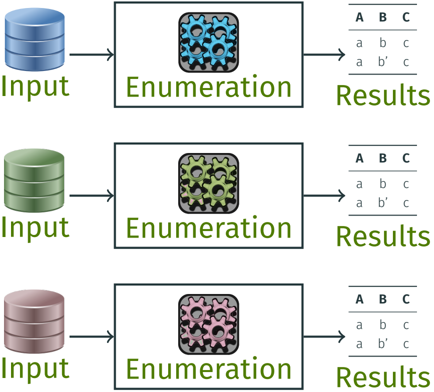


WITH factorized representations:

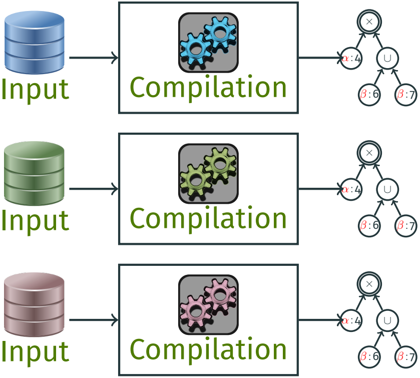


Advantage of factorized representations: Modularity

WITHOUT factorized representations:

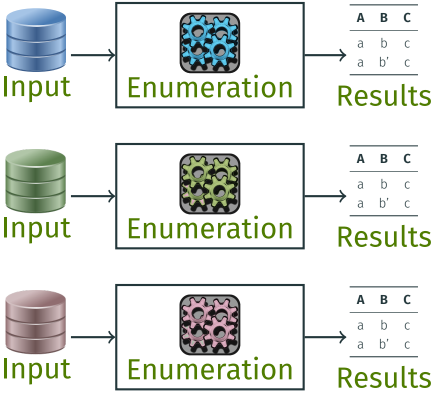


WITH factorized representations:

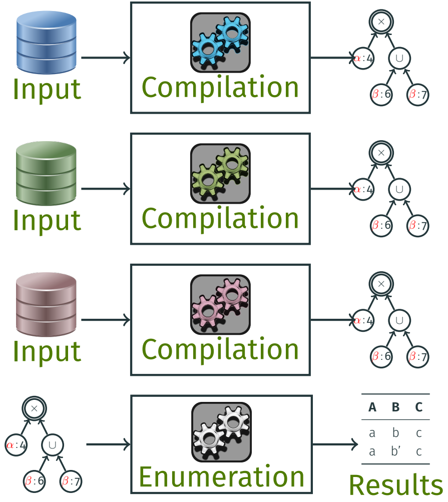


Advantage of factorized representations: Modularity

WITHOUT factorized representations:



WITH factorized representations:



Roadmap of the talk

Results on enumeration for query evaluation, especially via factorized representations

Roadmap of the talk

Results on enumeration for query evaluation, especially via factorized representations

- Conjunctive queries (CQs) and extensions:
  - Yannakakis's algorithm for acyclic and free-connex conjunctive queries
  - Lower bounds for non-free-connex conjunctive queries without self-joins
  - Extensions: CQs with self joins, unions of CQs

Roadmap of the talk

Results on **enumeration** for **query evaluation**, especially via **factorized representations**

- **Conjunctive queries** (CQs) and extensions:
  - **Yannakakis's algorithm** for acyclic and free-connex conjunctive queries
  - **Lower bounds** for non-free-connex conjunctive queries without self-joins
  - **Extensions**: CQs with self joins, unions of CQs
- **Other settings**: Queries defined by automata / monadic second-order logic
  - Efficient enumeration on **trees**
  - Efficient enumeration on **text**
  - Other settings

Roadmap of the talk

Results on **enumeration** for **query evaluation**, especially via **factorized representations**

- **Conjunctive queries** (CQs) and extensions:
  - **Yannakakis's algorithm** for acyclic and free-connex conjunctive queries
  - **Lower bounds** for non-free-connex conjunctive queries without self-joins
  - **Extensions**: CQs with self joins, unions of CQs
- **Other settings**: Queries defined by automata / monadic second-order logic
  - Efficient enumeration on **trees**
  - Efficient enumeration on **text**
  - Other settings
- **Other tasks**: ranked enumeration, direct access, incremental maintenance, etc.

Table of contents

Conjunctive query basics

- Fix the **relation names** (the database tables) and their **arity** (number of columns)

  e.g., Follows (arity-2), Subscribed (arity-2)

Conjunctive query basics

- Fix the **relation names** (the database tables) and their **arity** (number of columns)

  e.g., Follows (arity-2), Subscribed (arity-2)

- A **full conjunctive query** (CQ) is a **conjunction of atoms**

$$Q_1(x, y) : \text{Follows}(x, y)$$

$$Q_2(x, y, z) : \text{Follows}(x, y) \wedge \text{Subscribed}(y, z)$$

Conjunctive query basics

- Fix the **relation names** (the database tables) and their **arity** (number of columns)

  e.g., Follows (arity-2), Subscribed (arity-2)

- A **full conjunctive query** (CQ) is a **conjunction of atoms**

$$Q_1(x, y) : \text{Follows}(x, y)$$

$$Q_2(x, y, z) : \text{Follows}(x, y) \wedge \text{Subscribed}(y, z)$$

- The **answers** of a CQ $Q(x_1, \ldots, x_n)$ on a database D are the tuples of domain elements $(a_1, \ldots, a_n)$ such that the corresponding facts exist in the database

Conjunctive query basics

- Fix the **relation names** (the database tables) and their **arity** (number of columns)

  e.g., Follows (arity-2), Subscribed (arity-2)

- A **full conjunctive query** (CQ) is a **conjunction of atoms**

$$Q_1(x, y) : \text{Follows}(x, y)$$

$$Q_2(x, y, z) : \text{Follows}(x, y) \wedge \text{Subscribed}(y, z)$$

- The **answers** of a CQ $Q(x_1, \ldots, x_n)$ on a database D are the tuples of domain elements $(a_1, \ldots, a_n)$ such that the corresponding facts exist in the database

Follows

a	b
a	b'
a'	b'
a''	b''

Subscribed

b	c
b	c'
b'	c'

- Query $Q_2(x, y, z) : \text{Follows}(x, y) \wedge \text{Subscribed}(y, z)$

Conjunctive query basics

- Fix the relation names (the database tables) and their arity (number of columns)

  e.g., Follows (arity-2), Subscribed (arity-2)

- A full conjunctive query (CQ) is a conjunction of atoms

$$Q_1(x, y) : \text{Follows}(x, y)$$

$$Q_2(x, y, z) : \text{Follows}(x, y) \wedge \text{Subscribed}(y, z)$$

- The answers of a CQ $Q(x_1, \ldots, x_n)$ on a database D are the tuples of domain elements $(a_1, \ldots, a_n)$ such that the corresponding facts exist in the database

Follows

a	b
a	b'
a'	b'
a''	b''

Subscribed

b	c
b	c'
b'	c'

- Query $Q_2(x, y, z) : \text{Follows}(x, y) \wedge \text{Subscribed}(y, z)$
- Database D on the left

Conjunctive query basics

- Fix the **relation names** (the database tables) and their **arity** (number of columns)

  e.g., Follows (arity-2), Subscribed (arity-2)

- A **full conjunctive query** (CQ) is a **conjunction of atoms**

$$Q_1(x, y) : \text{Follows}(x, y)$$

$$Q_2(x, y, z) : \text{Follows}(x, y) \wedge \text{Subscribed}(y, z)$$

- The **answers** of a CQ $Q(x_1, \ldots, x_n)$ on a database D are the tuples of domain elements $(a_1, \ldots, a_n)$ such that the corresponding facts exist in the database

Follows

a	b
a	b'
a'	b'
a''	b''

Subscribed

b	c
b	c'
b'	c'

- Query $Q_2(x, y, z) : \text{Follows}(x, y) \wedge \text{Subscribed}(y, z)$
- Database D on the left
- There are **four answers**:

  $(a, b, c), (a, b, c'), (a, b', c'), (a', b', c')$

Cyclic vs acyclic CQs

Assuming that all relations are arity-2, let's distinguish acyclic CQs and cyclic CQs

Cyclic vs acyclic CQs

Assuming that all relations are arity-2, let's distinguish acyclic CQs and cyclic CQs

Acyclic CQs: the Gaifman graph is acyclic

Cyclic vs acyclic CQs

Assuming that all relations are **arity-2**, let's distinguish **acyclic CQs** and **cyclic CQs**

Acyclic CQs: the Gaifman graph is acyclic

$Q_1(x, y, z) : F(x, y), S(y, z)$

Cyclic vs acyclic CQs

Assuming that all relations are **arity-2**, let's distinguish **acyclic CQs** and **cyclic CQs**

Acyclic CQs: the Gaifman graph is acyclic

$Q_1(x, y, z) : F(x, y), S(y, z)$

$$x \longrightarrow y \longrightarrow z \qquad\qquad x \text{ --- } y \text{ --- } z$$

Cyclic vs acyclic CQs

Assuming that all relations are **arity-2**, let's distinguish **acyclic CQs** and **cyclic CQs**

Acyclic CQs: the Gaifman graph is acyclic

$Q_1(x, y, z) : F(x, y), S(y, z)$

$$x \longrightarrow y \longrightarrow z \qquad x —— y —— z$$

$Q_2(x, y) : F(x, x), S(x, y), F(y, x)$

Cyclic vs acyclic CQs

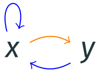
Assuming that all relations are **arity-2**, let's distinguish **acyclic CQs** and **cyclic CQs**

