Discovering XSD Keys from XML Data

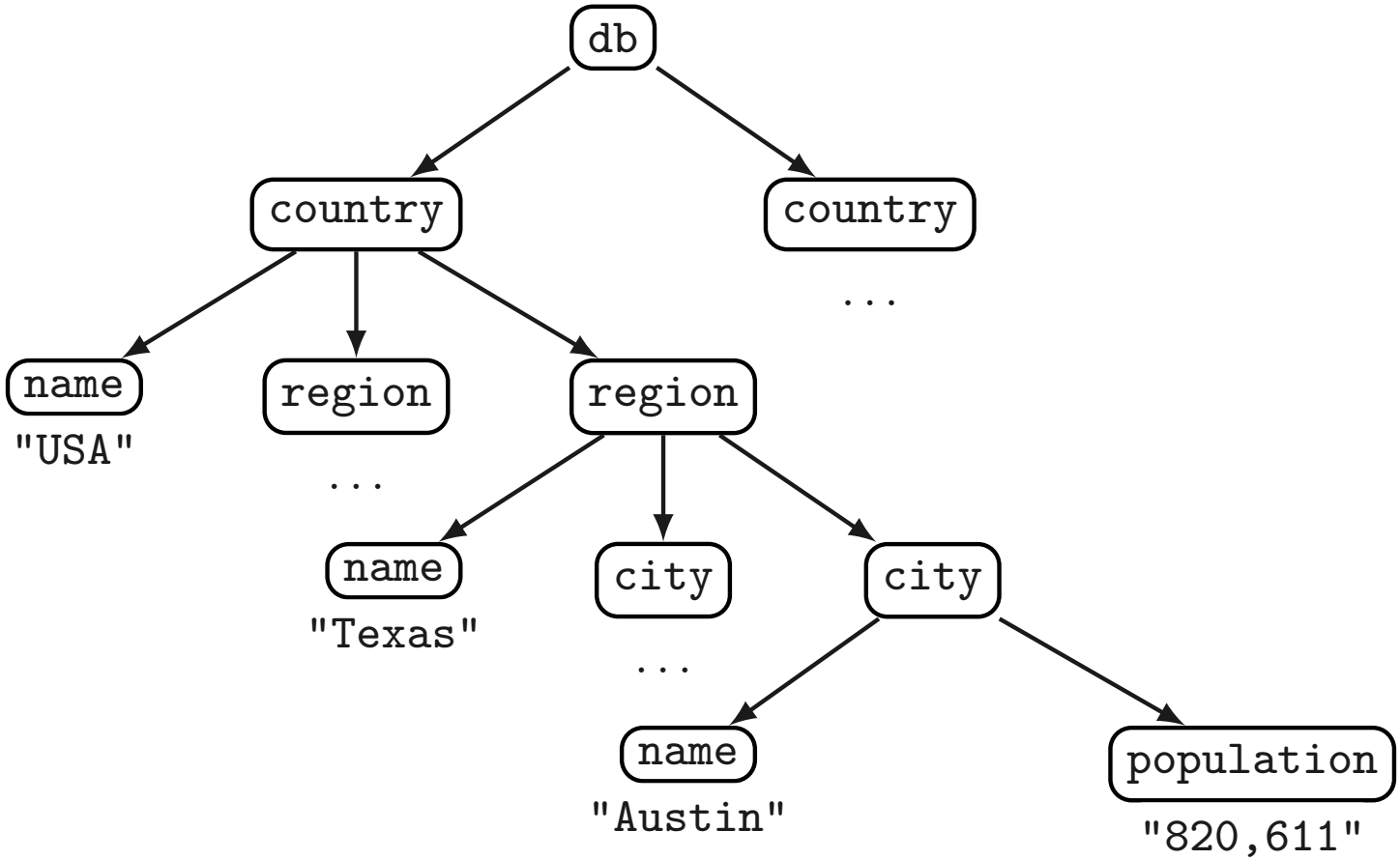
Marcelo Arenas

PUC Chile & U. of Oxford

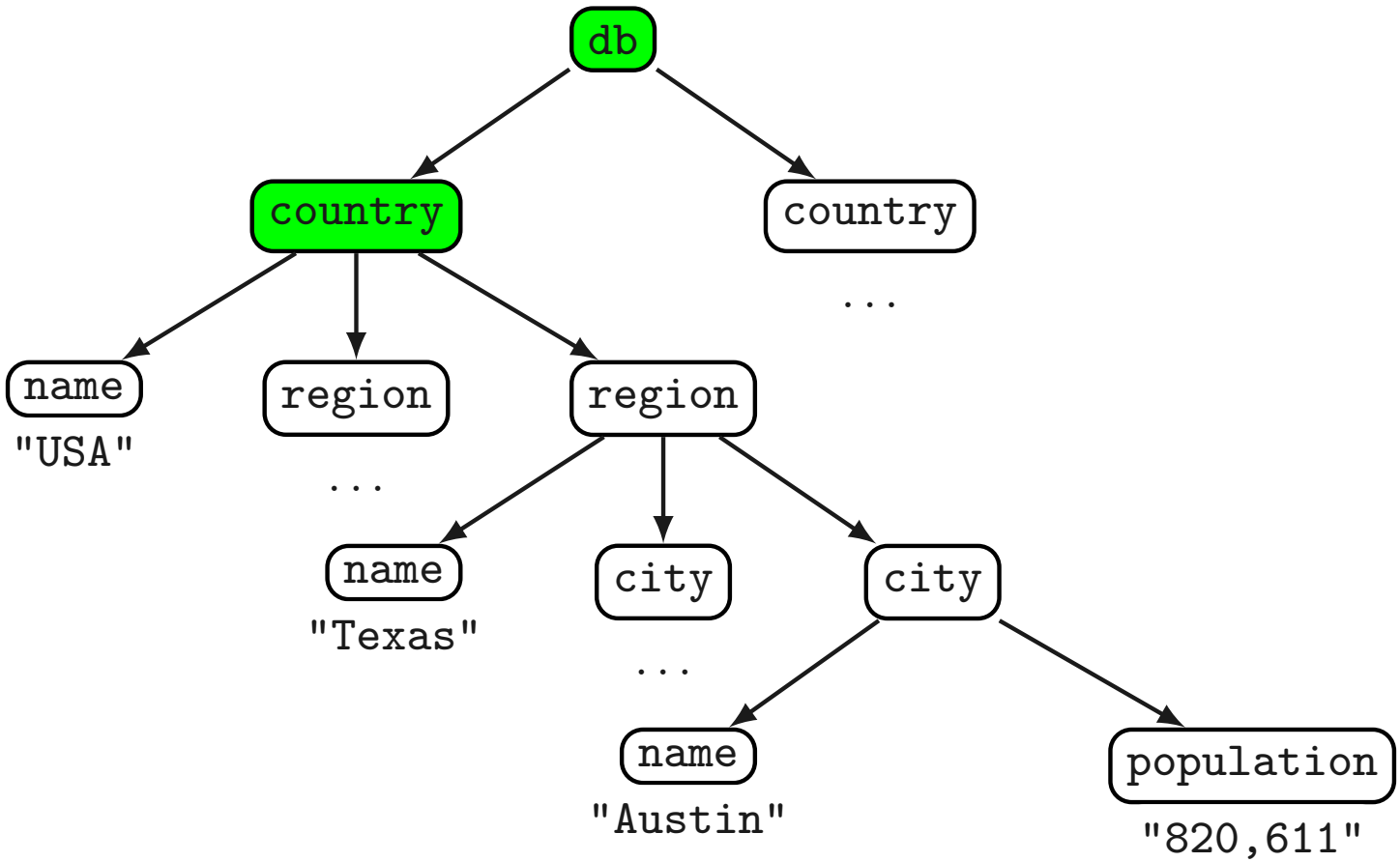
Joint work with Jonny Daenen, Frank Neven, Martin Ugarte, Jan Van den Bussche and Stijn Vansummeren