Acyclic CQs: the Gaifman graph is acyclic

$Q_1(x, y, z) : F(x, y), S(y, z)$

$$x \longrightarrow y \longrightarrow z \qquad\qquad x \;—\; y \;—\; z$$

$Q_2(x, y) : F(x, x), S(x, y), F(y, x)$



$$x \;—\; y$$

Cyclic vs acyclic CQs

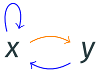
Assuming that all relations are **arity-2**, let's distinguish **acyclic CQs** and **cyclic CQs**

Acyclic CQs: the Gaifman graph is acyclic

$Q_1(x, y, z) : F(x, y), S(y, z)$

$$x \longrightarrow y \longrightarrow z \qquad\qquad x \text{ --- } y \text{ --- } z$$

$Q_2(x, y) : F(x, x), S(x, y), F(y, x)$



$$x \text{ --- } y$$

Cyclic CQs:

$Q_3(x, z) : F(x, y), F(y, z), F(z, x)$

Cyclic vs acyclic CQs

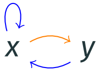
Assuming that all relations are **arity-2**, let's distinguish **acyclic CQs** and **cyclic CQs**

Acyclic CQs: the Gaifman graph is acyclic

$Q_1(x, y, z) : F(x, y), S(y, z)$

$$x \longrightarrow y \longrightarrow z \qquad\qquad x \text{ --- } y \text{ --- } z$$

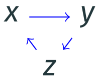
$Q_2(x, y) : F(x, x), S(x, y), F(y, x)$

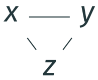


$$x \text{ --- } y$$

Cyclic CQs:

$Q_3(x, z) : F(x, y), F(y, z), F(z, x)$





Cyclic vs acyclic CQs

Assuming that all relations are **arity-2**, let's distinguish **acyclic CQs** and **cyclic CQs**

Acyclic CQs: the Gaifman graph is acyclic

$Q_1(x, y, z) : F(x, y), S(y, z)$

$Q_2(x, y) : F(x, x), S(x, y), F(y, x)$

Cyclic CQs:

$Q_3(x, z) : F(x, y), F(y, z), F(z, x)$

Intuition: the **cyclic** queries seem **harder** (e.g., searching for a triangle in an input directed graph)

Cyclic vs acyclic CQs

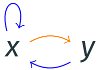
Assuming that all relations are **arity-2**, let's distinguish **acyclic CQs** and **cyclic CQs**

Acyclic CQs: the Gaifman graph is acyclic

$Q_1(x, y, z) : F(x, y), S(y, z)$

$$x \longrightarrow y \longrightarrow z \qquad\qquad x \;—\; y \;—\; z$$

$Q_2(x, y) : F(x, x), S(x, y), F(y, x)$



$$x \;—\; y$$

Cyclic CQs:

$Q_3(x, z) : F(x, y), F(y, z), F(z, x)$

Intuition: the **cyclic** queries seem **harder** (e.g., searching for a triangle in an input directed graph)

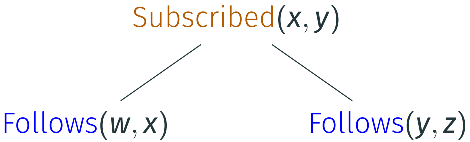
We can generalize **acyclic CQs** to arbitrary arity (= α-acyclic Gaifman hypergraph)

Join trees for acyclic CQs

Fact: a CQ is **acyclic** iff it has a **join tree**:

- The vertices are the **atoms** of the query
- For each variable, its occurrences form a **connected subtree**
- (For experts: width-1 generalized hypertree decomposition of the Gaifman hypergraph)

Take the query: $Q(w, x, y, z) : \text{Follows}(w, x) \wedge \text{Subscribed}(x, y) \wedge \text{Follows}(y, z)$



Yannakakis's algorithm for acyclic CQs

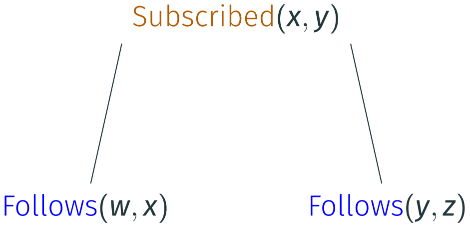
Theorem ([Yannakakis, 1981])

Given an acyclic CQ Q and database D, we can compute $Q(D)$ in time $O(|Q| \times (|D| + m))$, where m is the output size

Yannakakis's algorithm for acyclic CQs

Theorem ([Yannakakis, 1981])

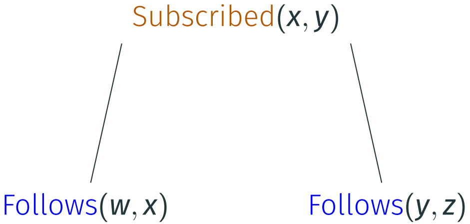
Given an acyclic CQ Q and database D, we can compute $Q(D)$ in time $O(|Q| \times (|D| + m))$, where m is the output size



Yannakakis's algorithm for acyclic CQs

Theorem ([Yannakakis, 1981])

Given an acyclic CQ Q *and database* D*, we can compute* $Q(D)$ *in time* $O(|Q| \times (|D| + m))$*, where* m *is the output size*



- On every node n, write a copy R_n of the relation of the corresponding atom

Yannakakis's algorithm for acyclic CQs

Theorem ([Yannakakis, 1981])

Given an acyclic CQ Q and database D, we can compute $Q(D)$ in time $O(|Q| \times (|D| + m))$, where m is the output size

Subscribed(x, y)

x	y
b	a
b	a'
b	a''
b'	a'

Follows(w, x)

w	x
a	b
a	b''
a''	b'

Follows(y, z)

y	z
a	b
a	b''
a''	b'

- On every node n, write a copy R_n of the relation of the corresponding atom

Yannakakis's algorithm for acyclic CQs

Theorem ([Yannakakis, 1981])

Given an acyclic CQ Q and database D, we can compute $Q(D)$ in time $O(|Q| \times (|D| + m))$, where m is the output size

Subscribed(x, y)

x	y
b	a
b	a'
b	a''
b'	a'

Follows(w, x)

w	x
a	b
a	b''
a''	b'

Follows(y, z)

y	z
a	b
a	b''
a''	b'

- On every node n, write a **copy** R_n of the relation of the corresponding atom
- Do **semijoins** on the tree **bottom-up**:
  - → On every node n, for each child n', keep only the tuples of R_n that have a match in $R_{n'}$

Yannakakis's algorithm for acyclic CQs

Theorem ([Yannakakis, 1981])

Given an acyclic CQ $\boldsymbol{Q}$ *and database* $\boldsymbol{D}$*, we can compute* $\boldsymbol{Q}(\boldsymbol{D})$ *in time* $\boldsymbol{O}(|\boldsymbol{Q}| \times (|\boldsymbol{D}| + \boldsymbol{m}))$*, where* $\boldsymbol{m}$ *is the output size*

Subscribed(x, y)

x	y
b	a
~~b~~	~~a'~~
b	a''
~~b'~~	~~a'~~

Follows(w, x)

w	x
a	b
a	b''
a''	b'

Follows(y, z)

y	z
a	b
a	b''
a''	b'

- On every node $\boldsymbol{n}$, write a **copy** $\boldsymbol{R_n}$ of the relation of the corresponding atom
- Do **semijoins** on the tree **bottom-up**:
  - → On every node $\boldsymbol{n}$, for each child $\boldsymbol{n'}$, keep only the tuples of $\boldsymbol{R_n}$ that have a match in $\boldsymbol{R_{n'}}$

Yannakakis's algorithm for acyclic CQs

Theorem ([Yannakakis, 1981])

Given an acyclic CQ Q *and database* D*, we can compute* $Q(D)$ *in time* $O(|Q| \times (|D| + m))$*, where* m *is the output size*

Subscribed(x, y)

x	y
b	a
~~b~~	~~a'~~
b	a''
~~b'~~	~~a'~~

Follows(w, x)

w	x
a	b
a	b''
a''	b'

Follows(y, z)

y	z
a	b
a	b''
a''	b'

- On every node n, write a **copy** R_n of the relation of the corresponding atom
- Do **semijoins** on the tree **bottom-up**:
  - → On every node n, for each child n', keep only the tuples of R_n that have a match in $R_{n'}$
- Do **semijoins** on the tree **top-down**

Yannakakis's algorithm for acyclic CQs

Theorem ([Yannakakis, 1981])

Given an acyclic CQ Q and database D, we can compute $Q(D)$ in time $O(|Q| \times (|D| + m))$, where m is the output size

Subscribed(x, y)

x	y
b	a
~~b~~	~~a'~~
b	a''
~~b'~~	~~a'~~

Follows(w, x)

w	x
a	b
~~a~~	~~b''~~
~~a''~~	~~b'~~

Follows(y, z)

y	z
a	b
a	b''
a''	b'

- On every node n, write a **copy** R_n of the relation of the corresponding atom
- Do **semijoins** on the tree **bottom-up**:
  - → On every node n, for each child n', keep only the tuples of R_n that have a match in $R_{n'}$
- Do **semijoins** on the tree **top-down**

Yannakakis's algorithm for acyclic CQs

Theorem ([Yannakakis, 1981])

Given an acyclic CQ Q and database D, we can compute $Q(D)$ in time $O(|Q| \times (|D| + m))$, where m is the output size

Subscribed(x, y)

x	y
b	a
~~b~~	~~a'~~
b	a''
~~b'~~	~~a'~~

Follows(w, x)

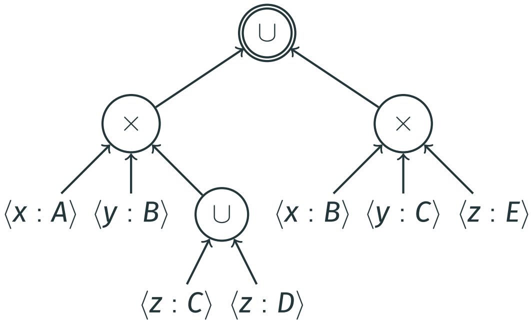
w	x
a	b
~~a~~	~~b''~~
~~a''~~	~~b'~~

Follows(y, z)

y	z
a	b
a	b''
a''	b'

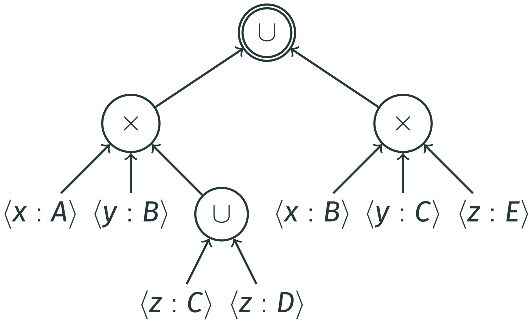
- On every node n, write a **copy** R_n of the relation of the corresponding atom
- Do **semijoins** on the tree **bottom-up**:
  - → On every node n, for each child n', keep only the tuples of R_n that have a match in $R_{n'}$
- Do **semijoins** on the tree **top-down**
- Join together all relations to get the full result

Factorized representations [Olteanu and Závodný, 2015]



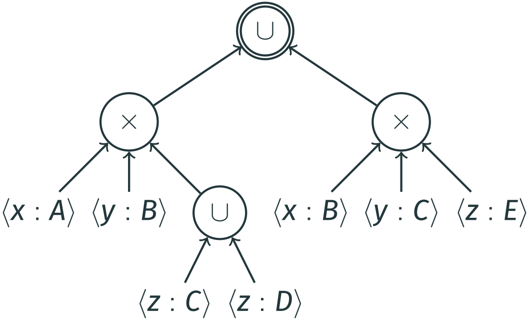
- Directed acyclic graph of **gates**
- **Output** gate: 

Factorized representations [Olteanu and Závodnỳ, 2015]



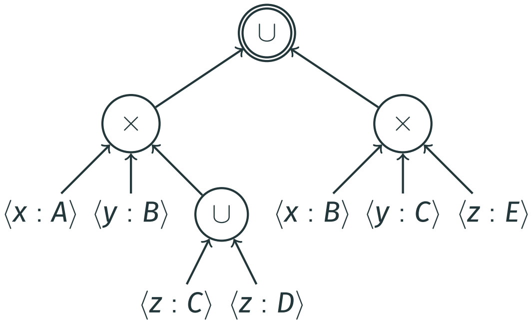
- Directed acyclic graph of **gates**
- **Output** gate: 
- **Variable** gates $\langle x : a \rangle$: represent a **single-tuple** and **single-column** relation

Factorized representations [Olteanu and Závodnỳ, 2015]



- Directed acyclic graph of **gates**
- **Output** gate: 
- **Variable** gates $\langle x : a \rangle$: represent a **single-tuple** and **single-column** relation
- **Relational product** gates: 
  (input domains are disjoint)

Factorized representations [Olteanu and Závodný, 2015]

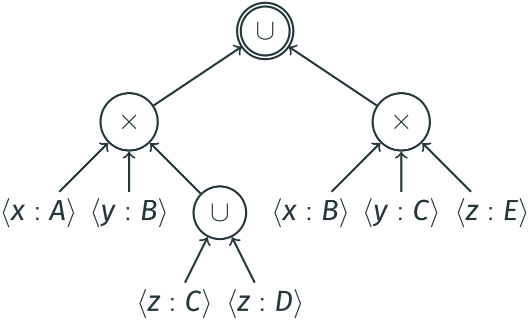


- Directed acyclic graph of **gates**
- **Output** gate: 
- **Variable** gates $\langle x : a \rangle$: represent a **single-tuple** and **single-column** relation
- **Relational product** gates: 

  (input domains are disjoint)
- **Union** gates: 

  (inputs have same domains)

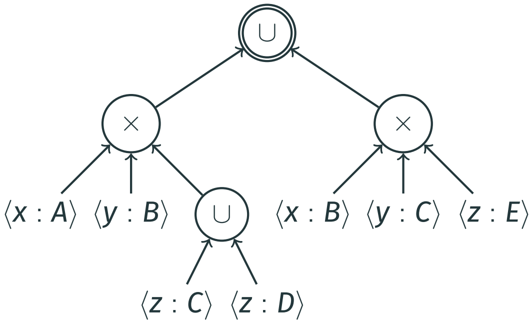
Factorized representations [Olteanu and Závodnỳ, 2015]



x	y	z
A	B	C
A	B	D
B	C	E

- Directed acyclic graph of **gates**
- **Output** gate: 
- **Variable** gates $\langle x : a \rangle$: represent a **single-tuple** and **single-column** relation
- **Relational product** gates: 
  (input domains are disjoint)
- **Union** gates: 
  (inputs have same domains)

Factorized representations [Olteanu and Závodný, 2015]



x	y	z
A	B	C
A	B	D
B	C	E

- Directed acyclic graph of **gates**
- **Output** gate: 
- **Variable** gates $\langle x : a \rangle$: represent a **single-tuple** and **single-column** relation
- **Relational product** gates: 

  (input domains are disjoint)
- **Union** gates: 

  (inputs have same domains)

Conditions on d-representations:

- **Deterministic**: all unions are disjoint
- **Normal**: no union is an input to a union

Enumerating tuples for normal deterministic d-representations

Task: Enumerate the tuples of the relation $R(g)$ captured by a gate g

Enumerating tuples for normal deterministic d-representations

Task: Enumerate the tuples of the relation $R(g)$ captured by a gate g

Base case: variable $\langle x : a \rangle$:

Enumerating tuples for normal deterministic d-representations

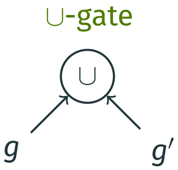
Task: Enumerate the tuples of the relation $R(g)$ captured by a gate g

Base case: variable $\langle x : a \rangle$: enumerate $\langle x : a \rangle$ and stop

Enumerating tuples for normal deterministic d-representations

Task: Enumerate the tuples of the relation $R(g)$ captured by a gate g

Base case: variable $\langle x : a \rangle$: enumerate $\langle x : a \rangle$ and stop

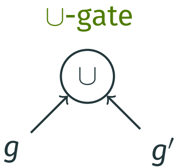


Union: enumerate $R(g)$ and then enumerate $R(g')$

Enumerating tuples for normal deterministic d-representations

Task: Enumerate the tuples of the relation $R(g)$ captured by a gate g

Base case: variable $\langle x : a \rangle$: enumerate $\langle x : a \rangle$ and stop



Union: enumerate $R(g)$ and then enumerate $R(g')$

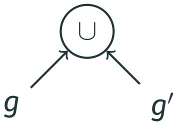
Determinism: no duplicates

Enumerating tuples for normal deterministic d-representations

Task: Enumerate the tuples of the relation $R(g)$ captured by a gate g

Base case: variable $\langle x : a \rangle$: enumerate $\langle x : a \rangle$ and stop

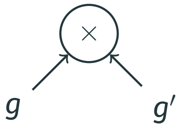
$\cup$-gate



Union: enumerate $R(g)$ and then enumerate $R(g')$

Determinism: no duplicates

$\times$-gate



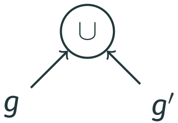
Product: enumerate $R(g)$ and for each result t

Enumerating tuples for normal deterministic d-representations

Task: Enumerate the tuples of the relation $R(g)$ captured by a gate g

Base case: variable $\langle x : a \rangle$: enumerate $\langle x : a \rangle$ and stop

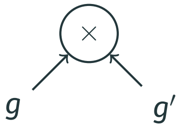
$\cup$-gate



Union: enumerate $R(g)$ and then enumerate $R(g')$

Determinism: no duplicates

$\times$-gate



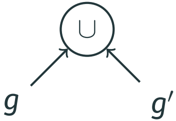
Product: enumerate $R(g)$ and for each result t enumerate $R(g')$ and for each result t'

Enumerating tuples for normal deterministic d-representations

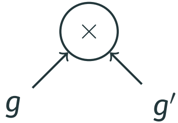
Task: Enumerate the tuples of the relation $R(g)$ captured by a gate g

Base case: variable $\langle x : a \rangle$: enumerate $\langle x : a \rangle$ and stop

$\cup$-gate



$\times$-gate



Union: enumerate $R(g)$ and then enumerate $R(g')$

Determinism: no duplicates

Product: enumerate $R(g)$ and for each result t
  enumerate $R(g')$ and for each result t'
    concatenate t and t'

Enumerating tuples for normal deterministic d-representations (2)

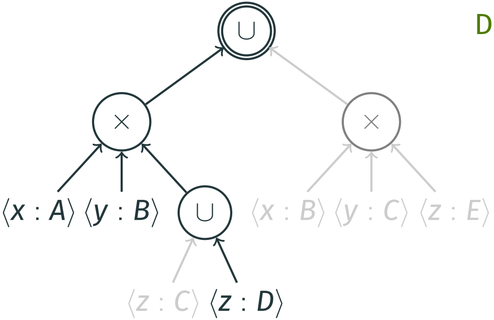
Theorem ([Olteanu and Závodný, 2015], Theorem 4.11)

For any fixed schema $S = (x_1, \ldots, x_k)$, the tuples of a normal deterministic d-representation with schema S can be enumerated in constant delay

Enumerating tuples for normal deterministic d-representations (2)

Theorem ([Olteanu and Závodný, 2015], Theorem 4.11)