Oxford, May 2013

XML trees: An example

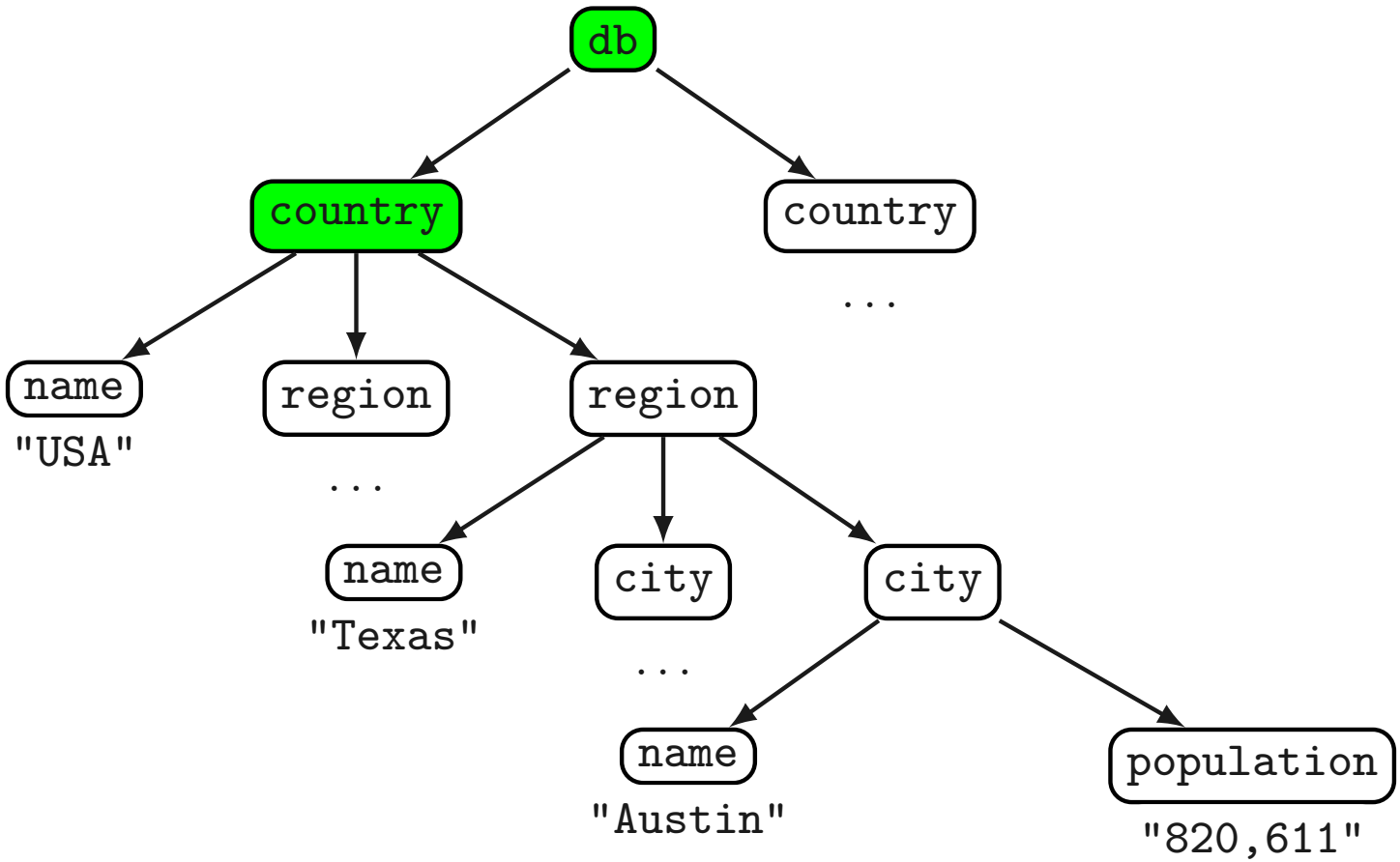


XML schema (XSD): An example



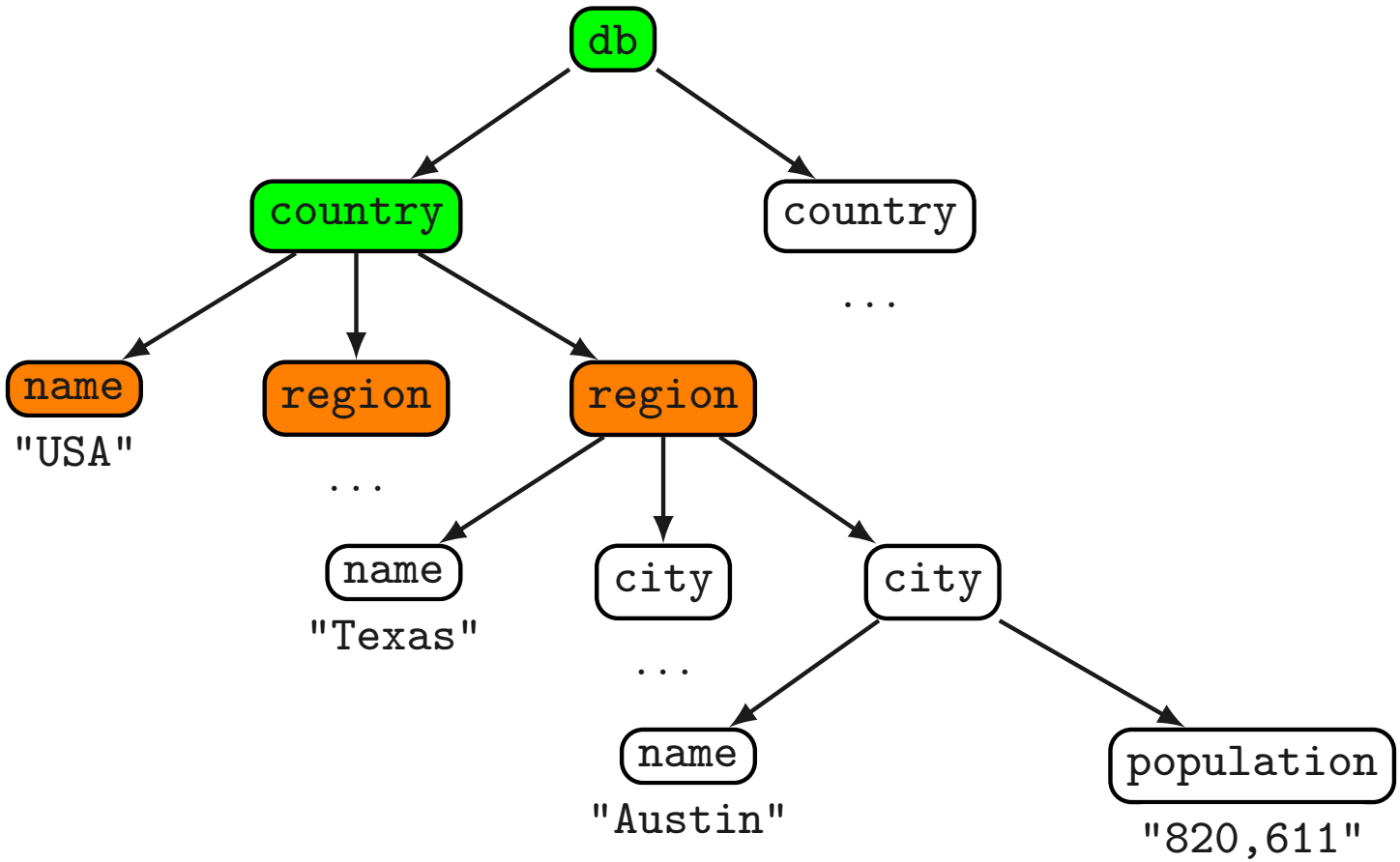
XML schema (XSD): An example

db/country $\mapsto$ name region*



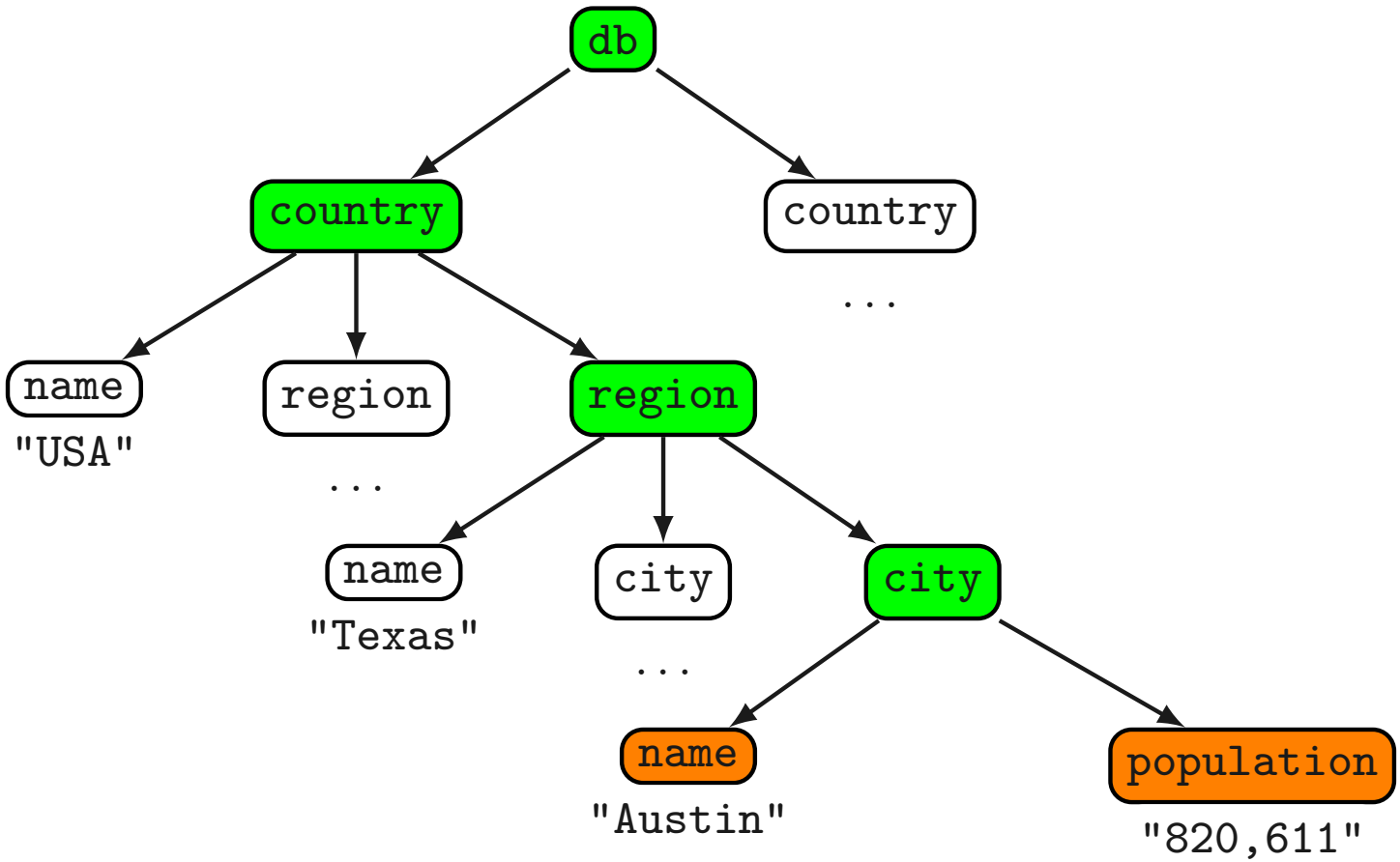
XML schema (XSD): An example

db/country $\mapsto$ name region*



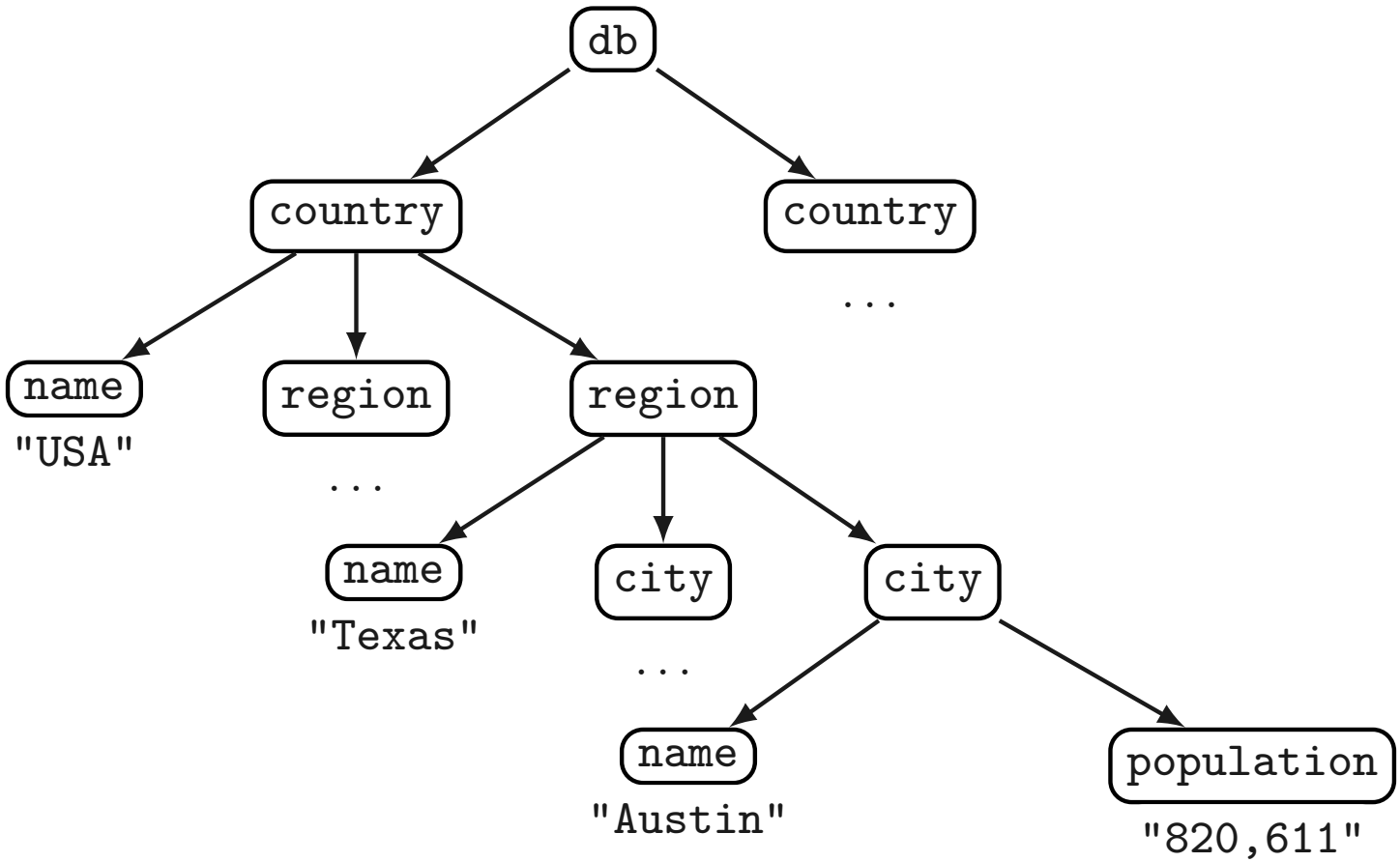
XML schema (XSD): An example

db/country/region/city $\mapsto$ name population



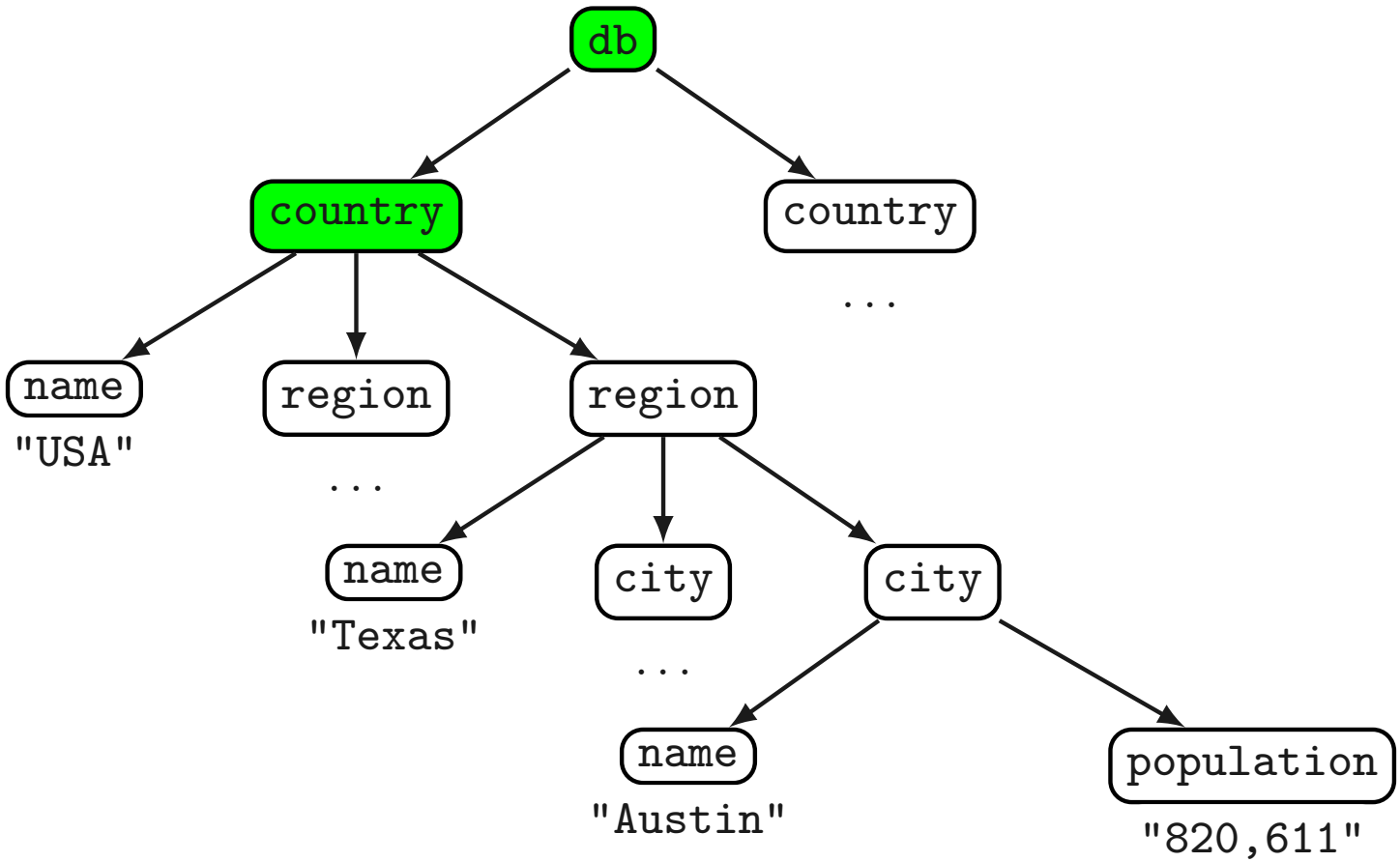
XSD Keys: An example

(db/country, .//city, (./name))



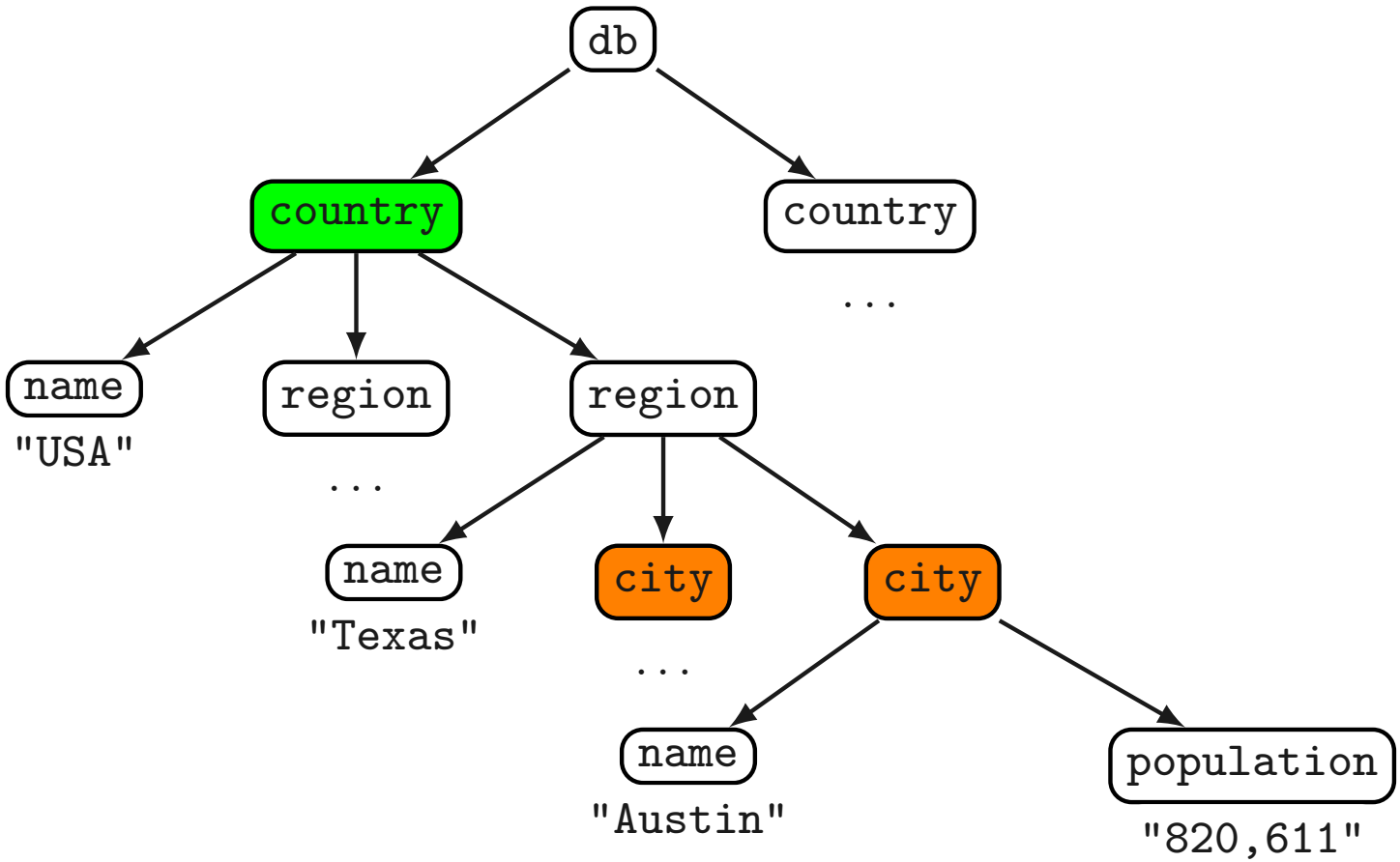
XSD Keys: An example

(db/country, .//city, (./name))



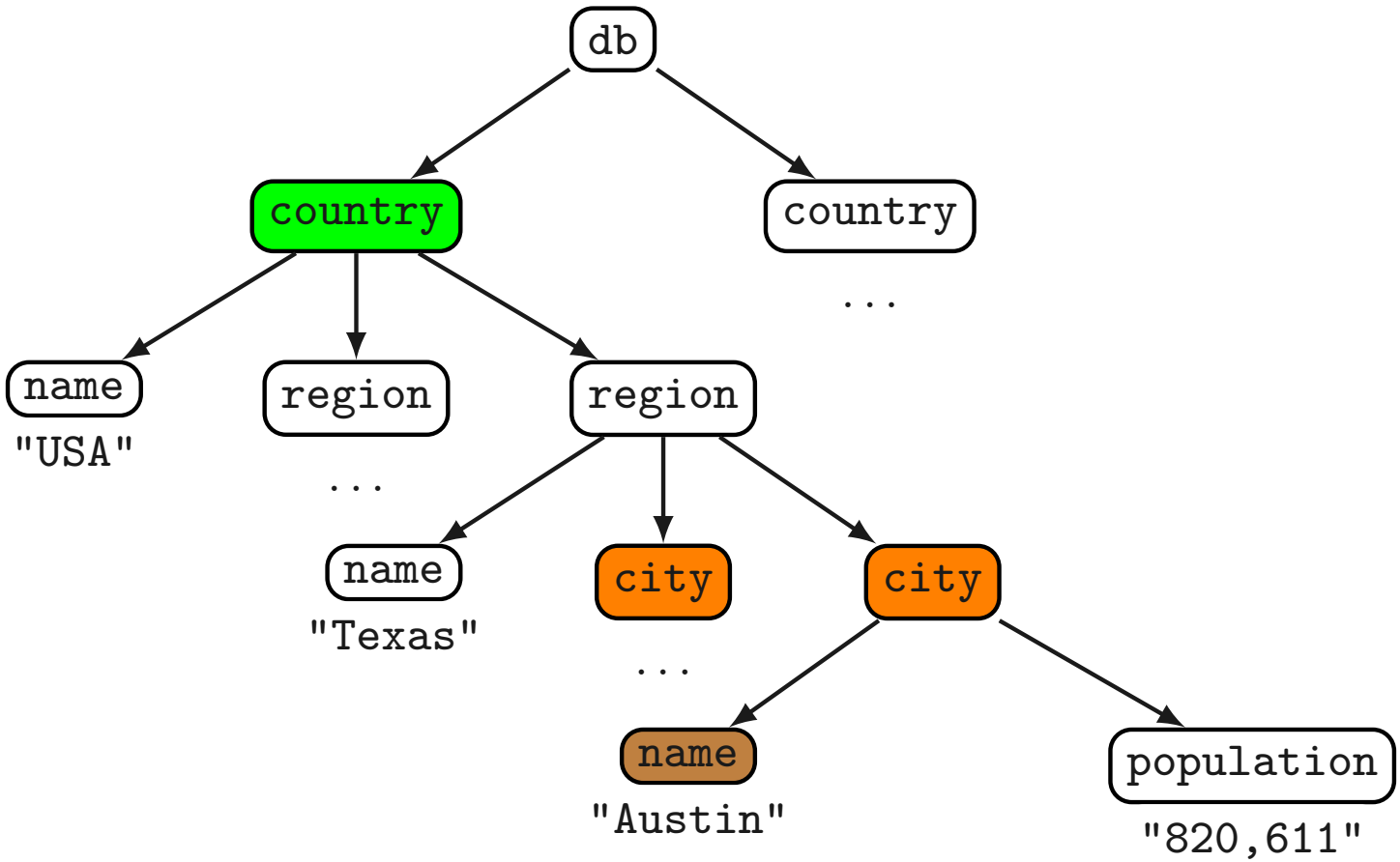
XSD Keys: An example

(db/country, .//city, (./name))