For any fixed schema $S = (x_1, \ldots, x_k)$, the tuples of a normal deterministic d-representation with schema S can be enumerated in constant delay

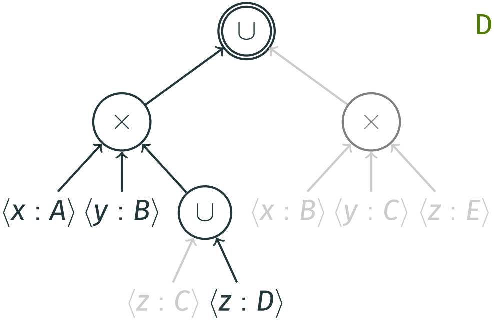


Delay analysis:

Enumerating tuples for normal deterministic d-representations (2)

Theorem ([Olteanu and Závodný, 2015], Theorem 4.11)

For any fixed schema $S = (x_1, \ldots, x_k)$, the tuples of a normal deterministic d-representation with schema S can be enumerated in constant delay



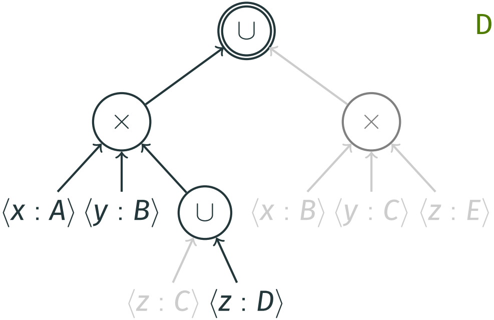
Delay analysis:

- Every product gate nontrivially splits the assignment to produce

Enumerating tuples for normal deterministic d-representations (2)

Theorem ([Olteanu and Závodný, 2015], Theorem 4.11)

For any fixed schema $S = (x_1, \ldots, x_k)$, the tuples of a normal deterministic d-representation with schema S can be enumerated in constant delay



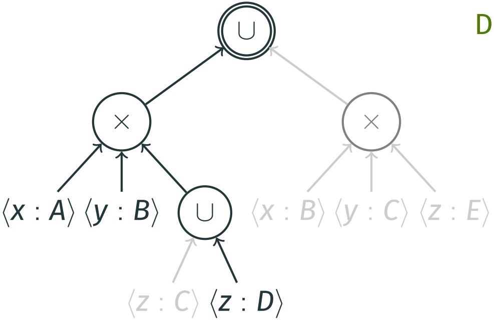
Delay analysis:

- Every product gate nontrivially splits the assignment to produce
- The inputs to union gates are not union gates (the representation is normal)

Enumerating tuples for normal deterministic d-representations (2)

Theorem ([Olteanu and Závodný, 2015], Theorem 4.11)

For any fixed schema $S = (x_1, \ldots, x_k)$, the tuples of a normal deterministic d-representation with schema S can be enumerated in constant delay



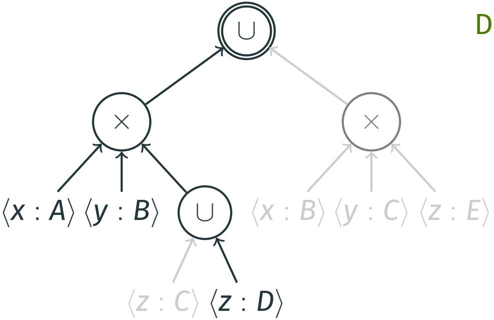
Delay analysis:

- Every product gate nontrivially splits the assignment to produce
- The inputs to union gates are not union gates (the representation is normal)
- Hence, the trace (gates visited to get a tuple) has size linear in the tuple arity, hence constant

Enumerating tuples for normal deterministic d-representations (2)

Theorem ([Olteanu and Závodný, 2015], Theorem 4.11)

For any fixed schema $S = (x_1, \ldots, x_k)$, the tuples of a normal deterministic d-representation with schema S can be enumerated in constant delay



Delay analysis:

- Every product gate **nontrivially splits** the assignment to produce
- The inputs to union gates are **not union gates** (the representation is **normal**)
- Hence, the **trace** (gates visited to get a tuple) has size **linear in the tuple arity**, hence **constant**

Note: normal deterministic d-representations also allow us to:

- **Count** the number of solutions in linear time
- **Access** the i-th solution, given i, in logarithmic time

Factorized representations for full acyclic CQs

Theorem

*Given an acyclic CQ Q and database D, we can compute a **deterministic normal d-representation of $Q(D)$** in time $O(|Q| \times |D|)$ and hence enumerate $Q(D)$ with **linear preprocessing and constant delay***

Factorized representations for full acyclic CQs

Theorem

*Given an acyclic CQ **Q** and database **D**, we can compute a **deterministic normal d-representation of Q(D)** in time $O(|Q| \times |D|)$ and hence enumerate $Q(D)$ with **linear preprocessing and constant delay***

Subscribed(x, y)

x	y
b	a
b	a'
b	a''
b'	a'

Follows(w, x)

w	x
a	b
a	b''
a''	b'

Follows(y, z)

y	z
a	b
a	b''
a''	b'

Factorized representations for full acyclic CQs

Theorem

*Given an acyclic CQ **Q** and database **D**, we can compute a **deterministic normal d-representation of Q(D)** in time $O(|Q| \times |D|)$ and hence enumerate $Q(D)$ with **linear preprocessing and constant delay***

Subscribed(x, y)

x	y
b	a
~~b~~	~~a'~~
b	a''
~~b'~~	~~a'~~

Follows(w, x)

w	x
a	b
~~a~~	~~b''~~
~~a''~~	~~b'~~

Follows(y, z)

y	z
a	b
a	b''
a''	b'

Factorized representations for full acyclic CQs

Theorem

Given an acyclic CQ $\mathbf{Q}$ *and database* $\mathbf{D}$*, we can compute a* ***deterministic normal d-representation of*** $Q(D)$ *in time* $O(|\mathbf{Q}| \times |\mathbf{D}|)$ *and hence enumerate* $\mathbf{Q}(\mathbf{D})$ *with* ***linear preprocessing and constant delay***

Subscribed(x, y)

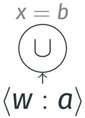
x	y
b	a
~~b~~	~~a'~~
b	a''
~~b'~~	~~a'~~

Follows(w, x)

w	x
a	b
~~a~~	~~b''~~
~~a''~~	~~b'~~

Follows(y, z)

y	z
a	b
a	b''
a''	b'



Factorized representations for full acyclic CQs

Theorem

Given an acyclic CQ $\mathbf{Q}$ *and database* $\mathbf{D}$*, we can compute a* ***deterministic normal d-representation of*** $Q(D)$ *in time* $O(|\mathbf{Q}| \times |\mathbf{D}|)$ *and hence enumerate* $\mathbf{Q}(\mathbf{D})$ *with* ***linear preprocessing and constant delay***

Subscribed(x, y)

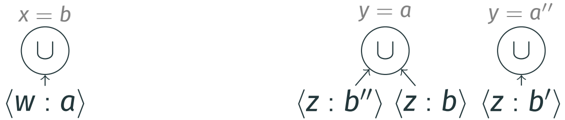
x	y
b	a
~~b~~	~~a'~~
b	a''
~~b'~~	~~a'~~

Follows(w, x)

w	x
a	b
~~a~~	~~b''~~
~~a''~~	~~b'~~

Follows(y, z)

y	z
a	b
a	b''
a''	b'



Factorized representations for full acyclic CQs

Theorem

Given an acyclic CQ $\mathbf{Q}$ *and database* $\mathbf{D}$*, we can compute a* ***deterministic normal d-representation of*** $Q(D)$ *in time* $O(|\mathbf{Q}| \times |\mathbf{D}|)$ *and hence enumerate* $\mathbf{Q}(D)$ *with* ***linear preprocessing and constant delay***

Subscribed(x, y)

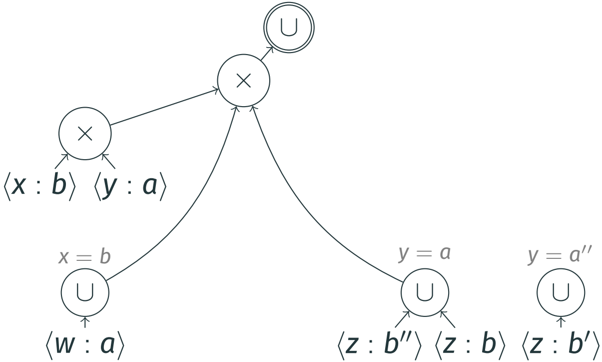
x	y
b	a
~~b~~	~~a'~~
b	a''
~~b'~~	~~a'~~

Follows(w, x)

w	x
a	b
~~a~~	~~b''~~
~~a''~~	~~b'~~

Follows(y, z)

y	z
a	b
a	b''
a''	b'



Factorized representations for full acyclic CQs

Theorem

Given an acyclic CQ $\mathbf{Q}$ *and database* $\mathbf{D}$*, we can compute a* ***deterministic normal d-representation of*** $Q(D)$ *in time* $O(|\mathbf{Q}| \times |\mathbf{D}|)$ *and hence enumerate* $\mathbf{Q}(\mathbf{D})$ *with* ***linear preprocessing and constant delay***

Subscribed(x, y)

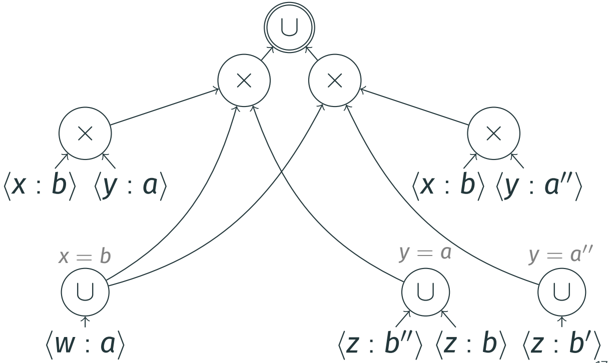
x	y
b	a
~~b~~	~~a'~~
b	a''
~~b'~~	~~a'~~