XSD Keys: An example

(db/country, .//city, (./name))

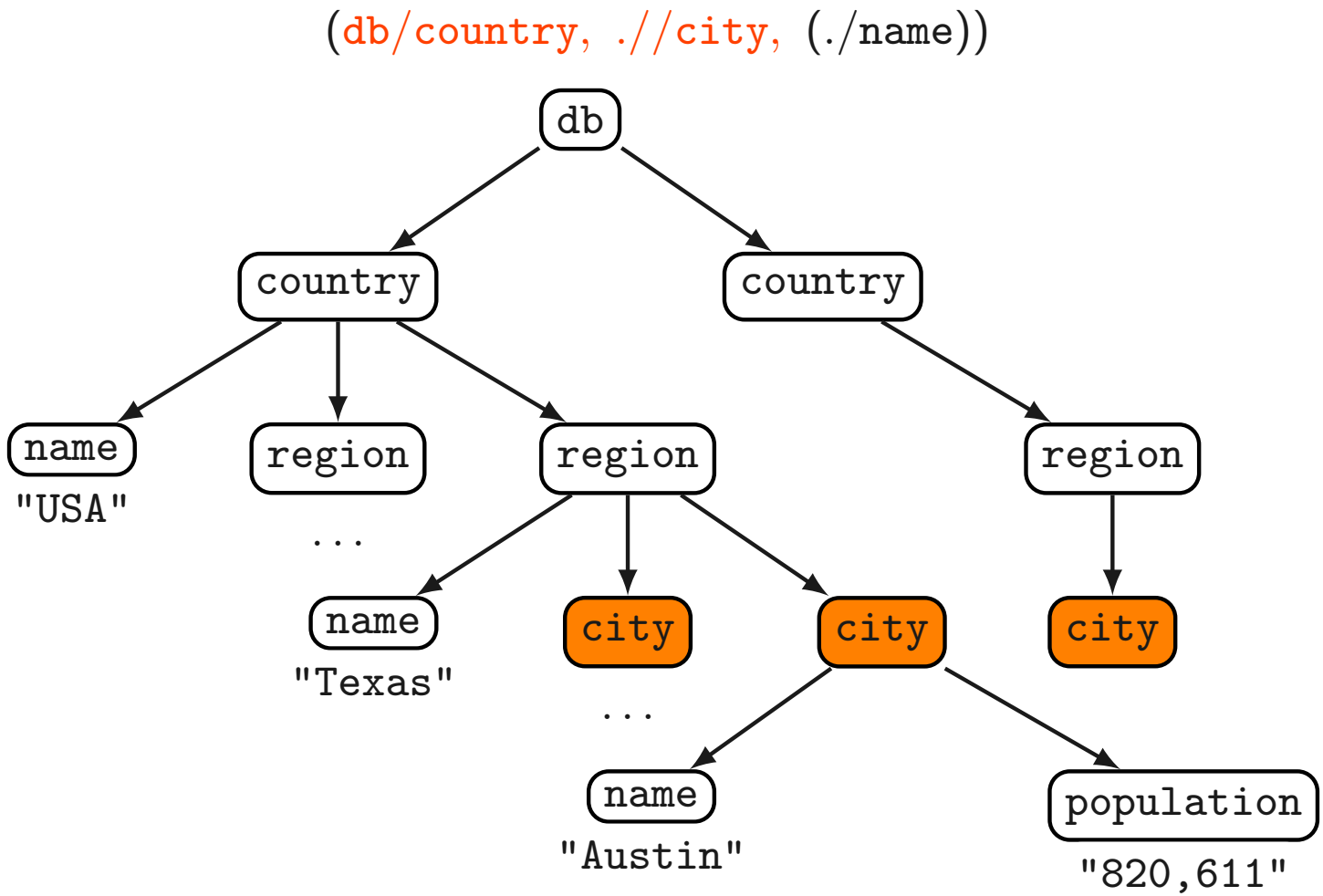


XSD key mining

Support of a key in an XML tree: Number of nodes captured by the key

XSD key mining

Support of a key in an XML tree: Number of nodes captured by the key



XSD key mining (cont'd)

Given an XSD X, an XML tree t satisfying X and a threshold N, we want to find *all* keys ϕ such that:

1. t satisfies ϕ
2. the support of ϕ in t is greater than N
3. ϕ meets some quality requirements (which are defined only with respect to X)

XSD key mining (cont'd)

Important issues:

- Collections of XML trees are not considered, as a collection can always be combined into a single XML tree by introducing a common root
- The standard definition of XML keys (given by the W3C) is considered in this work
- The quality requirements play a fundamental role in this work

Outline

- A bit of notation: XML trees and XSDs
- XSD keys: Formal definition
- XSD key mining problem: Formal definition
- The consistency problem: Complexity
- An XSD key mining algorithm
- Experimental evaluation: A few words
- Take-home message

Outline

- A bit of notation: XML trees and XSDs
- XSD keys: Formal definition
- XSD key mining problem: Formal definition
- The consistency problem: Complexity
- An XSD key mining algorithm
- Experimental evaluation: A few words
- Take-home message

XML trees

Given: Finite set Σ of element names and an infinite set **Data** of data elements.

- Σ, **Data** are assumed to be disjoint

XML trees

Given: Finite set Σ of element names and an infinite set **Data** of data elements.

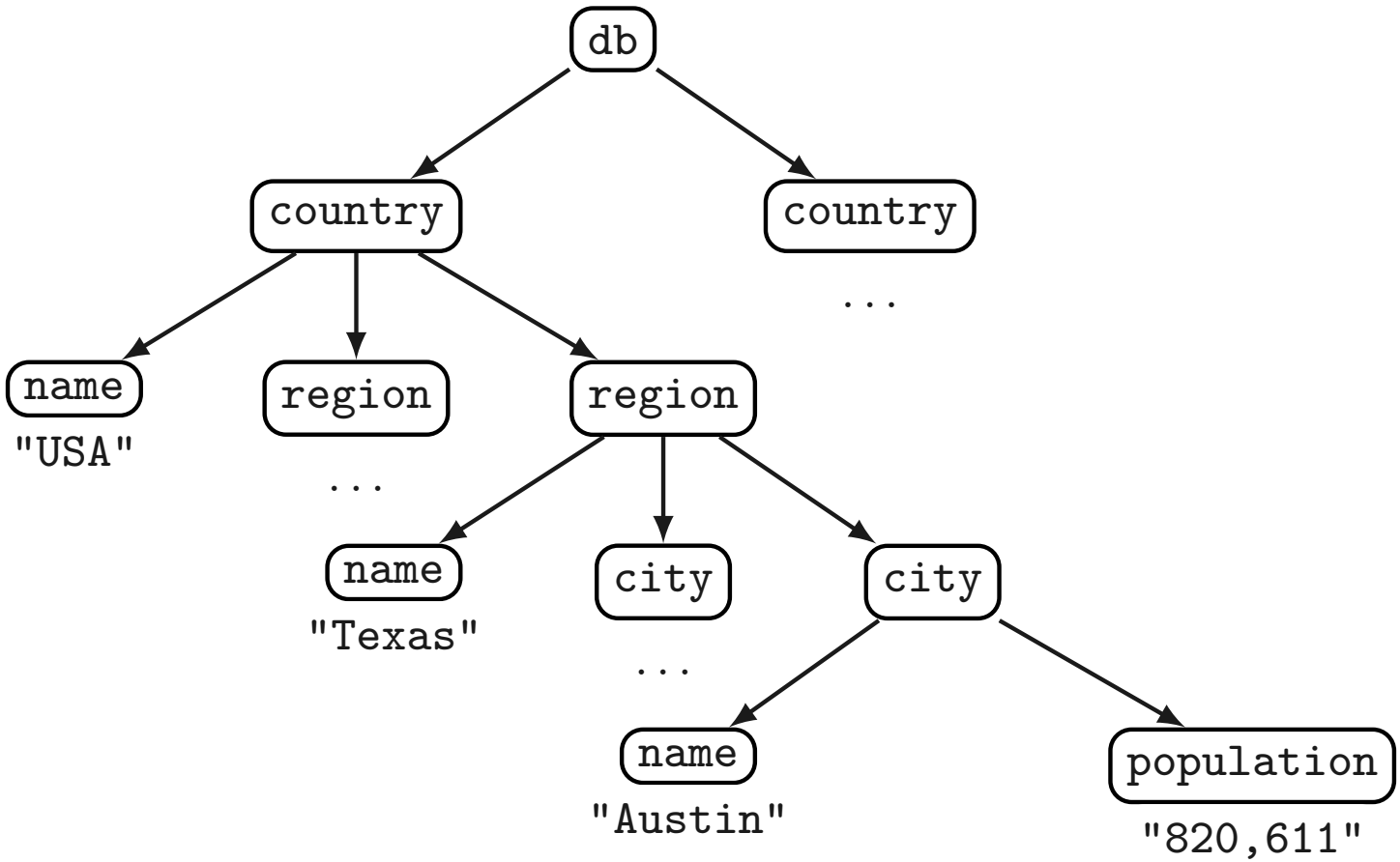
- Σ, **Data** are assumed to be disjoint

XML tree (or just tree): (t, lab_t)

- t: Finite, unranked and sibling-ordered tree
- lab_t: node labeling function with range $(\Sigma \cup \mathbf{Data})$, which labels non-leaf nodes with Σ

Mixed content is not allowed: When a node is labeled with **Data**, then it is the only child of its parent.

XML trees: An example



XSD: Definition

An XSD defines the structure of a collection of XML trees.

Formally: $X = (A, \lambda)$

- $A = (Q, \Sigma \cup \{\texttt{data}\}, q_0, \delta)$ is a DFA without final states
- λ assigns to each state in Q a (one-unambiguous) regular expression over $(\Sigma \cup \{\texttt{data}\})$

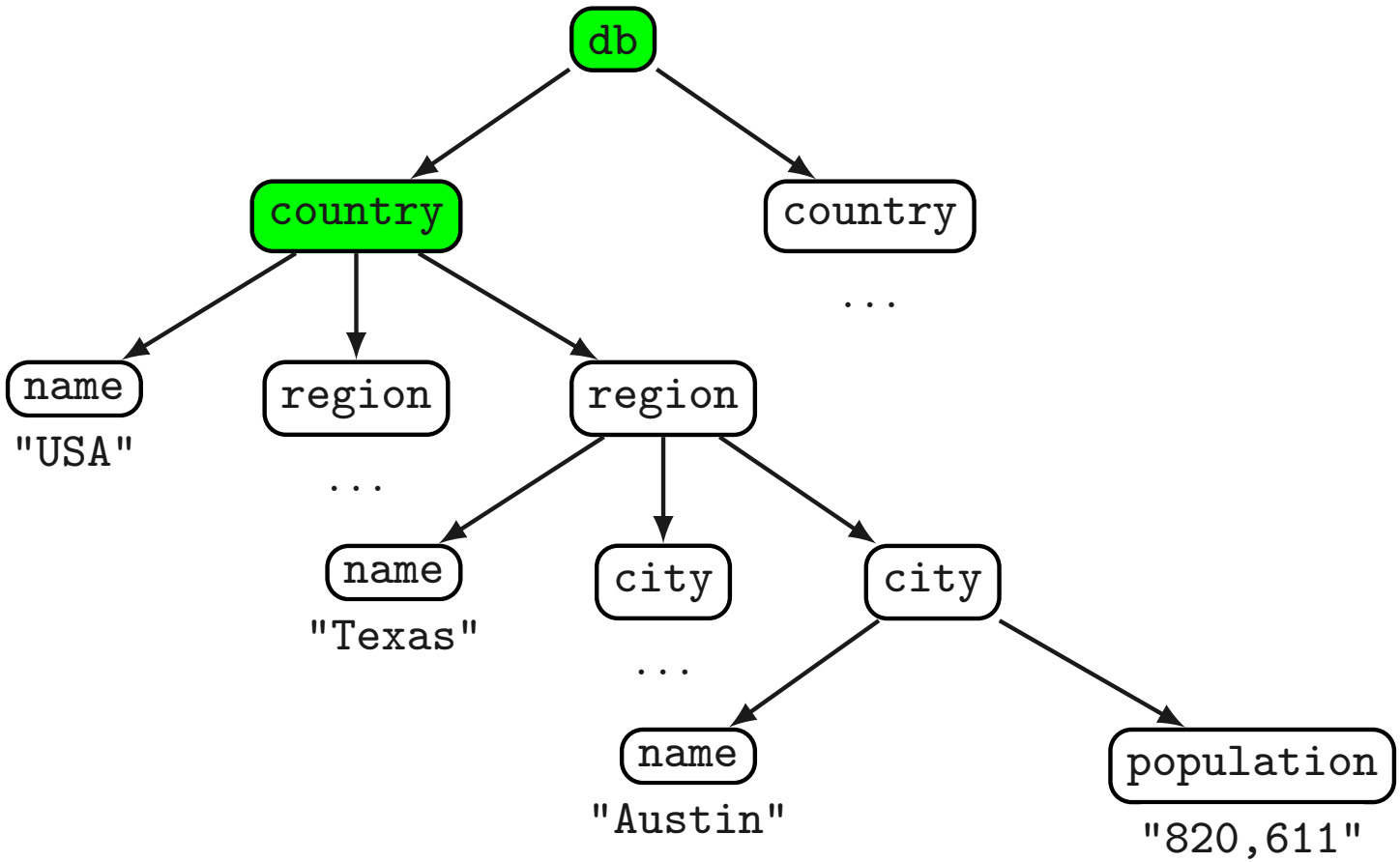
XSD: Definition

An XSD defines the structure of a collection of XML trees.

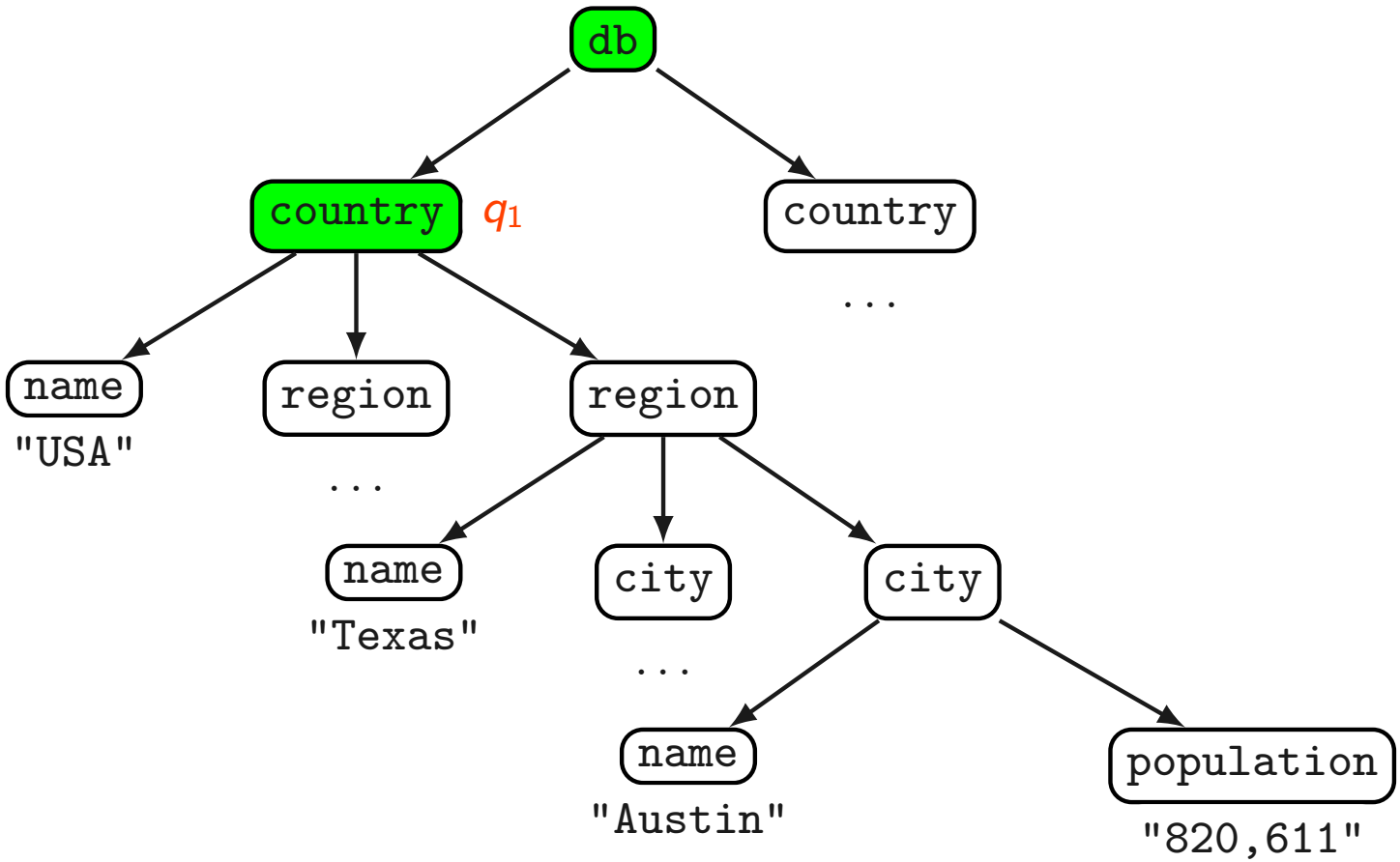
Formally: $X = (A, \lambda)$

- $A = (Q, \Sigma \cup \{\texttt{data}\}, q_0, \delta)$ is a DFA without final states
- λ assigns to each state in Q a (one-unambiguous) regular expression over $(\Sigma \cup \{\texttt{data}\})$
  - There exists a polynomial-time algorithm that, given $q \in Q$, generates a DFA accepting $\lambda(q)$

Conforming to an XSD

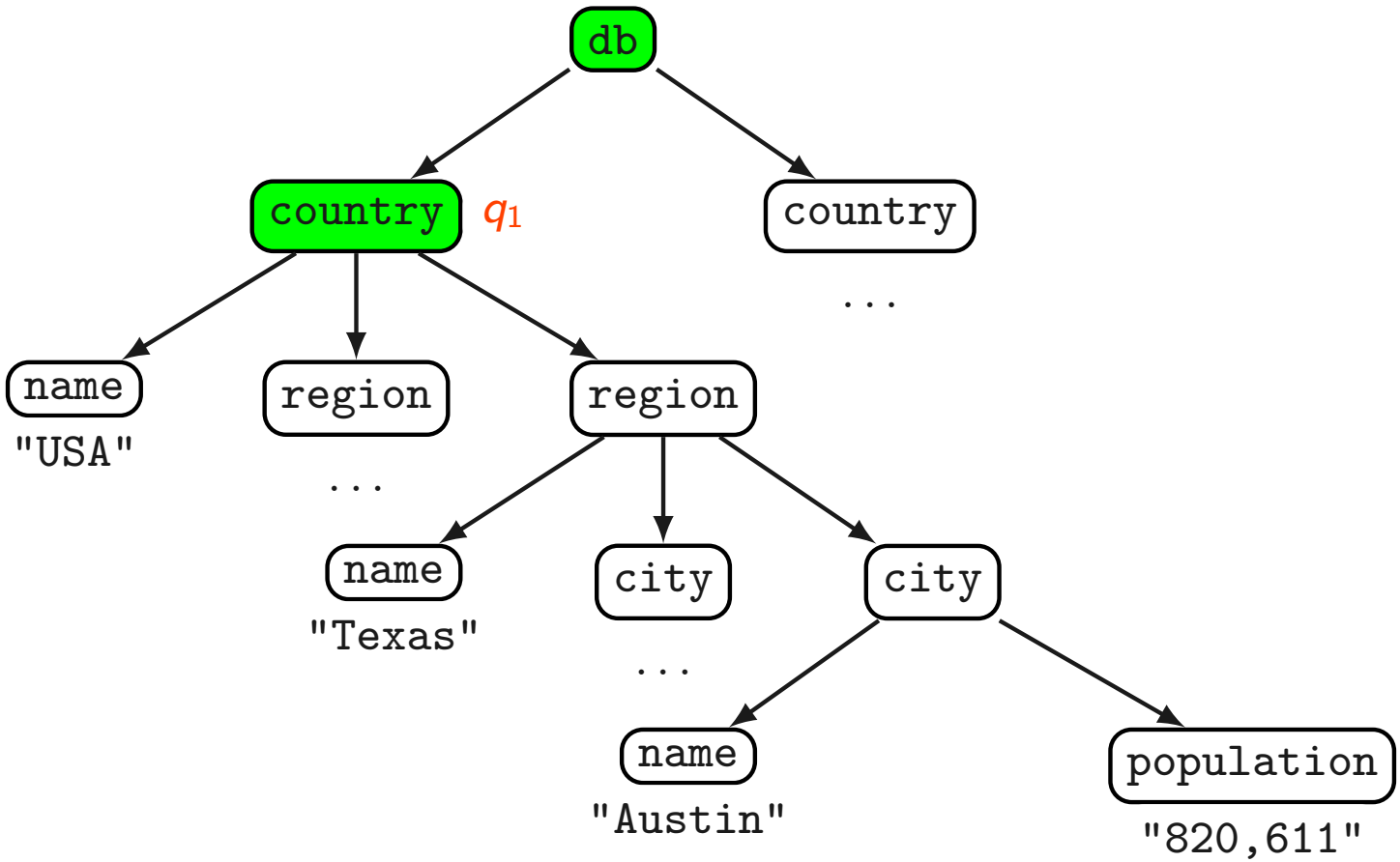


Conforming to an XSD



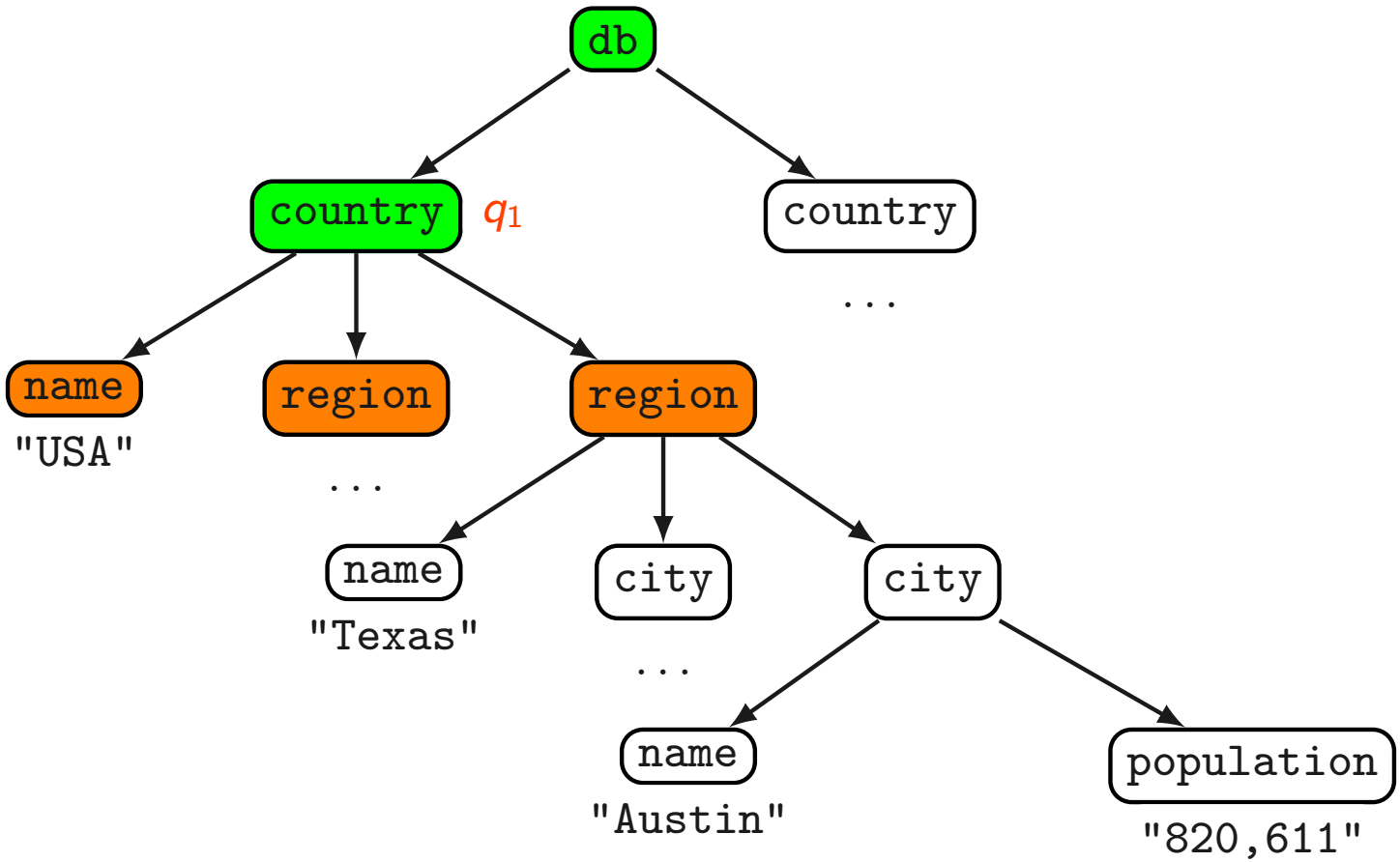
Conforming to an XSD

$\lambda(q_1) = \texttt{name region}^*$

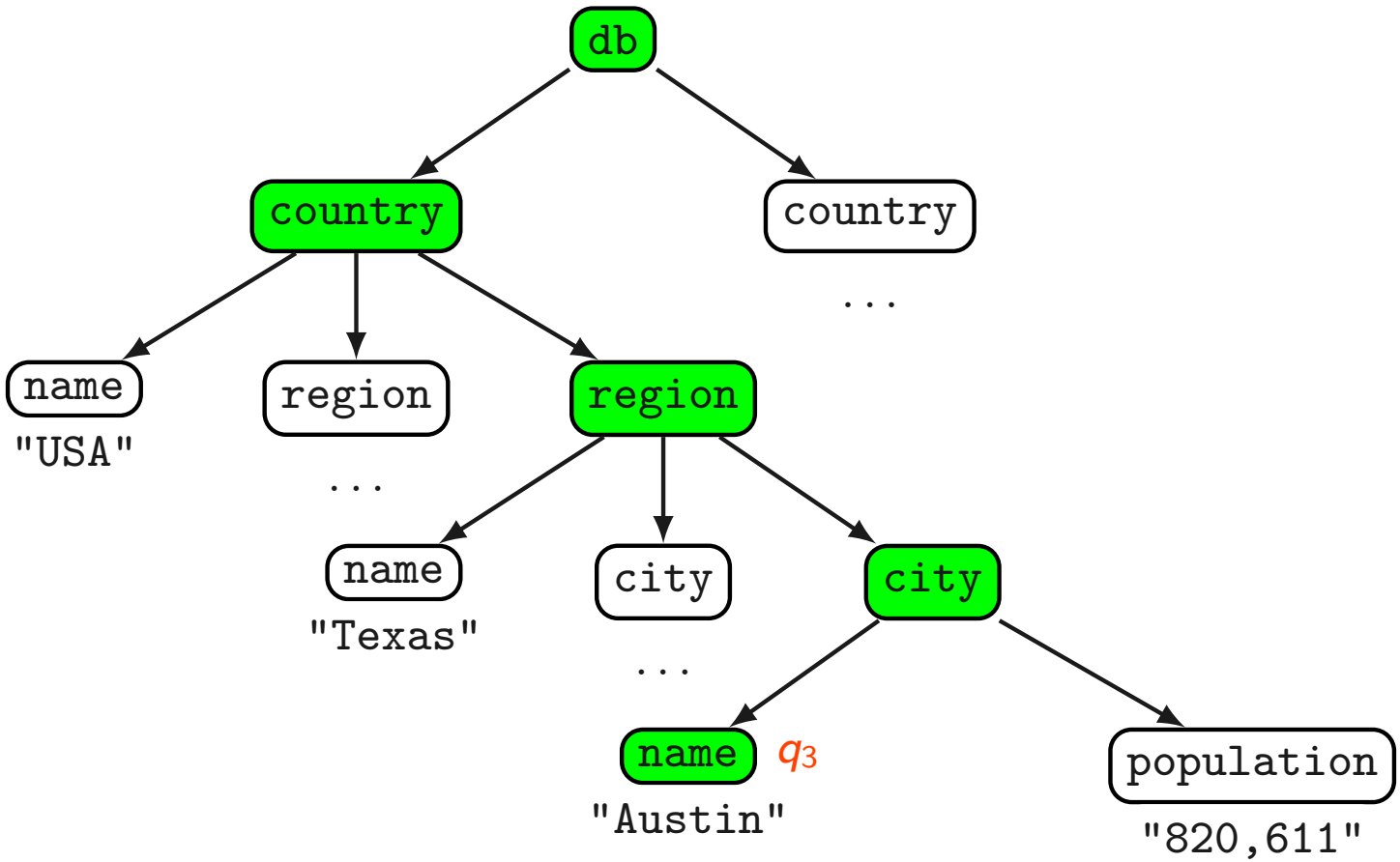


Conforming to an XSD

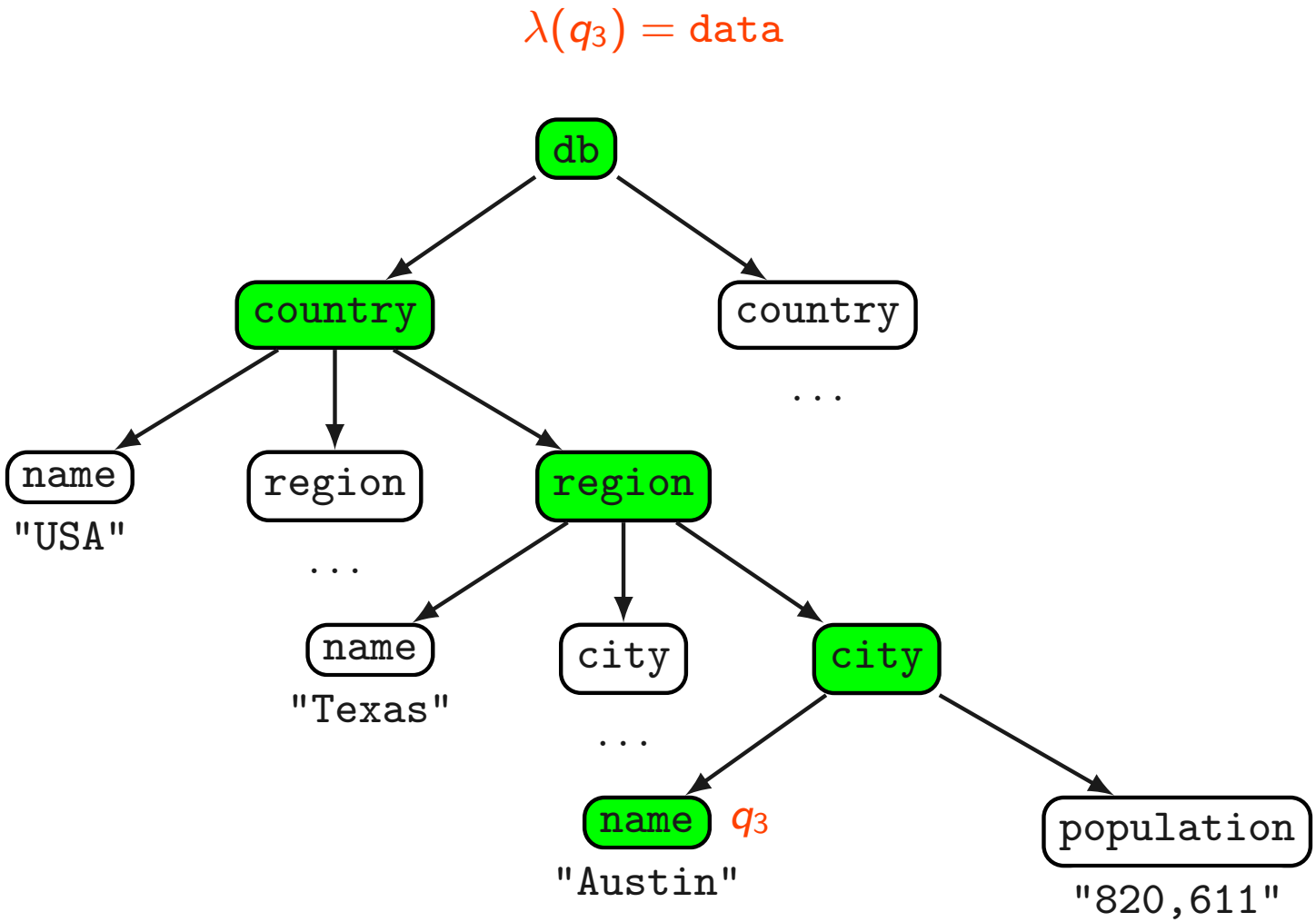
$\lambda(q_1) = \texttt{name region}^*$



Conforming to an XSD



Conforming to an XSD



Outline

- A bit of notation: XML trees and XSDs
- XSD keys: Formal definition
- XSD key mining problem: Formal definition
- The consistency problem: Complexity
- An XSD key mining algorithm
- Experimental evaluation: A few words
- Take-home message

Defining XSD keys: Contexts

Let $X = (A, \lambda)$ be an XSD, where Q is the set of states of A

A context $c = (q, a)$ is a pair in $Q \times \Sigma$

Defining XSD keys: Contexts

Let $X = (A, \lambda)$ be an XSD, where Q is the set of states of A

A context $c = (q, a)$ is a pair in $Q \times \Sigma$

Given an XML tree t, $\text{CNodes}_t(c)$ is the set of nodes v in t such that

1. $\text{lab}_t(v) = a$
2. A halts in state q on input the string formed by the labels on the path from t's root to v

Defining XSD keys: Selector expressions

A selector expression is of one of the following three forms:

- . (the dot symbol)
- $./a_1/\cdots/a_k$ (starting with the child axis), where $k \geq 1$ and for every $i \in \{1, \ldots, k\}$, either $a_i \in \Sigma$ or $a_i = *$
- $.//a_1/\cdots/a_k$ (starting with the descendant axis), where $k \geq 1$ and for every $i \in \{1, \ldots, k\}$, either $a_i \in \Sigma$ or $a_i = *$

A union of selector expressions is of the form $(\tau_1|\cdots|\tau_m)$, where $m \geq 1$ and each τ_i is a selector expression.