Follows(w, x)

w	x
a	b
~~a~~	~~b''~~
~~a''~~	~~b'~~

Follows(y, z)

y	z
a	b
a	b''
a''	b'



Enumeration for CQs with projections

General CQs extend **full CQs** by making it possible to **project away** some variables:

$$Q(x,z) : \exists y\ \mathsf{Follows}(x,y) \land \mathsf{Follows}(y,z)$$

Enumeration for CQs with projections

General CQs extend **full CQs** by making it possible to **project away** some variables:

$Q(x, z) : \exists y \ \text{Follows}(x, y) \wedge \text{Follows}(y, z)$ Join tree: $\text{Follows}(x, y) - \text{Follows}(y, z)$

Enumeration for CQs with projections

General CQs extend **full CQs** by making it possible to **project away** some variables:

$Q(x, z) : \exists y \text{ Follows}(x, y) \wedge \text{Follows}(y, z)$ Join tree: $\text{Follows}(x, y) - \text{Follows}(y, z)$

A CQ is **free-connex** if it is acyclic and admits a **join tree** which is **free-connex**: there is a **connected subtree** of tree nodes whose union is **exactly** the free variables

$\rightarrow$ In particular, the **free-connex full CQs** are simply the **acyclic CQs**

Enumeration for CQs with projections

General CQs extend full CQs by making it possible to project away some variables:

$Q(x, z) : \exists y \text{ Follows}(x, y) \wedge \text{Follows}(y, z)$ Join tree: $\text{Follows}(x, y) - \text{Follows}(y, z)$

A CQ is free-connex if it is acyclic and admits a join tree which is free-connex: there is a connected subtree of tree nodes whose union is exactly the free variables

$\rightarrow$ In particular, the free-connex full CQs are simply the acyclic CQs

Theorem ([Bagan et al., 2007])

For any fixed free-connex CQ Q, given a database D, we can enumerate $Q(D)$ with linear preprocessing and constant delay

Enumeration for CQs with projections

General CQs extend **full CQs** by making it possible to **project away** some variables:

$Q(x, z) : \exists y \text{ Follows}(x, y) \wedge \text{Follows}(y, z)$ Join tree: $\text{Follows}(x, y) — \text{Follows}(y, z)$

A CQ is **free-connex** if it is acyclic and admits a **join tree** which is **free-connex**: there is a **connected subtree** of tree nodes whose union is **exactly** the free variables

→ In particular, the **free-connex full CQs** are simply the **acyclic CQs**

Theorem ([Bagan et al., 2007])

For any fixed free-connex CQ Q*, given a database* D*, we can enumerate* $Q(D)$ *with* ***linear preprocessing*** *and* ***constant delay***

This can also be shown via **deterministic normal d-representations**

Lower bounds for CQ enumeration

What about enumeration for non-free-connex CQs? Let us assume:

- The query is **minimized**: can always be done without loss of generality
- The query is **without self joins**: uses only each relation name once
  - $\rightarrow$ $Q(x, y, z) : \text{Follows}(x, y) \land \text{Subscribed}(y, z)$ but not $Q(x, y, z) : \text{Follows}(x, y) \land \text{Follows}(y, z)$

Lower bounds for CQ enumeration

What about enumeration for non-free-connex CQs? Let us assume:

- The query is **minimized**: can always be done without loss of generality
- The query is **without self joins**: uses only each relation name once
  - $\rightarrow$ $Q(x, y, z)$: Follows$(x, y) \wedge$ Subscribed(y, z) but not $Q(x, y, z)$: Follows$(x, y) \wedge$ Follows(y, z)

Then **conditional lower bounds** are known:

Theorem ([Bagan et al., 2007, Carmeli and Segoufin, 2023])

Let Q *be a* ***self-join free CQ*** *enumerable with linear preprocessing and constant delay:*

Lower bounds for CQ enumeration

What about enumeration for non-free-connex CQs? Let us assume:

- The query is **minimized**: can always be done without loss of generality
- The query is **without self joins**: uses only each relation name once
  - $\rightarrow$ $Q(x, y, z)$: Follows$(x, y) \wedge$ Subscribed(y, z) but not $Q(x, y, z)$: Follows$(x, y) \wedge$ Follows(y, z)

Then **conditional lower bounds** are known:

Theorem ([Bagan et al., 2007, Carmeli and Segoufin, 2023])

Let Q be a ***self-join free CQ*** *enumerable with linear preprocessing and constant delay:*

- *If Q is* ***not acyclic****, then for some k we can detect* ***k-hypercliques*** *in linear time*
  - $\rightarrow$ *for $k = 3$: we can find triangles in undirected graphs in linear time*

Lower bounds for CQ enumeration

What about enumeration for non-free-connex CQs? Let us assume:

- The query is **minimized**: can always be done without loss of generality
- The query is **without self joins**: uses only each relation name once
  - → $Q(x, y, z)$: Follows$(x, y) \wedge$ Subscribed(y, z) but not $Q(x, y, z)$: Follows$(x, y) \wedge$ Follows(y, z)

Then **conditional lower bounds** are known:

Theorem ([Bagan et al., 2007, Carmeli and Segoufin, 2023])

*Let Q be a **self-join free CQ** enumerable with linear preprocessing and constant delay:*

- *If Q is **not acyclic**, then for some k we can detect **k-hypercliques** in linear time*
  - → *for $k = 3$: we can find triangles in undirected graphs in linear time*
- *If Q is **acyclic but not free-connex**, then we can multiply n-by-n matrices in $O(n^2)$*
  - → *we can even do it on sparse matrices*

What about CQs with self-joins?

Can we lift the self-join-freeness hypothesis?

What about CQs with self-joins?

Can we **lift the self-join-freeness hypothesis**?

- No problem with self-joins in the **upper bound** (Yannakakis's algorithm)

What about CQs with self-joins?

Can we lift the self-join-freeness hypothesis?

- No problem with self-joins in the upper bound (Yannakakis's algorithm)
- Queries with self-joins do not get harder if we distinguish every atom

What about CQs with self-joins?

Can we lift the self-join-freeness hypothesis?

- No problem with self-joins in the upper bound (Yannakakis's algorithm)
- Queries with self-joins do not get harder if we distinguish every atom

However, the presence of self-joins can make queries easier!

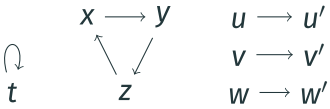
What about CQs with self-joins?

Can we **lift the self-join-freeness hypothesis**?

- No problem with self-joins in the **upper bound** (Yannakakis's algorithm)
- Queries with self-joins do not get **harder** if we distinguish every atom

However, the presence of self-joins can make queries **easier**!

$$Q(t, x, y, z, u, u', v, v', w, w') : R(t, t), R(x, y), R(y, z), R(z, x), R(u, u'), R(v, v'), R(w, w')$$



(Example from [Berkholz et al., 2020])

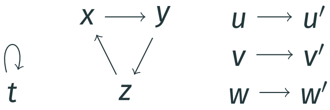
What about CQs with self-joins?

Can we **lift the self-join-freeness hypothesis**?

- No problem with self-joins in the **upper bound** (Yannakakis's algorithm)
- Queries with self-joins do not get **harder** if we distinguish every atom

However, the presence of self-joins can make queries **easier**!

$$Q(t, x, y, z, u, u', v, v', w, w') : R(t, t), R(x, y), R(y, z), R(z, x), R(u, u'), R(v, v'), R(w, w')$$



(Example from [Berkholz et al., 2020])

- Q is **easy**: intuitively, the results from the last 3 atoms easily "reveal" all triangles

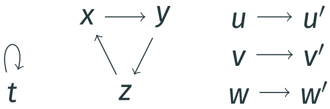
What about CQs with self-joins?

Can we **lift the self-join-freeness hypothesis**?

- No problem with self-joins in the **upper bound** (Yannakakis's algorithm)
- Queries with self-joins do not get **harder** if we distinguish every atom

However, the presence of self-joins can make queries **easier**!

$$Q(t, x, y, z, u, u', v, v', w, w') : R(t,t), R(x,y), R(y,z), R(z,x), R(u,u'), R(v,v'), R(w,w')$$



(Example from [Berkholz et al., 2020])

- Q is **easy**: intuitively, the results from the last 3 atoms easily "reveal" all triangles
- Q' obtained from Q by distinguishing every atom is **hard** (can find triangles)

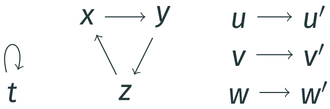
What about CQs with self-joins?

Can we **lift the self-join-freeness hypothesis**?

- No problem with self-joins in the **upper bound** (Yannakakis's algorithm)
- Queries with self-joins do not get **harder** if we distinguish every atom

However, the presence of self-joins can make queries **easier**!

$$Q(t, x, y, z, u, u', v, v', w, w') : R(t, t), R(x, y), R(y, z), R(z, x), R(u, u'), R(v, v'), R(w, w')$$



(Example from [Berkholz et al., 2020])

- Q is **easy**: intuitively, the results from the last 3 atoms easily "reveal" all triangles
- Q' obtained from Q by distinguishing every atom is **hard** (can find triangles)

Open problem: dichotomy on CQs with self-joins? see [Carmeli and Segoufin, 2023]

What about unions of CQs?

Union of CQs (UCQs): a disjunction of conjunctive queries

$$Q(x, z) : \left(\exists y \; \mathsf{Follows}(x, y) \wedge \mathsf{Subscribed}(y, z) \right) \vee \mathsf{Subscribed}(z, x)$$

What about unions of CQs?

Union of CQs (UCQs): a disjunction of conjunctive queries

$$Q(x, z) : \left(\exists y \ \mathrm{Follows}(x, y) \wedge \mathrm{Subscribed}(y, z) \right) \vee \mathrm{Subscribed}(z, x)$$

- The union of **easy CQs** is always easy
  - $\rightarrow$ Only subtlety is removing **duplicates**, but there are constantly many

What about unions of CQs?

Union of CQs (UCQs): a disjunction of conjunctive queries

$$Q(x,z) : \left(\exists y \; \text{Follows}(x,y) \wedge \text{Subscribed}(y,z)\right) \vee \text{Subscribed}(z,x)$$

- The union of **easy CQs** is always easy
  - $\rightarrow$ Only subtlety is removing **duplicates**, but there are constantly many
- The union of an **easy CQ** and a **hard CQ** can be easy!

What about unions of CQs?

Union of CQs (UCQs): a disjunction of conjunctive queries

$$Q(x,z) : \left(\exists y \ \mathrm{Follows}(x,y) \land \mathrm{Subscribed}(y,z)\right) \lor \mathrm{Subscribed}(z,x)$$

- The union of **easy CQs** is always easy
  - → Only subtlety is removing **duplicates**, but there are constantly many
- The union of an **easy CQ** and a **hard CQ** can be easy!
- The union of a **hard CQ** and a **hard CQ** can be easy!

What about unions of CQs?

Union of CQs (UCQs): a disjunction of conjunctive queries

$$Q(x,z) : \left(\exists y \ \text{Follows}(x,y) \land \text{Subscribed}(y,z)\right) \lor \text{Subscribed}(z,x)$$

- The union of **easy CQs** is always easy
  - → Only subtlety is removing **duplicates**, but there are constantly many
- The union of an **easy CQ** and a **hard CQ** can be easy!
- The union of a **hard CQ** and a **hard CQ** can be easy!
- This can happen even if each CQ does not have **self-joins**! [Carmeli, 2022]

What about unions of CQs?

Union of CQs (UCQs): a disjunction of conjunctive queries

$$Q(x, z) : \left(\exists y \; \text{Follows}(x, y) \wedge \text{Subscribed}(y, z)\right) \vee \text{Subscribed}(z, x)$$

- The union of **easy CQs** is always easy
  - → Only subtlety is removing **duplicates**, but there are constantly many
- The union of an **easy CQ** and a **hard CQ** can be easy!
- The union of a **hard CQ** and a **hard CQ** can be easy!
- This can happen even if each CQ does not have **self-joins**! [Carmeli, 2022]

Open problem: dichotomy on UCQs? see [Carmeli and Kröll, 2021]

Table of contents

Introduction: From CQs/UCQs to automata

- So far we have seen results on enumeration for CQs and UCQs

Introduction: From CQs/UCQs to automata

- So far we have seen results on enumeration for CQs and UCQs
- Enumeration has also been studied for other query languages, especially when restricting the input data
  - For first-order logic (FO) on bounded-degree graphs [Durand and Grandjean, 2007, Kazana and Segoufin, 2011]

Introduction: From CQs/UCQs to automata

- So far we have seen results on enumeration for CQs and UCQs
- Enumeration has also been studied for other query languages, especially when restricting the input data
  - For first-order logic (FO) on bounded-degree graphs [Durand and Grandjean, 2007, Kazana and Segoufin, 2011]
  - For FO on nowhere-dense graphs [Schweikardt et al., 2022]

Introduction: From CQs/UCQs to automata

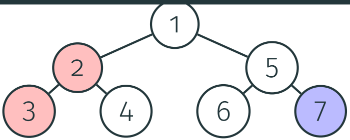
- So far we have seen results on enumeration for **CQs** and **UCQs**
- Enumeration has also been studied for **other query languages**, especially when restricting the input **data**
  - For **first-order logic** (FO) on **bounded-degree graphs** [Durand and Grandjean, 2007, Kazana and Segoufin, 2011]
  - For FO on **nowhere-dense graphs** [Schweikardt et al., 2022]
  - For **monadic second-order logic** (MSO) on **trees** [Bagan, 2006, Kazana and Segoufin, 2013]

Introduction: From CQs/UCQs to automata

- So far we have seen results on enumeration for CQs and UCQs
- Enumeration has also been studied for other query languages, especially when restricting the input data
  - For first-order logic (FO) on bounded-degree graphs [Durand and Grandjean, 2007, Kazana and Segoufin, 2011]
  - For FO on nowhere-dense graphs [Schweikardt et al., 2022]
  - For monadic second-order logic (MSO) on trees [Bagan, 2006, Kazana and Segoufin, 2013]
- Now: review enumeration results for MSO, in terms of factorized representations (not necessarily normal or deterministic)

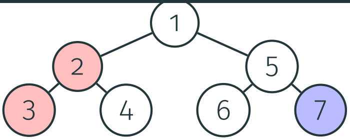
MSO query evaluation on trees

Data: a **tree** *T* where nodes have a color from an alphabet



MSO query evaluation on trees

Data: a **tree** T where nodes have a color from an alphabet ○ ○ ○



Query Q in monadic second-order logic (MSO)

- $P_{\bigcirc}(x)$ means "x is blue"
- $x \rightarrow y$ means "x is the parent of y"

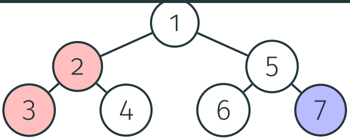
Equivalent formalism: **tree automata**

"Find the pairs of a pink node and a blue node?"

$Q(x, y) := P_{\bigcirc}(x) \wedge P_{\bigcirc}(y)$

MSO query evaluation on trees

Data: a **tree** T where nodes have a color from an alphabet ○ ○ ○



Query Q in monadic second-order logic (MSO)

- $P_{\circ}(x)$ means "x is blue"
- $x \rightarrow y$ means "x is the parent of y"

Equivalent formalism: **tree automata**

"Find the pairs of a pink node and a blue node?"

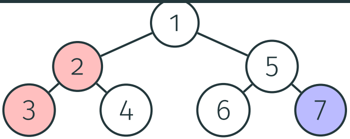
$Q(x, y) := P_{\circ}(x) \wedge P_{\circ}(y)$

Result: **Enumerate** all pairs (a, b) of nodes of T such that $Q(a, b)$ holds

results: $(2, 7), (3, 7)$

MSO query evaluation on trees

Data: a **tree** T where nodes have a color from an alphabet ○ ○ ○



Query Q in monadic second-order logic (MSO)

- $P_{○}(x)$ means "x is blue"
- $x \rightarrow y$ means "x is the parent of y"

Equivalent formalism: **tree automata**

"Find the pairs of a pink node and a blue node?"

$Q(x, y) := P_{○}(x) \wedge P_{○}(y)$

Result: **Enumerate** all pairs (a, b) of nodes of T such that $Q(a, b)$ holds

results: $(2, 7), (3, 7)$

Data complexity: Measure efficiency as a function of T (the query Q is **fixed**)

Results for MSO on trees

Theorem [Bagan, 2006, Kazana and Segoufin, 2013]

We can enumerate the answers of MSO queries on trees with linear-time preprocessing and constant delay.

Results for MSO on trees

Theorem [Bagan, 2006, Kazana and Segoufin, 2013]

We can enumerate the answers of MSO queries on trees with linear-time preprocessing and constant delay.

We can reprove this via factorized representations:

Theorem (A., Bourhis, Jachiet, Mengel, ICALP'17)

For any fixed bottom-up deterministic tree automaton A with "captures", given a tree T, we can build a deterministic d-representation capturing the results of A on T in $O(|T|)$

Results for MSO on trees

Theorem [Bagan, 2006, Kazana and Segoufin, 2013]

We can enumerate the answers of MSO queries on trees with linear-time preprocessing and constant delay.

We can reprove this via factorized representations:

Theorem (A., Bourhis, Jachiet, Mengel, ICALP'17)

For any fixed bottom-up deterministic tree automaton A with "captures", given a tree T, we can build a deterministic d-representation capturing the results of A on T in $O(|T|)$

Note that the d-representation is no longer normal, but we show with some effort:

Theorem (A., Bourhis, Jachiet, Mengel, ICALP'17)

For any fixed schema $S = (x_1, \ldots, x_k)$, the tuples of a deterministic d-representation with schema S can be enumerated with linear preprocessing and constant delay

Enumerating matches of nondeterministic document spanners

Data: a text *T*

```
Antoine Amarilli Description Name Antoine Amarilli.  Handle:  a3nm.  Identity Born 1990-02-07.  French national.  Appearance as
of 2017.  Auth OpenPGP. OpenId.  Bitcoin.  Contact Email and XMPP a3nm@a3nm.net Affiliation Associate professor of computer
science (office C201-4) in the DIG team of Télécom ParisTech, 46 rue Barrault, F-75634 Paris Cedex 13, France.  Studies
PhD in computer science awarded by Télécom ParisTech on March 14, 2016.  Former student of the École normale supérieure.
test@example.com More Résumé Location Other sites Blogging:  a3nm.net/blog Git:  a3nm.net/git ...
```