$\mathcal{SE}$ denotes the class of selector expressions and $\mathcal{DSE}$ the class of union of selector expressions.

Selector expressions: Semantics

Let $w = w_1 \dots w_n$ be a string, where each $w_i \in \Sigma$.

- w is said to match $./a_1/\cdots/a_k$ if $n = k$ and for every $i \in \{1, \dots, k\}$: $w_i = a_i$ or $a_i = *$
- w is said to match $.//a_1/\cdots/a_k$ if a suffix of w matches $./a_1/\cdots/a_k$

Selector expressions: Semantics (cont'd)

Let t be an XML tree, v a node in t and τ a selector expression.

$\tau(t, v)$: Nodes in t that are reachable from v by following τ.

Selector expressions: Semantics (cont'd)

Let t be an XML tree, v a node in t and τ a selector expression.

$\tau(t, v)$: Nodes in t that are reachable from v by following τ.

- $\tau(t, v) = \{v\}$ if $\tau = .$
- Otherwise, $\tau(t, v)$ contains every node v' such that
  - v' is a descendant of v in t, and
  - the path of labels from v to v' (but excluding the label of v) matches τ

Selector expressions: Semantics (cont'd)

Let t be an XML tree, v a node in t and τ a selector expression.

$\tau(t, v)$: Nodes in t that are reachable from v by following τ.

- $\tau(t, v) = \{v\}$ if $\tau = .$
- Otherwise, $\tau(t, v)$ contains every node v' such that
  - v' is a descendant of v in t, and
  - the path of labels from v to v' (but excluding the label of v) matches τ

If $\tau' = (\tau_1 | \cdots | \tau_m)$ is a disjunction of selector expressions:

$$\tau'(t, v) \quad = \quad \tau_1(t, v) \cup \cdots \cup \tau_m(t, v)$$

XSD keys: Definition

Definition

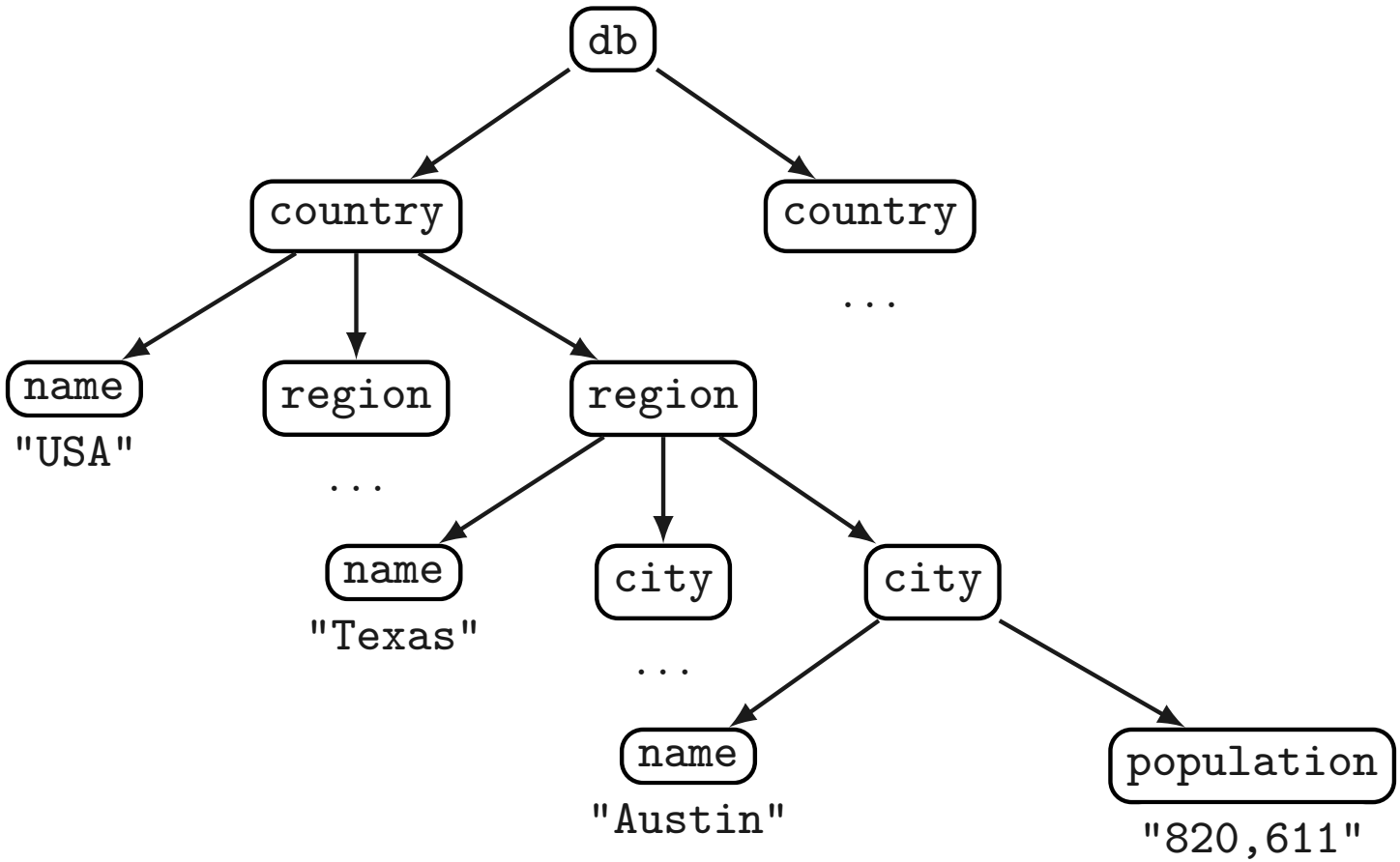
An XSD key, defined w.r.t. an XSD X, is a tuple $\phi = (c, \tau, P)$:

- c is a context in X,
- $\tau \in \mathcal{DSE}$, and
- $P = (p_1, \ldots, p_k)$, where $k \geq 1$ and each $p_i \in \mathcal{DSE}$