Enumerating matches of nondeterministic document spanners

Data: a text T

```
Antoine Amarilli Description Name Antoine Amarilli.  Handle:  a3nm.  Identity Born 1990-02-07.  French national.  Appearance as
of 2017.  Auth OpenPGP. OpenId.  Bitcoin.  Contact Email and XMPP a3nm@a3nm.net Affiliation Associate professor of computer
science (office C201-4) in the DIG team of Télécom ParisTech, 46 rue Barrault, F-75634 Paris Cedex 13, France.  Studies
PhD in computer science awarded by Télécom ParisTech on March 14, 2016.  Former student of the École normale supérieure.
test@example.com More Résumé Location Other sites Blogging:  a3nm.net/blog Git:  a3nm.net/git ...
```

Query: a pattern P given as a regular expression

$$P := \texttt{␣ [a-z0-9.]}^{*}\ \texttt{@ [a-z0-9.]}^{*}\ \texttt{␣}$$

Enumerating matches of nondeterministic document spanners

Data: a **text** T

```
Antoine Amarilli Description Name Antoine Amarilli.  Handle:  a3nm.  Identity Born 1990-02-07.  French national.  Appearance as
of 2017.  Auth OpenPGP. OpenId.  Bitcoin.  Contact Email and XMPP a3nm@a3nm.net Affiliation Associate professor of computer
science (office C201-4) in the DIG team of Télécom ParisTech, 46 rue Barrault, F-75634 Paris Cedex 13, France.  Studies
PhD in computer science awarded by Télécom ParisTech on March 14, 2016.  Former student of the École normale supérieure.
test@example.com More Résumé Location Other sites Blogging:  a3nm.net/blog Git:  a3nm.net/git ...
```

Query: a **pattern** P given as a regular expression

$$P := \textvisiblespace\ \texttt{[a-z0-9.]}^*\ \texttt{@}\ \texttt{[a-z0-9.]}^*\ \textvisiblespace$$

Output: the list of **substrings** of T that match P:

$$[186, 200\rangle, \quad [483, 500\rangle, \quad \ldots$$

Enumerating matches of nondeterministic document spanners

Data: a **text** T

```
Antoine Amarilli Description Name Antoine Amarilli. Handle: a3nm. Identity Born 1990-02-07. French national. Appearance as
of 2017. Auth OpenPGP. OpenId. Bitcoin. Contact Email and XMPP a3nm@a3nm.net Affiliation Associate professor of computer
science (office C201-4) in the DIG team of Télécom ParisTech, 46 rue Barrault, F-75634 Paris Cedex 13, France. Studies
PhD in computer science awarded by Télécom ParisTech on March 14, 2016. Former student of the École normale supérieure.
test@example.com More Résumé Location Other sites Blogging: a3nm.net/blog Git: a3nm.net/git ...
```

Query: a **pattern** P given as a regular expression

$$P := \sqcup\ \texttt{[a-z0-9.]}^*\ \texttt{@}\ \texttt{[a-z0-9.]}^*\ \sqcup$$

Output: the list of **substrings** of T that match P:

$$[186, 200\rangle,\quad [483, 500\rangle,\quad \dots$$

Goal:

- be **very efficient** in T (constant-delay)
- be **reasonably efficient** in P (polynomial-time)

Results for nondeterministic document spanners

Theorem (A., Bourhis, Mengel, Niewerth, ICDT'19; see also PODS'19)

We can enumerate all matches of an input nondeterministic automaton with captures on an input text with

- *Preprocessing linear in the text and polynomial in the automaton*
- *Delay constant in the text and polynomial in the automaton*

Results for nondeterministic document spanners

Theorem (A., Bourhis, Mengel, Niewerth, ICDT'19; see also PODS'19)

*We can enumerate all matches of an input **nondeterministic automaton with captures** on an input **text** with*

- *Preprocessing **linear** in the text and **polynomial** in the automaton*
- *Delay **constant** in the text and **polynomial** in the automaton*

- Generalizes earlier result on **deterministic automata** [Florenzano et al., 2018]
- To make the algorithm polynomial in the **(nondeterministic) automaton**, we need efficient enumeration for a certain kind of **non-deterministic d-representations**

Other enumeration settings

Efficient enumeration is now being studied in **many settings** in data management (sometimes with weaker guarantees than linear preprocessing and constant delay):

- For **regular path queries** [Martens and Trautner, 2018, David et al., 2024]
- For **compressed structures**:
  - Compressed trees [Lohrey and Schmid, 2024]
  - SLP-compressed documents [Schmid and Schweikardt, 2021, Muñoz and Riveros, 2023]
- For **visibly pushdown languages** [Muñoz and Riveros, 2022]
- For **context-free languages** with annotations [Peterfreund, 2021], [A., Jachiet, Muñoz, Riveros, 2023]

There are also **software implementations** [Riveros et al., 2023]

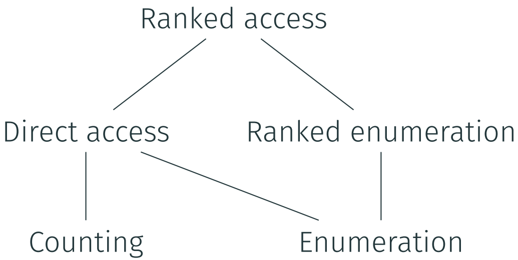
Table of contents

Introduction: From enumeration to more general tasks

Sometimes, we want **more** than enumerating query results in an unspecified order:

Introduction: From enumeration to more general tasks

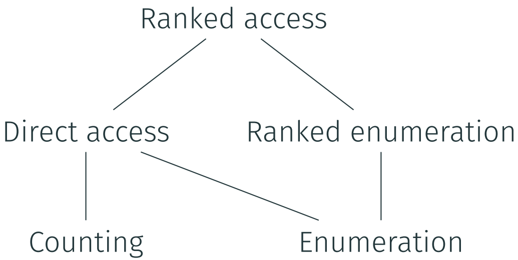
Sometimes, we want **more** than enumerating query results in an unspecified order:



(Adapted from [Carmeli, 2023])

Introduction: From enumeration to more general tasks

Sometimes, we want **more** than enumerating query results in an unspecified order:

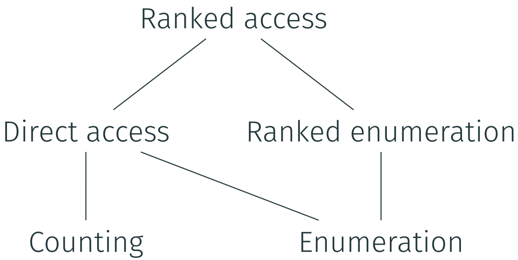


(Adapted from [Carmeli, 2023])

- **Direct access**: getting the i-th answer
- **Counting** the answers
- **Ranked enumeration**: enumerating in a prescribed order
- **Ranked access**: getting the i-th tuple in a prescribed order

Introduction: From enumeration to more general tasks

Sometimes, we want **more** than enumerating query results in an unspecified order:



(Adapted from [Carmeli, 2023])

- **Direct access**: getting the i-th answer
- **Counting** the answers
- **Ranked enumeration**: enumerating in a prescribed order
- **Ranked access**: getting the i-th tuple in a prescribed order

Another question: **maintain** an enumeration structure under **updates** to the data

Results on ranked enumeration / ranked access

For **CQs and UCQs**:

- Most works study self-join-free CQs under **lexicographic orders** and aim for **logarithmic** access time or delay:
  - Characterization of **tractable orders** for CQs [Carmeli et al., 2023]

Results on ranked enumeration / ranked access

For CQs and UCQs:

- Most works study self-join-free CQs under lexicographic orders and aim for logarithmic access time or delay:
  - Characterization of tractable orders for CQs [Carmeli et al., 2023]
  - Characterization of optimal preprocessing time for polylog direct access [Bringmann et al., 2022]

Results on ranked enumeration / ranked access

For CQs and UCQs:

- Most works study self-join-free CQs under lexicographic orders and aim for logarithmic access time or delay:
  - Characterization of tractable orders for CQs [Carmeli et al., 2023]
  - Characterization of optimal preprocessing time for polylog direct access [Bringmann et al., 2022]
  - Extensions to CQs with aggregation [Eldar et al., 2024]

Results on ranked enumeration / ranked access

For **CQs and UCQs**:

- Most works study self-join-free CQs under **lexicographic orders** and aim for **logarithmic** access time or delay:
  - Characterization of **tractable orders** for CQs [Carmeli et al., 2023]
  - Characterization of **optimal preprocessing time** for polylog direct access [Bringmann et al., 2022]
  - Extensions to CQs with **aggregation** [Eldar et al., 2024]
  - Extensions to CQs with **negated atoms** [Capelli and Irwin, 2024]

Results on ranked enumeration / ranked access

For **CQs and UCQs**:

- Most works study self-join-free CQs under **lexicographic orders** and aim for **logarithmic** access time or delay:
  - Characterization of **tractable orders** for CQs [Carmeli et al., 2023]
  - Characterization of **optimal preprocessing time** for polylog direct access [Bringmann et al., 2022]
  - Extensions to CQs with **aggregation** [Eldar et al., 2024]
  - Extensions to CQs with **negated atoms** [Capelli and Irwin, 2024]
- Other directions:
  - **Other ranking functions** defined by dioids [Tziavelis et al., 2020]

Results on ranked enumeration / ranked access

For CQs and UCQs:

- Most works study self-join-free CQs under lexicographic orders and aim for logarithmic access time or delay:
  - Characterization of tractable orders for CQs [Carmeli et al., 2023]
  - Characterization of optimal preprocessing time for polylog direct access [Bringmann et al., 2022]
  - Extensions to CQs with aggregation [Eldar et al., 2024]
  - Extensions to CQs with negated atoms [Capelli and Irwin, 2024]
- Other directions:
  - Other ranking functions defined by dioids [Tziavelis et al., 2020]
  - Random access and random-order enumeration [Carmeli et al., 2022]

Results on ranked enumeration / ranked access

For **CQs and UCQs**:

- Most works study self-join-free CQs under **lexicographic orders** and aim for **logarithmic** access time or delay:
  - Characterization of **tractable orders** for CQs [Carmeli et al., 2023]
  - Characterization of **optimal preprocessing time** for polylog direct access [Bringmann et al., 2022]
  - Extensions to CQs with **aggregation** [Eldar et al., 2024]
  - Extensions to CQs with **negated atoms** [Capelli and Irwin, 2024]
- Other directions:
  - **Other ranking functions** defined by dioids [Tziavelis et al., 2020]
  - **Random access** and **random-order** enumeration [Carmeli et al., 2022]

For **MSO** queries on trees:

- **Ranked enumeration** shown with **logarithmic delay** on **words** [Bourhis et al., 2021]
- Recent extension to **trees** [A., Bourhis, Capelli, Monet, 2024]

Incremental maintenance of enumeration structures

- Say the input data is often **modified**; we **restart** the enumeration after each update

Incremental maintenance of enumeration structures

- Say the input data is often **modified**; we **restart** the enumeration after each update
- Can we avoid **re-running** the preprocessing phase from scratch?