Notation: τ is the *target* path and each p_i is a *key* path

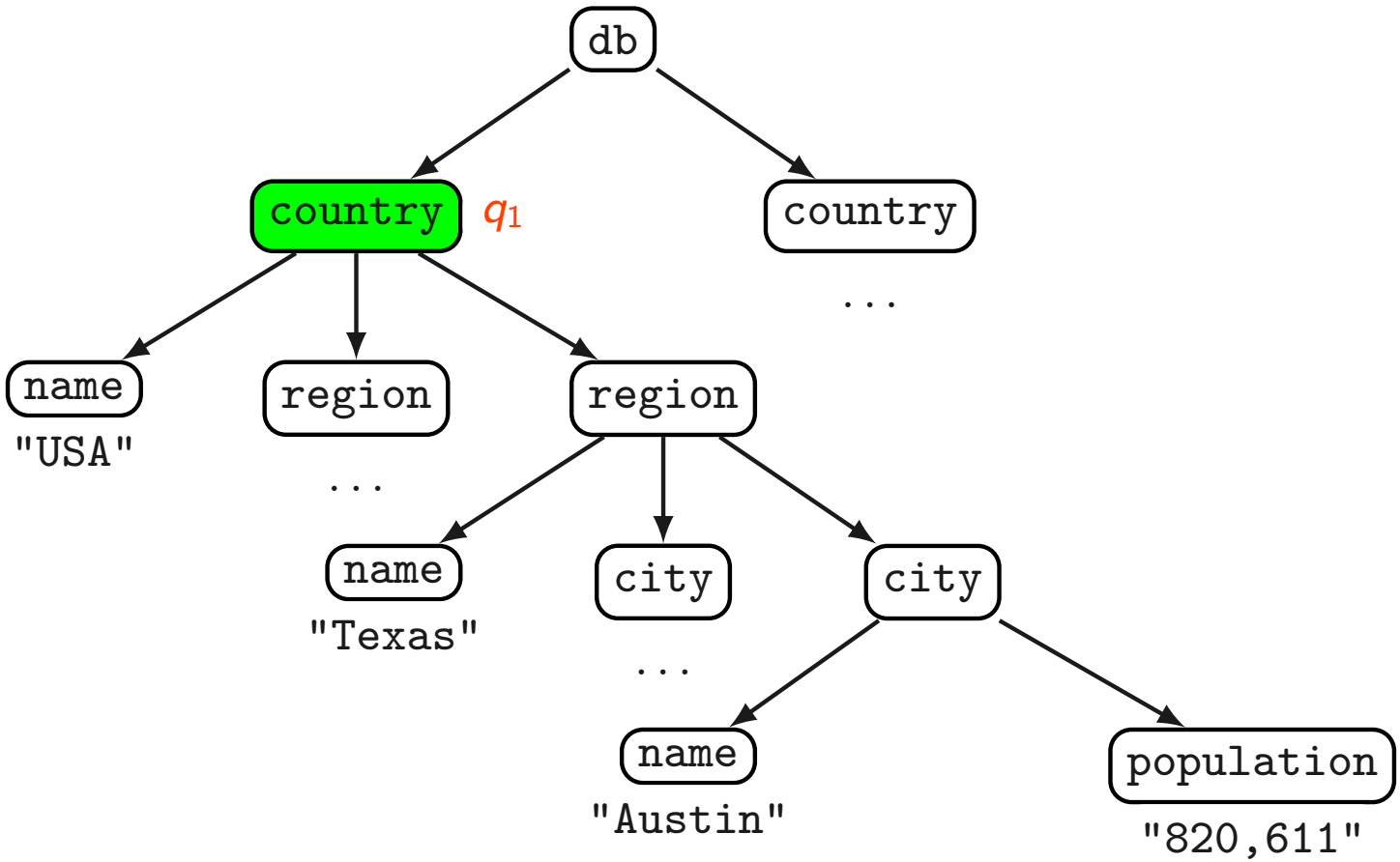
XSD Keys: Some examples

$((\texttt{country}, q_1),\ .//\texttt{city},\ (./\texttt{name}))$



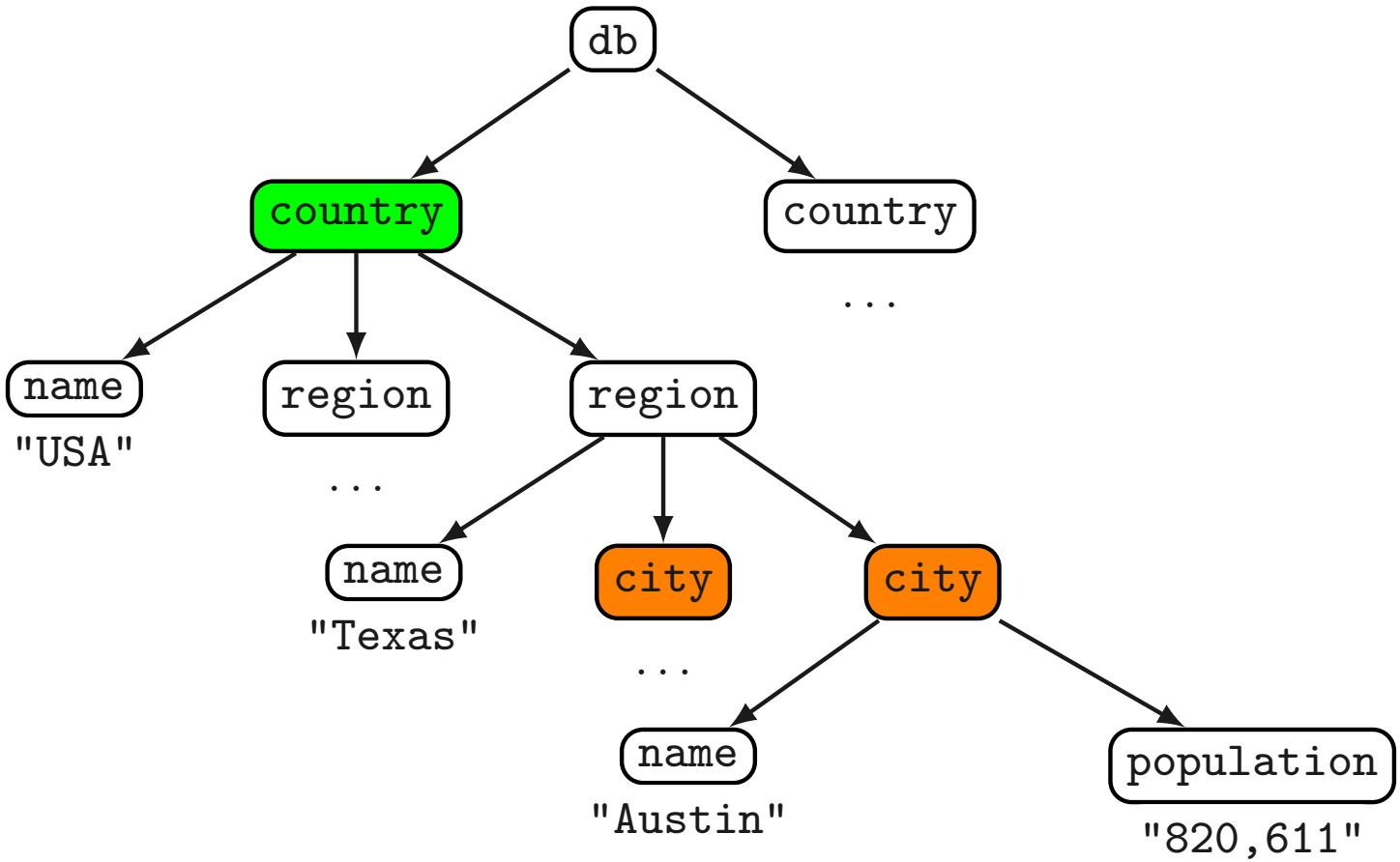
XSD Keys: Some examples

$((\texttt{country}, q_1),\ .//\texttt{city},\ (./\texttt{name}))$



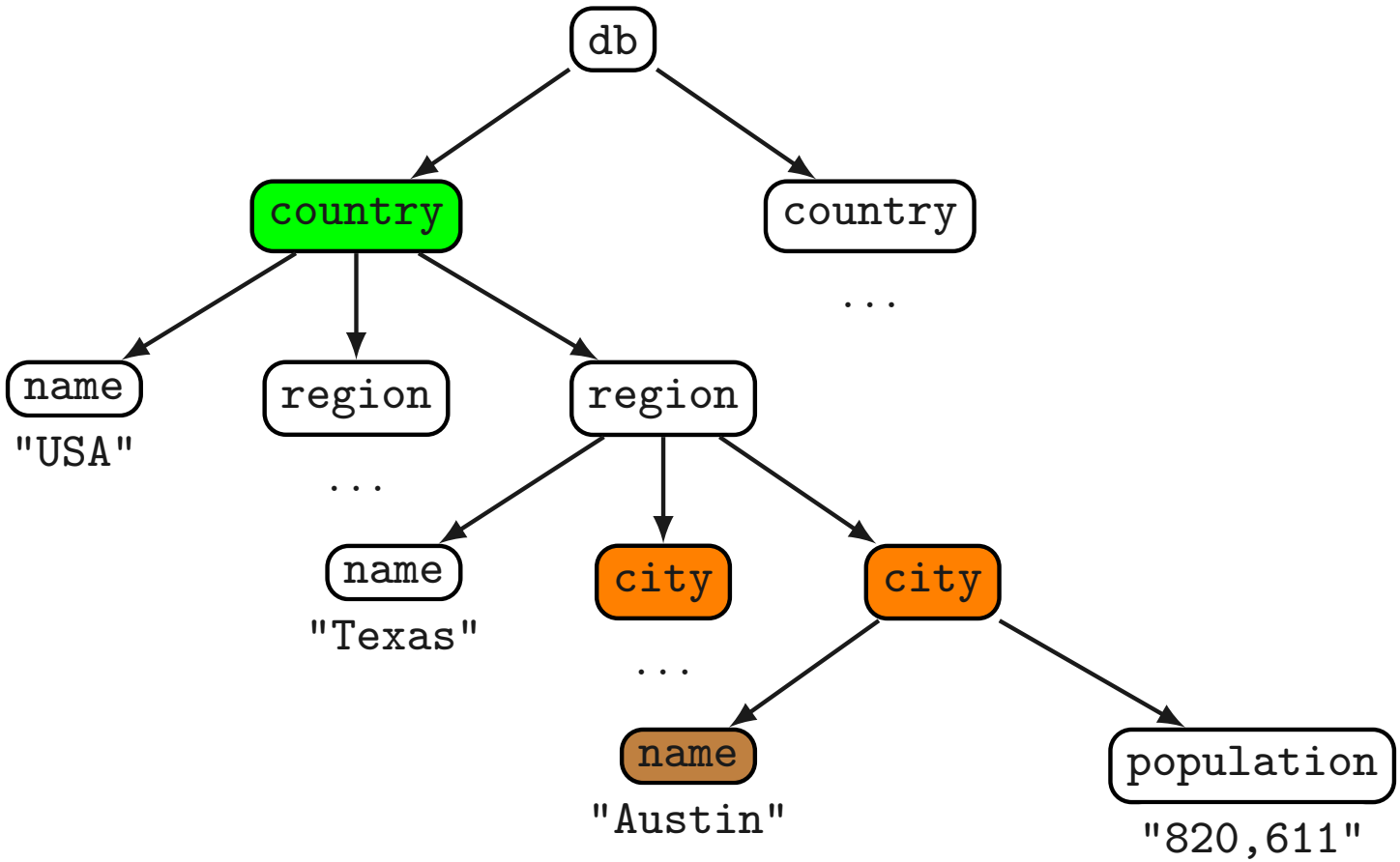
XSD Keys: Some examples

$((\texttt{country}, q_1),\ \texttt{.//city},\ (\texttt{./name}))$



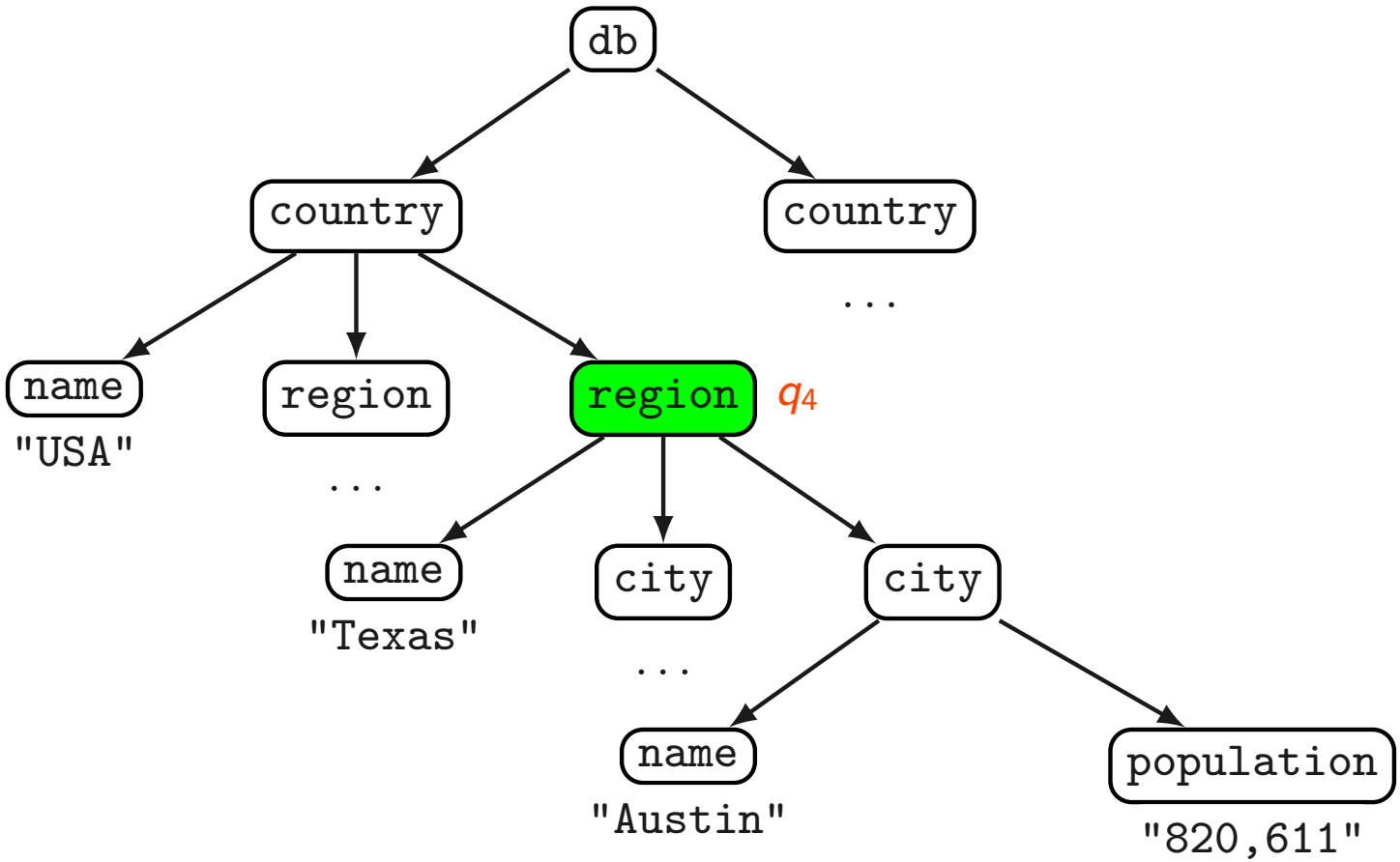
XSD Keys: Some examples

$((\texttt{country}, q_1),\ .//\texttt{city},\ (./\texttt{name}))$



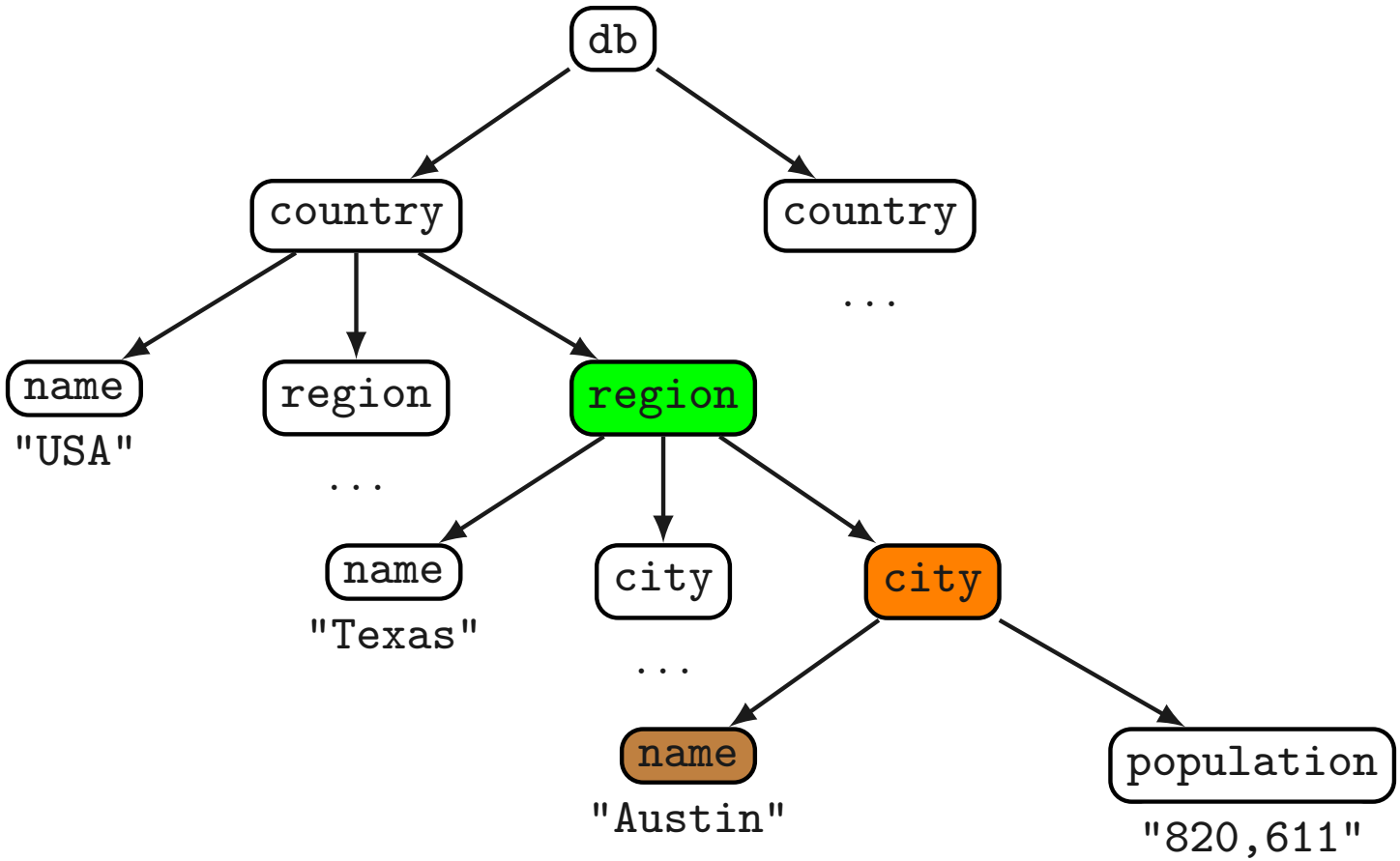
XSD Keys: Some examples

$((\texttt{region}, q_4),\ \texttt{.//city},\ (\texttt{./name}))$



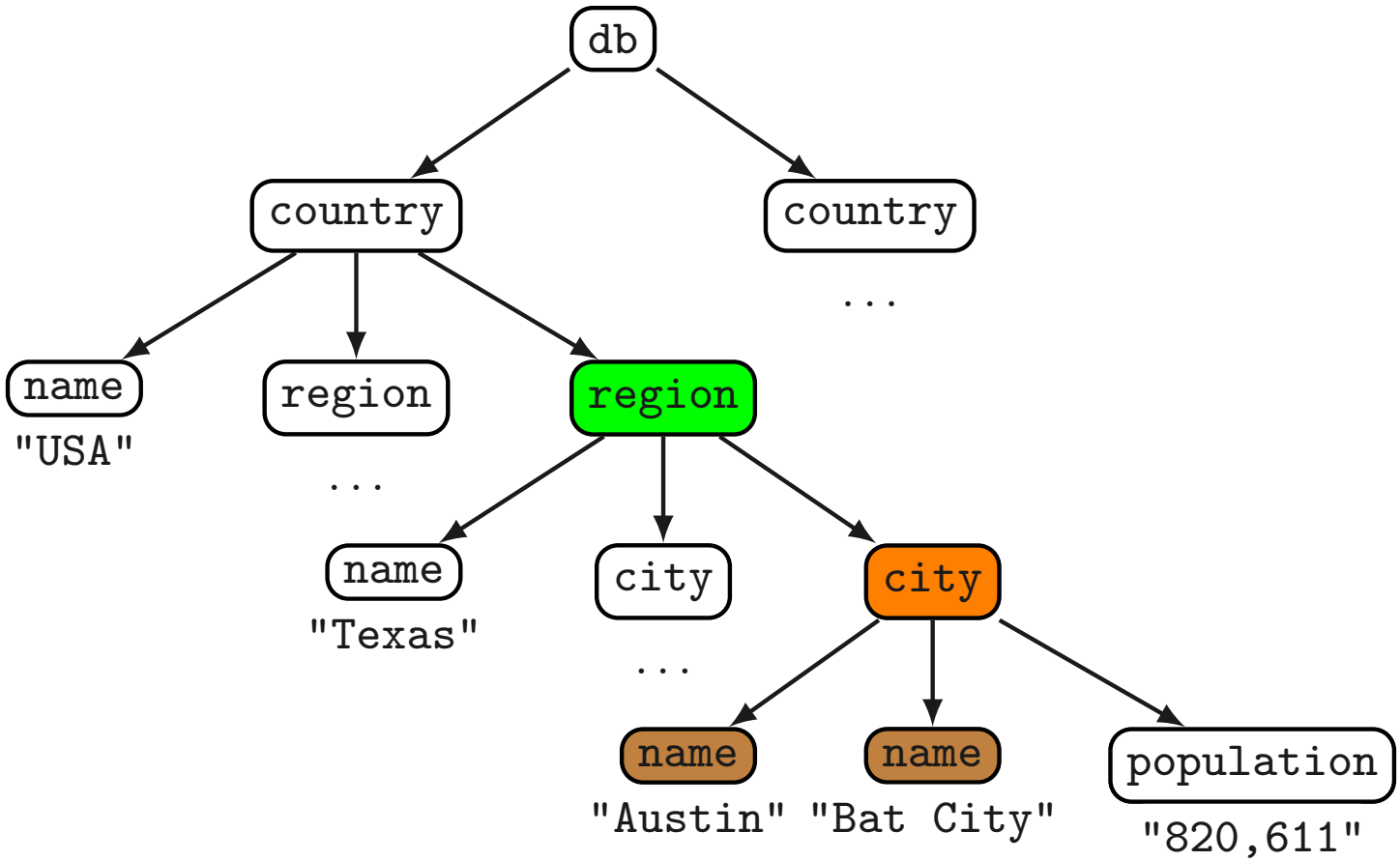
Satisfaction of XSD keys: Structural requirements

$((\texttt{region}, q_4),\ .//\texttt{city},\ (./\texttt{name}))$



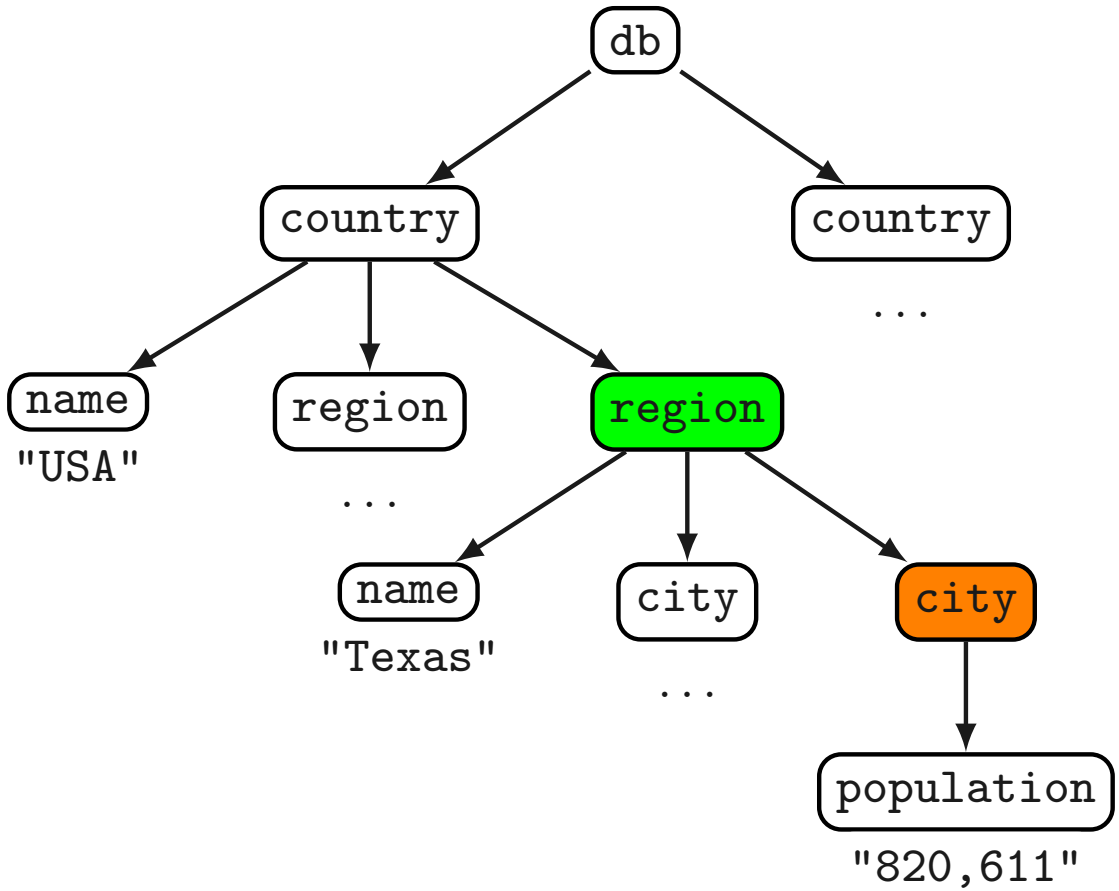
Satisfaction of XSD keys: Structural requirements

$((\texttt{region}, q_4),\ .//\texttt{city},\ (./\texttt{name}))$



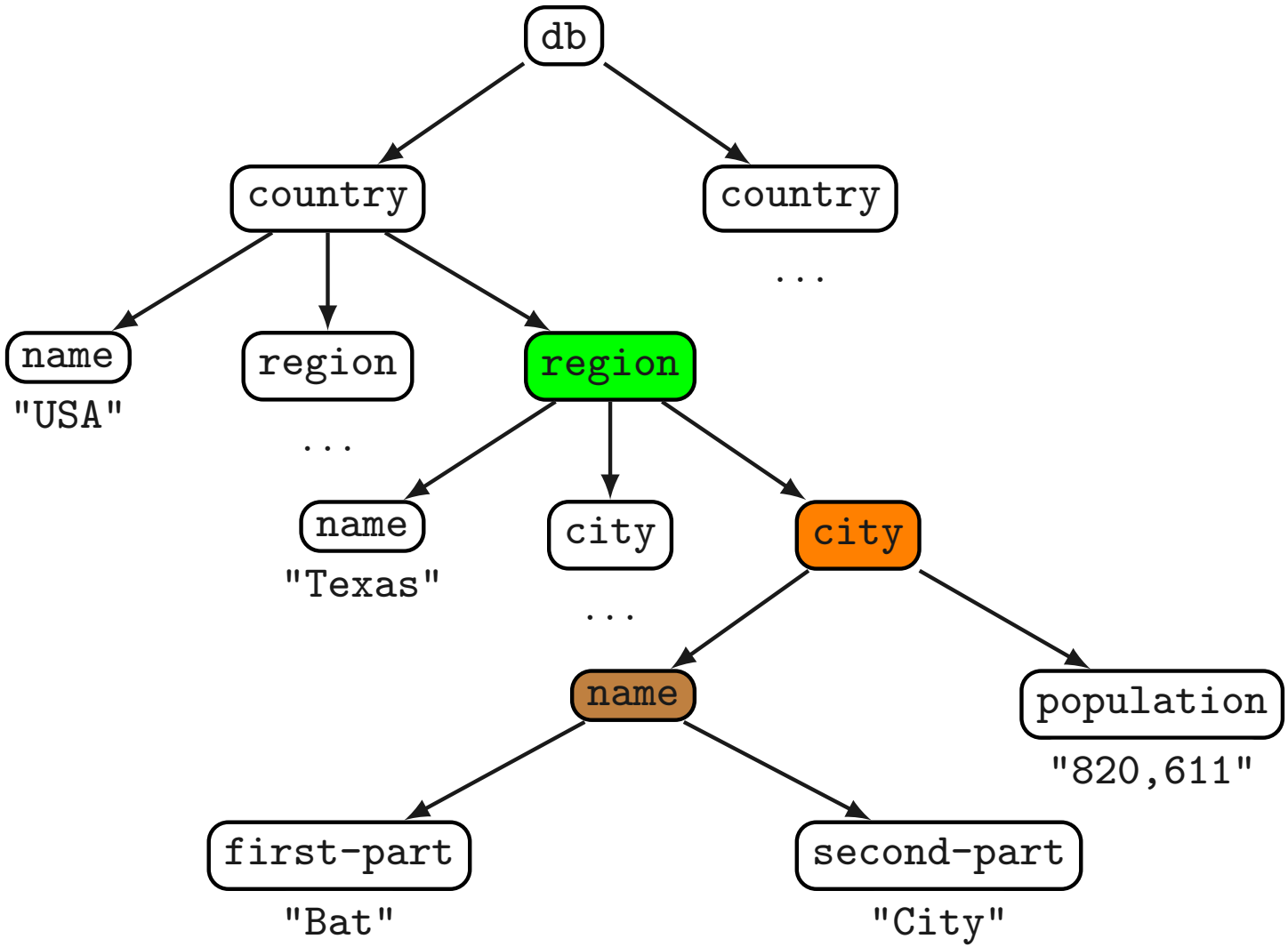
Satisfaction of XSD keys: Structural requirements

$((\texttt{region}, q_4),\ .//\texttt{city},\ (./\texttt{name}))$



Satisfaction of XSD keys: Structural requirements

$((\texttt{region}, q_4),\ .//\texttt{city},\ (./\texttt{name}))$



Satisfaction of XSD keys: Formalizing the structural requirements

v is a **Data**-node in an XML tree t if v has a single child u, and $\mathsf{lab}_t(u) \in \mathbf{Data}$.

- Then $\mathsf{value}_t(v) = \mathsf{lab}_t(u)$

Satisfaction of XSD keys: Formalizing the structural requirements

v is a **Data**-node in an XML tree t if v has a single child u, and $\mathsf{lab}_t(u) \in \mathbf{Data}$.

- Then $\mathsf{value}_t(v) = \mathsf{lab}_t(u)$

Definition

A key $\phi = (c, \tau, (p_1, \ldots, p_k))$ qualifies in an XML tree t if for every $v \in \mathsf{CNodes}_t(c)$, every $u \in \tau(t, v)$ and every p_i:

$p_i(t, u)$ is a singleton containing a **Data**-node

Satisfaction of XSD keys: Complete definition

Let u be a node in an XML tree t, and $P = (p_1, \ldots, p_k)$ a sequence of disjunctions of selector expressions.

Assume that $p_i(t, u) = \{u_i\}$ with u_i a **Data**-node, for every $i \in \{1, \ldots, k\}$.

- Then $\text{record}_P(t, u) = [\text{value}_t(u_1), \ldots, \text{value}_t(u_k)]$

Satisfaction of XSD keys: Complete definition

Let u be a node in an XML tree t, and $P = (p_1, \ldots, p_k)$ a sequence of disjunctions of selector expressions.

Assume that $p_i(t, u) = \{u_i\}$ with u_i a **Data**-node, for every $i \in \{1, \ldots, k\}$.

- Then $\text{record}_P(t, u) = [\text{value}_t(u_1), \ldots, \text{value}_t(u_k)]$

Definition

An XML tree t satisfies a key $\phi = (c, \tau, P)$, denoted by $t \models \phi$, if:

1. ϕ qualifies in t
2. for every $v \in \text{CNodes}_t(c)$ and every pair u_1, u_2 of distinct nodes in $\tau(t, v)$, it holds that $\text{record}_P(t, u_1) \neq \text{record}_P(t, u_2)$

Outline

- A bit of notation: XML trees and XSDs
- XSD keys: Formal definition
- XSD key mining problem: Formal definition
- The consistency problem: Complexity
- An XSD key mining algorithm
- Experimental evaluation: A few words
- Take-home message

XSD key mining problem: Definition

First, we need to define the support of a key in an XML tree.

Assume that $\phi = (c, \tau, P)$. Given an XML tree t:

$$\mathsf{TNodes}_t(\phi) \quad = \quad \bigcup_{v \in \mathsf{CNodes}_t(c)} \tau(t, v)$$

Then $\mathsf{supp}(\phi, t) = |\mathsf{TNodes}_t(\phi)|$

- $\mathsf{supp}(c, \tau, t)$ is also used to denote $\mathsf{supp}(\phi, t)$

XSD key mining problem: Definition (cont'd)

Second, we need to introduce a fundamental quality requirement:

Definition

A key ϕ is consistent w.r.t. an XSD X if ϕ qualifies in every XML tree conforming to X

XSD key mining problem: Definition (cont'd)

Second, we need to introduce a fundamental quality requirement:

Definition

A key ϕ is consistent w.r.t. an XSD X if ϕ qualifies in every XML tree conforming to X

Now we can formally define the XSD mining problem:

Definition

Given an XSD X, and XML tree t conforming to X and a threshold N, the XSD key mining problem consists of finding all keys ϕ such that:

- $t \models \phi$
- $\text{supp}(\phi, t) > N$
- ϕ is consistent w.r.t. X

Outline

- A bit of notation: XML trees and XSDs
- XSD keys: Formal definition
- XSD key mining problem: Formal definition
- The consistency problem: Complexity
- An XSD key mining algorithm
- Experimental evaluation: A few words
- Take-home message

First problem to solve: The consistency problem

$\textsc{Consistency}(\mathcal{DSE})$ is the problem of verifying, given an XSD X and a key ϕ, whether ϕ is consistent w.r.t. X.

- $\textsc{Consistency}(\mathcal{SE})$ is defined analogously, but assuming that target and key paths are selector expressions

To determine the complexity of these problems, we study a closely related problem.

Cardinality of selector expression results

Notation:

- $\mathcal{L}(X)$: set of trees t conforming to XSD X
- $\tau(t)$: defined as $\tau(t, r)$, where r is the root of tree t

Cardinality of selector expression results

Notation:

- $\mathcal{L}(X)$: set of trees t conforming to XSD X
- $\tau(t)$: defined as $\tau(t, r)$, where r is the root of tree t

Definition

Given $\bullet \in \{<, =, >\}$ and $k \geq 0$:

$$\forall_{\text{tree}}^{\bullet k, \mathcal{SE}} = \{(X, p) \mid X \text{ is an XSD, } p \in \mathcal{SE} \text{ and for every } t \in \mathcal{L}(X) : |p(t)| \bullet k\}$$

$\forall_{\text{tree}}^{\bullet k, \mathcal{DSE}}$ is defined analogously.

Cardinality of selector expression results and the consistency problem

Proposition

- $\forall_{\text{tree}}^{=1,\mathcal{SE}}$ *is polynomially equivalent to* $\textsc{Consistency}(\mathcal{SE})$
- $\forall_{\text{tree}}^{=1,\mathcal{DSE}}$ *is polynomially equivalent to* $\textsc{Consistency}(\mathcal{DSE})$

We study the more general family of problems $\forall_{\text{tree}}^{\bullet k,\mathcal{SE}}$

A useful detour: The string case

Notation:

- $\mathcal{L}(A)$: set of strings s accepted by DFA A
- $\tau(s)$: defined as before but considering string s as a tree

Definition

Given $\bullet \in \{<, =, >\}$ and $k \geq 0$:

$$\forall^{\bullet k, \mathcal{SE}}_{\text{string}} = \{(A, p) \mid A \text{ is a DFA}, \\ p \in \mathcal{SE} \text{ and for every } s \in \mathcal{L}(A) : |p(s)| \bullet k\}$$

The complexity of $\forall^{<k,\mathcal{SE}}_{\text{string}}$

Proposition

$\forall^{<k,\mathcal{SE}}_{\text{string}}$ *is in* $\textsc{Ptime}$

The complexity of $\forall^{<k,\mathcal{SE}}_{\text{string}}$

Proposition

$\forall^{<k,\mathcal{SE}}_{\text{string}}$ *is in* PTIME

Proof idea: Assume given a DFA A and selector expression p.

Construct an NFA B^k_p accepting every string s such that $|p(s)| \geq k$

- It can be constructed in polynomial-time as k is fixed

Verify whether $\mathcal{L}(A \times B^k_p) = \emptyset$

- $(A, p) \in \forall^{<k,\mathcal{SE}}_{\text{string}}$ if and only if $\mathcal{L}(A \times B^k_p) = \emptyset$

The complexity of $\forall^{>k,\mathcal{SE}}_{\text{string}}$: A more difficult problem

We show how to reduce $\overline{\text{CNF-SAT}}$ to $\forall^{>0,\mathcal{SE}}_{\text{string}}$.

Let φ be a propositional formula in CNF.

- We define a DFA A and a selector expression p such that φ is not satisfiable if and only if $(A, p) \in \forall^{>0,\mathcal{SE}}_{\text{string}}$

Assume that $\varphi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ and that φ mentions variables $x_1, \ldots, x_n$.

- For each x_i: strings 01, 10 are used to encode values *false* and *true*, respectively

The complexity of $\forall^{>k,\mathcal{SE}}_{\text{string}}$: A more difficult problem (cont'd)

A string w is accepted by A iff $w = s_1\#s_2\#\cdots\#s_m$, where

- s_i is a string of length $2n$ representing a satisfying assignment for clause C_i

Example

If $\varphi = (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_3 \vee \neg x_4)$, then the following strings are accepted by A:

$$10010101\#10010101$$
$$01011001\#10100101$$

The complexity of $\forall^{>k,\mathcal{SE}}_{\text{string}}$: A more difficult problem

Assignments s_i, s_j $(i \neq j)$ can be inconsistent.

- Inconsistent bits will be at distance $2n$. In particular, one will find a 0 followed by 2n symbols, and then followed by a 1

Thus, we consider the following selector expression:

$$p \quad = \quad .//0\underbrace{/*/\cdots/*/}_{2n \text{ times}}1$$

The complexity of $\forall^{>k,\mathcal{SE}}_{\text{string}}$: A more difficult problem

Assignments s_i, s_j $(i \neq j)$ can be inconsistent.

- Inconsistent bits will be at distance $2n$. In particular, one will find a 0 followed by 2n symbols, and then followed by a 1

Thus, we consider the following selector expression:

$$p \quad = \quad .//0\underbrace{/*/\cdots/*/}_{2n \text{ times}}1$$

□

We have shown that $\forall^{>0,\mathcal{SE}}_{\text{string}}$ is coNP-hard

- It can be proved that $\forall^{>k,\mathcal{SE}}_{\text{string}}$ is PSPACE-complete for every k

The complexity of $\forall_{\text{string}}^{=k,\mathcal{SE}}$: A more surprising result

We have that $\forall_{\text{string}}^{=1,\mathcal{SE}} = \forall_{\text{string}}^{<2,\mathcal{SE}} \cap \forall_{\text{string}}^{>0,\mathcal{SE}}$

- We conclude that $\forall_{\text{string}}^{=1,\mathcal{SE}}$ is in $\textsc{Pspace}$

The complexity of $\forall^{=k,\mathcal{SE}}_{\text{string}}$: A more surprising result

We have that $\forall^{=1,\mathcal{SE}}_{\text{string}} = \forall^{<2,\mathcal{SE}}_{\text{string}} \cap \forall^{>0,\mathcal{SE}}_{\text{string}}$

- We conclude that $\forall^{=1,\mathcal{SE}}_{\text{string}}$ is in $\textsc{Pspace}$

But a lot more can be proved:

Theorem

$\forall^{=1,\mathcal{SE}}_{\text{string}}$ *is in* $\textsc{Ptime}$

$\forall^{=1,\mathcal{SE}}_{\text{string}}$ is in PTIME: Proof idea

Assume given a DFA A and selector expression p.

The following polynomial-time algorithm verifies whether $(A, p) \in \forall^{=1,\mathcal{SE}}_{\text{string}}$:

1. Check whether $(A, p) \in \forall^{<2,\mathcal{SE}}_{\text{string}}$. If this condition holds, then go to step 2. Otherwise return **no**.