Incremental maintenance of enumeration structures

- Say the input data is often **modified**; we **restart** the enumeration after each update
- Can we avoid **re-running** the preprocessing phase from scratch?

For **self-join-free CQs**:

- Notion of **q-hierarchical CQs** that admit linear preprocessing and constant delay enumeration and **constant-time updates**; lower bounds [Berkholz et al., 2017]
- Results on the **preprocessing-delay-update tradeoff** for some CQs [Kara et al., 2023]

Incremental maintenance of enumeration structures

- Say the input data is often **modified**; we **restart** the enumeration after each update
- Can we avoid **re-running** the preprocessing phase from scratch?

For **self-join-free CQs**:

- Notion of **q-hierarchical CQs** that admit linear preprocessing and constant delay enumeration and **constant-time updates**; lower bounds [Berkholz et al., 2017]
- Results on the **preprocessing-delay-update tradeoff** for some CQs [Kara et al., 2023]

For **MSO** queries on trees, aiming for **logarithmic** update time:

- On **words**, linear preprocessing and constant delay enumeration is possible under **insert/delete updates** [Niewerth and Segoufin, 2018]
- On **trees**, linear preprocessing and constant delay enumeration is possible under **substitution updates** [A., Bourhis, Mengel, 2018] and possibly more

Table of contents

Summary and future work

- We have seen enumeration algorithms to produce query answers in streaming
  - $\rightarrow$ Ideally, we want linear preprocessing and constant delay
- Modular approach: compute a factorized representation of the results
- Tractable enumeration is possible for free-connex CQs and for MSO queries on trees
- Ongoing research: ranked enumeration, ranked access, incremental maintenance...

Summary and future work

- We have seen **enumeration algorithms** to produce query answers in streaming
  - $\rightarrow$ Ideally, we want **linear preprocessing** and **constant delay**
- **Modular approach:** compute a factorized representation of the results
- Tractable enumeration is possible for **free-connex CQs** and for **MSO queries on trees**
- **Ongoing research:** ranked enumeration, ranked access, incremental maintenance...

Other broad directions for further research:

- Enumerating **diverse** / **representative** solutions?
- Understanding the **tradeoff** between preprocessing time, memory, and delay?
- Understanding how the **update complexity** depends on the specific **query** posed?
- Can we enumerate **large objects** by **editing previous solutions**? (e.g., Gray code)

Summary and future work

- We have seen **enumeration algorithms** to produce query answers in streaming
  - → Ideally, we want **linear preprocessing** and **constant delay**
- **Modular approach:** compute a factorized representation of the results
- Tractable enumeration is possible for **free-connex CQs** and for **MSO queries on trees**
- **Ongoing research:** ranked enumeration, ranked access, incremental maintenance...

Other broad directions for further research:

- Enumerating **diverse** / **representative** solutions?
- Understanding the **tradeoff** between preprocessing time, memory, and delay?
- Understanding how the **update complexity** depends on the specific **query** posed?
- Can we enumerate **large objects** by **editing previous solutions**? (e.g., Gray code)

Thanks for your attention!

Amarilli, A., Bourhis, P., Capelli, F., and Monet, M. (2024).
Ranked enumeration for MSO on trees via knowledge compilation.
In *ICDT*.

Amarilli, A., Bourhis, P., Jachiet, L., and Mengel, S. (2017).
A circuit-based approach to efficient enumeration.
In *ICALP*.

Amarilli, A., Bourhis, P., and Mengel, S. (2018).
Enumeration on trees under relabelings.
In *ICDT*.

Amarilli, A., Bourhis, P., Mengel, S., and Niewerth, M. (2019a).
Constant-delay enumeration for nondeterministic document spanners.
In *ICDT*.

Amarilli, A., Bourhis, P., Mengel, S., and Niewerth, M. (2019b).
Enumeration on trees with tractable combined complexity and efficient updates.
In *PODS*.

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

Bagan, G. (2006).
MSO queries on Tree Decomposable Structures Are Computable with Linear Delay.
In *CSL*.

Bagan, G., Durand, A., and Grandjean, E. (2007).
On acyclic conjunctive queries and constant delay enumeration.
In *CSL*.

Berkholz, C., Gerhardt, F., and Schweikardt, N. (2020).
Constant delay enumeration for conjunctive queries: a tutorial.
ACM SIGLOG News, 7(1).

Berkholz, C., Keppeler, J., and Schweikardt, N. (2017).
Answering conjunctive queries under updates.
In *PODS*.

Bourhis, P., Grez, A., Jachiet, L., and Riveros, C. (2021).
Ranked enumeration of MSO logic on words.
In *ICDT*.

Bringmann, K., Carmeli, N., and Mengel, S. (2022).
Tight fine-grained bounds for direct access on join queries.
In *PODS*.

References v

Capelli, F. and Irwin, O. (2024).
Direct access for conjunctive queries with negation.
In *ICDT*.

Carmeli, N. (2022).
Answering unions of conjunctive queries with ideal time guarantees (invited talk).
In Olteanu, D. and Vortmeier, N., editors, *ICDT*, volume 220 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Carmeli, N. (2023).
Accessing answers to conjunctive queries with ideal time guarantees (abstract of invited talk).
In Kutz, O., Lutz, C., and Ozaki, A., editors, *DL*, volume 3515 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Carmeli, N. and Kröll, M. (2021).
On the enumeration complexity of unions of conjunctive queries.
TODS, 46(2).

Carmeli, N. and Segoufin, L. (2023).
Conjunctive queries with self-joins, towards a fine-grained enumeration complexity analysis.
In *PODS*.

Carmeli, N., Tziavelis, N., Gatterbauer, W., Kimelfeld, B., and Riedewald, M. (2023).
Tractable orders for direct access to ranked answers of conjunctive queries.
TODS, 48(1).

Carmeli, N., Zeevi, S., Berkholz, C., Conte, A., Kimelfeld, B., and Schweikardt, N. (2022).
Answering (unions of) conjunctive queries using random access and random-order enumeration.
TODS, 47(3).

David, C., Francis, N., and Marsault, V. (2024).
Distinct shortest walk enumeration for rpqs.
In *PODS*.

Durand, A. and Grandjean, E. (2007).
First-order queries on structures of bounded degree are computable with constant delay.
TOCL, 8(4).

Eldar, I., Carmeli, N., and Kimelfeld, B. (2024).
Direct access for answers to conjunctive queries with aggregation.
In *ICDT*.

Florenzano, F., Riveros, C., Ugarte, M., Vansummeren, S., and Vrgoc, D. (2018).
Constant delay algorithms for regular document spanners.
In *PODS*.

Kara, A., Nikolic, M., Olteanu, D., and Zhang, H. (2023).
Conjunctive queries with free access patterns under updates.
In *ICDT*.

Kazana, W. and Segoufin, L. (2011).
First-order query evaluation on structures of bounded degree.
Logical Methods in Computer Science, 7.

References x

Kazana, W. and Segoufin, L. (2013).
Enumeration of monadic second-order queries on trees.
TOCL, 14(4).

Lohrey, M. and Schmid, M. L. (2024).
Enumeration for MSO-queries on compressed trees.
In *PODS*.
To appear. arXiv preprint arXiv:2403.03067.

Martens, W. and Trautner, T. (2018).
Evaluation and enumeration problems for regular path queries.
In *ICDT*.

Muñoz, M. and Riveros, C. (2022).
Streaming enumeration on nested documents.
In *ICDT*.

Muñoz, M. and Riveros, C. (2023).
Constant-delay enumeration for SLP-compressed documents.
In *ICDT*.

Niewerth, M. and Segoufin, L. (2018).
Enumeration of MSO queries on strings with constant delay and logarithmic updates.
In *PODS*.

Olteanu, D. and Závodnỳ, J. (2015).
Size bounds for factorised representations of query results.
TODS, 40(1).

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

Riveros, C., Van Sint Jan, N., and Vrgoč, D. (2023).
Rematch: A novel regex engine for finding all matches.
PVLDB, 16(11).

Schmid, M. L. and Schweikardt, N. (2021).
Spanner evaluation over SLP-compressed documents.
In *PODS*.

Schweikardt, N., Segoufin, L., and Vigny, A. (2022).
Enumeration for FO queries over nowhere dense graphs.
JACM, 69(3).

Tziavelis, N., Ajwani, D., Gatterbauer, W., Riedewald, M., and Yang, X. (2020).
Optimal algorithms for ranked enumeration of answers to full conjunctive queries.
PVLDB, 13(9).

Yannakakis, M. (1981).
Algorithms for acyclic database schemes.
In *VLDB*, volume 81.