2. Construct an NFA B_p such that:
   - $s \in \mathcal{L}(B_p)$ iff $|p(s)| \geq 1$
   - the number of accepting runs of B_p on s is equal to $|p(s)|$

3. Compute $A \times B_p$

$\forall^{=1,\mathcal{SE}}_{\text{string}}$ is in PTIME: Proof idea (cont'd)

4. Check whether $\mathcal{L}(A) \subseteq \mathcal{L}(A \times B_p)$. If this condition holds, then return **yes**, otherwise return **no**
   - This is equivalent to verifying whether $\mathcal{L}(A) \subseteq \mathcal{L}(B_p)$

$\forall^{=1,\mathcal{SE}}_{\text{string}}$ is in PTIME: Proof idea (cont'd)

4. Check whether $\mathcal{L}(A) \subseteq \mathcal{L}(A \times B_p)$. If this condition holds, then return **yes**, otherwise return **no**
   - This is equivalent to verifying whether $\mathcal{L}(A) \subseteq \mathcal{L}(B_p)$

Key observations:

- Containment problem for unambiguous finite automata (UFAs) can be solved in polynomial time [SH85].
  - An NFA C is unambiguous if for every $s \in \mathcal{L}(C)$, there is only one accepting run of C on s
- A and $A \times B_p$ are UFAs
  - Given that A is a DFA, $(A, p) \in \forall^{<2,\mathcal{SE}}_{\text{string}}$ and the number of accepting runs of B_p on a string s is equal to $|p(s)|$

String case: Summary of results

$\mathcal{P}$	$\forall^{>k,\mathcal{P}}_{\text{string}}$	$\forall^{<k,\mathcal{P}}_{\text{string}}$	$\forall^{=k,\mathcal{P}}_{\text{string}}$ $(k \geq 1)$
$\mathcal{RE}$	PSPACE-complete	in PTIME	PSPACE-complete
$\mathcal{DSE}$	PSPACE-complete	in PTIME	coNP-complete
$\mathcal{SE}$	PSPACE-complete	in PTIME	in PTIME
$\mathcal{SE}^*$	in PTIME	in PTIME	in PTIME
$\mathcal{SE}^{//}$	in PTIME	in PTIME	in PTIME

Notation:

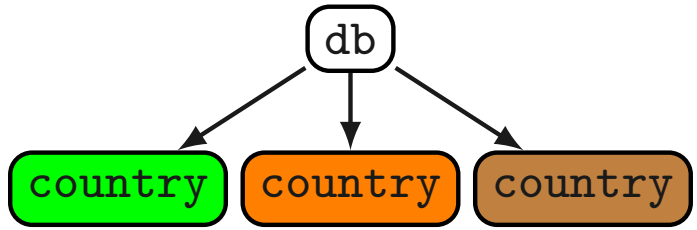
$\mathcal{RE}$: regular expressions
$\mathcal{SE}^*$: selector expressions not using $*$
$\mathcal{SE}^{//}$: selector expressions not using //

Tree case: Summary of results

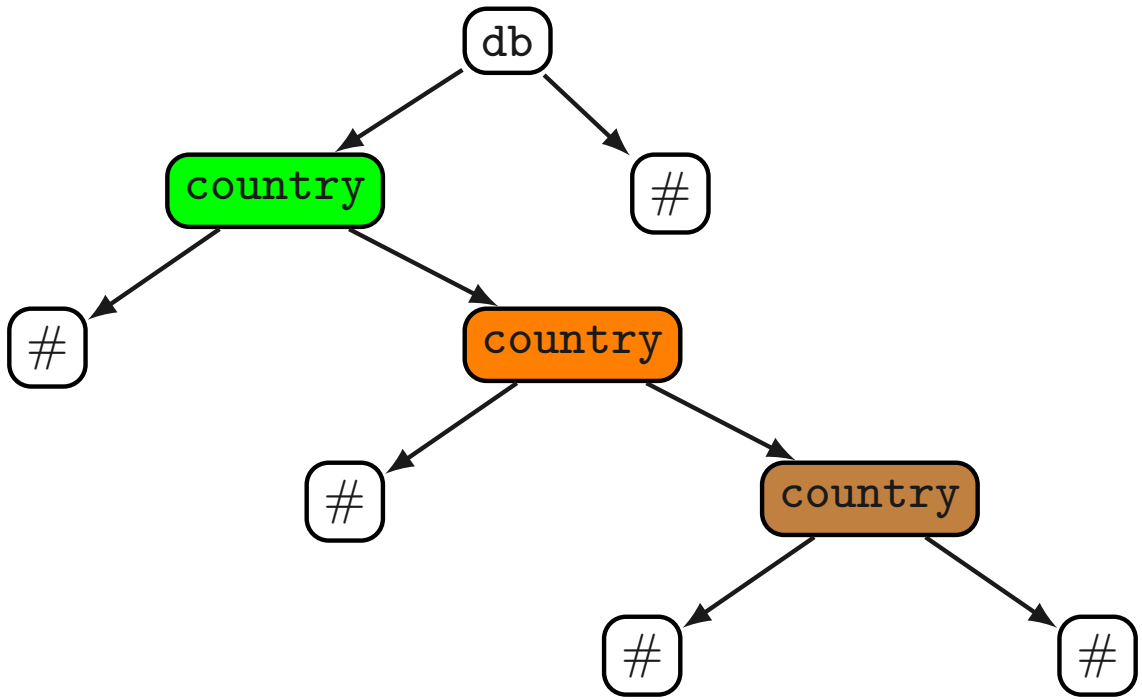
$\mathcal{P}$	$\forall^{>k,\mathcal{P}}_{\text{tree}}$	$\forall^{<k,\mathcal{P}}_{\text{tree}}$	$\forall^{=k,\mathcal{P}}_{\text{tree}}$ $(k \geq 1)$
$\mathcal{RE}$	EXPTIME-complete	in PTIME	in EXPTIME PSPACE-hard
$\mathcal{DSE}$	EXPTIME-complete	in PTIME	in EXPTIME coNP-hard
$\mathcal{SE}$	EXPTIME-complete	in PTIME	in PTIME
$\mathcal{SE}^*$	in EXPTIME	in PTIME	in PTIME
$\mathcal{SE}^{//}$	in PTIME	in PTIME	in PTIME

A polynomial-time algorithm for $\forall_{\text{tree}}^{=1,\mathcal{SE}}$: Binary representations

Tree t:



Binary representation fcns(t) of t:



Binary representations: Main results

We use binary top-down tree automata (BTA) to process binary trees.

- There exists a deterministic BTA $A_{\#}$ such that for every binary tree t': $t' \in \mathcal{L}(A_{\#})$ iff there exists a tree t for which $t' = \mathrm{fcns}(t)$

The following problems can be solved in polynomial-time:

- Given an XSD X, construct a deterministic BTA A_X such that for every tree t: $\mathrm{fcns}(t) \in \mathcal{L}(A_X)$ if and only if $t \in \mathcal{L}(X)$
- Given a selector expression p, construct a BTA B_p such that for every tree t:
  - $\mathrm{fcns}(t) \in \mathcal{L}(B_p)$ if and only if $|p(t)| \geq 1$
  - the number of accepting runs of B_p on $\mathrm{fcns}(t)$ is equal to $|p(t)|$

A polynomial-time algorithm for $\forall_{\text{tree}}^{=1,\mathcal{SE}}$: Putting all together

Assume given an XSD X and selector expression p.

The following polynomial-time algorithm verifies whether $(X, p) \in \forall_{\text{tree}}^{=1,\mathcal{SE}}$:

1. Check whether $(X, p) \in \forall_{\text{tree}}^{<2,\mathcal{SE}}$. If this condition holds, then go to step 2. Otherwise return **no**.

2. Construct a BTA B_p such that for every tree t:
   - $\text{fcns}(t) \in \mathcal{L}(B_p)$ iff $|p(t)| \geq 1$
   - the number of accepting runs of B_p on $\text{fcns}(t)$ is equal to $|p(t)|$

3. Compute $A_{\#} \times A_X$ and $A_{\#} \times A_X \times B_p$

A polynomial-time algorithm for $\forall^{=1,\mathcal{SE}}_{\text{tree}}$: Putting all together

4. Check whether $\mathcal{L}(A_{\#} \times A_X) \subseteq \mathcal{L}(A_{\#} \times A_X \times B_p)$. If this condition holds, then return **yes**, otherwise return **no**

Key observations:

- Containment problem for unambiguous BTAs can be solved in polynomial time [S90].
- $A_{\#} \times A_X$ and $A_{\#} \times A_X \times B_p$ are unambiguous BTAs
  - Given that $A_{\#}$, A_X are deterministic, $(X, p) \in \forall^{<2,\mathcal{SE}}_{\text{string}}$ and the number of accepting runs of B_p on a tree $t' = \text{fcns}(t)$ is equal to $|p(t)|$

The consistency problem: Main results

Theorem

- $\textsc{Consistency}(\mathcal{SE})$ *is in* $\textsc{Ptime}$
- $\textsc{Consistency}(\mathcal{DSE})$ *is* $\textsc{coNP}$*-hard and in* $\textsc{Exptime}$

Outline

- A bit of notation: XML trees and XSDs
- XSD keys: Formal definition
- XSD key mining problem: Formal definition
- The consistency problem: Complexity
- An XSD key mining algorithm
- Experimental evaluation: A few words
- Take-home message

An XSD key mining algorithm

Input: XSD X, a tree t conforming to X and a threshold N

Algorithm:

for all $c \in \text{ContextMiner}_{t,X}$ **do**
  for all $\tau \in \text{TargetPathMiner}_{t,X,N}(c)$ **do**
    $S := \text{OneKeyPathMiner}_{t,X}(c, \tau)$
    $\mathcal{P} := \text{MinimalKeyPathSetMiner}_{t,X}(c, \tau, S)$
    for each $P \in \mathcal{P}$ **return** (c, τ, P)

An XSD key mining algorithm

Input: XSD X, a tree t conforming to X and a threshold N

Algorithm:

for all $c \in \text{ContextMiner}_{t,X}$ **do**
  for all $\tau \in \text{TargetPathMiner}_{t,X,N}(c)$ **do**
    $S := \text{OneKeyPathMiner}_{t,X}(c, \tau)$
    $\mathcal{P} := \text{MinimalKeyPathSetMiner}_{t,X}(c, \tau, S)$
    for each $P \in \mathcal{P}$ **return** (c, τ, P)

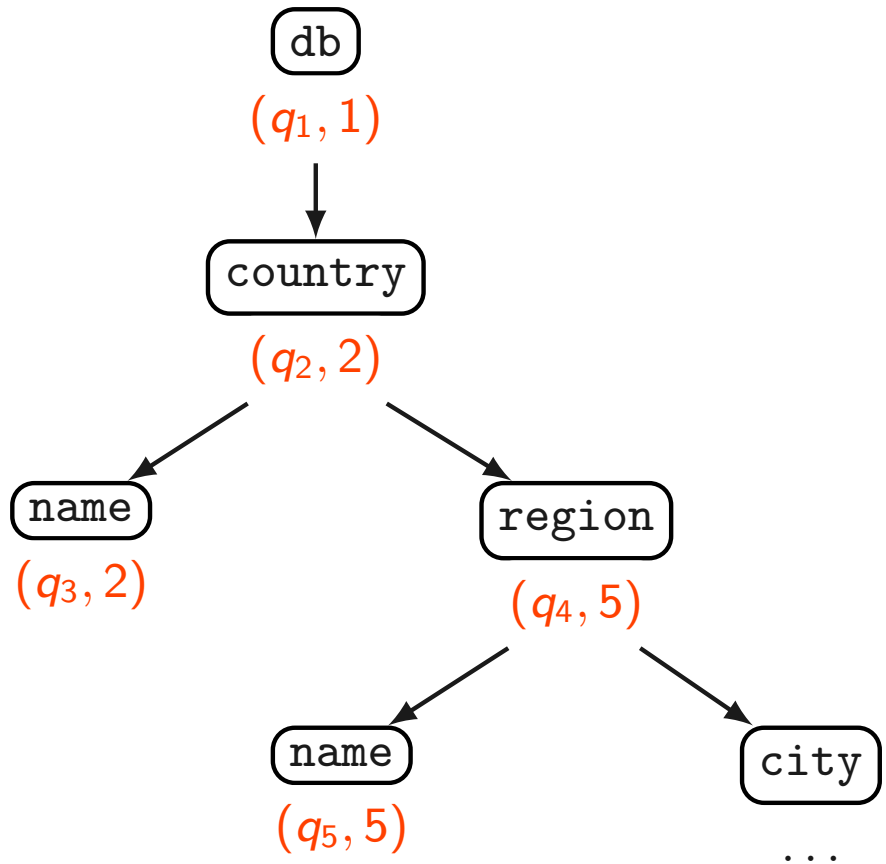
Restrictions:

- We focus on selector expressions (we disregards the union operator)
- We consider target and key paths up to a given length k_{max} (which can be at most the depth of t)

ContextMiner$_{t,X}$

It returns a list of possible contexts based on X and t.

Prefix tree $\mathrm{PT}(t)$ is used in this procedure: Obtained by collapsing all nodes with the same ancestor string.



$\mathsf{TargetPathMiner}_{t,X,N}(c)$

Given a context node c, it returns a list of target paths with support greater than N in t.

- It follows a framework of levelwise search [MT97]

Consider the following partial order on selector expressions:

$$\tau_1 \preceq \tau_2 \text{ iff for every tree } t\text{: } \tau_2(t) \subseteq \tau_1(t)$$

If $\tau_1 \preceq \tau_2$, then τ_2 is more specific than τ_1

Deciding $\preceq$

Consider the following "one-step specialization relation" on selector expressions:

- $\tau \prec_1 \tau'$ if τ' is obtained from τ by one of the following operations:
  - if τ starts with the descendant axis, replace it by the child axis
  - if τ starts with the descendant axis, insert a wildcard step right after it
  - replacing a wildcard with an element name

Deciding $\preceq$

Consider the following "one-step specialization relation" on selector expressions:

- $\tau \prec_1 \tau'$ if τ' is obtained from τ by one of the following operations:
  - if τ starts with the descendant axis, replace it by the child axis
  - if τ starts with the descendant axis, insert a wildcard step right after it
  - replacing a wildcard with an element name

We can decide whether $\tau \preceq \tau'$ by using relation $\prec_1$:

Proposition

$\preceq$ is equal to the reflexive and transitive closure of $\prec_1$.

$\mathsf{TargetPathMiner}_{t,X,N}(c)$ (cont'd)

Let U be the set of all selector expressions of length at most $k_{\max}$.

$\mathsf{TargetPathMiner}_{t,X,N}(c)$ uses partial order $\preceq$:

$C_0 := \{.//*\}$
$i := 0$
while $C_i \neq \emptyset$ **do**
$F_i := \{\tau \in C_i \mid \mathsf{supp}(c, \tau, t) > N\}$
$C_{i+1} := \{\tau \in U \mid \forall \tau' \in U : \tau' \prec \tau \Rightarrow \tau' \in \bigcup_{j \leq i} F_j\} \setminus \bigcup_{j \leq i} C_j$
$i := i + 1$
return $\bigcup_{j \leq i} F_j$

TargetPathMiner$_{t,X,N}(c)$ (cont'd)

Let U be the set of all selector expressions of length at most $k_{\max}$.

TargetPathMiner$_{t,X,N}(c)$ uses partial order $\preceq$:

$C_0 := \{.//*\}$
$i := 0$
while $C_i \neq \emptyset$ **do**
$F_i := \{\tau \in C_i \mid \text{supp}(c, \tau, t) > N\}$
$C_{i+1} := \{\tau \in U \mid \forall \tau' \in U : \tau' \prec \tau \Rightarrow \tau' \in \bigcup_{j \leq i} F_j\} \setminus \bigcup_{j \leq i} C_j$
$i := i + 1$
return $\bigcup_{j \leq i} F_j$

Some remarks:

TargetPathMiner$_{t,X,N}(c)$ (cont'd)

Let U be the set of all selector expressions of length at most $k_{\max}$.

TargetPathMiner$_{t,X,N}(c)$ uses partial order $\preceq$:

$C_0 := \{.//*\}$
$i := 0$
while $C_i \neq \emptyset$ **do**
$F_i := \{\tau \in C_i \mid \text{supp}(c, \tau, t) > N\}$
$C_{i+1} := \{\tau \in U \mid \forall \tau' \in U : \tau' \prec \tau \Rightarrow \tau' \in \bigcup_{j \leq i} F_j\} \setminus \bigcup_{j \leq i} C_j$
$i := i + 1$
return $\bigcup_{j \leq i} F_j$

Some remarks:

- Prefix tree $\mathrm{PT}(t)$ is used when checking $\text{supp}(c, \tau, t) > N$

TargetPathMiner$_{t,X,N}(c)$ (cont'd)

Let U be the set of all selector expressions of length at most $k_{\max}$.

TargetPathMiner$_{t,X,N}(c)$ uses partial order $\preceq$:

$C_0 := \{.//*\}$
$i := 0$
while $C_i \neq \emptyset$ **do**
$F_i := \{\tau \in C_i \mid \text{supp}(c, \tau, t) > N\}$
$C_{i+1} := \{\tau \in U \mid \forall \tau' \in U : \tau' \prec \tau \Rightarrow \tau' \in \bigcup_{j \leq i} F_j\} \setminus \bigcup_{j \leq i} C_j$
$i := i + 1$
return $\bigcup_{j \leq i} F_j$

Some remarks:

- Prefix tree $\mathrm{PT}(t)$ is used when checking $\text{supp}(c, \tau, t) > N$
- For more specific results: $\bigcup_{j \leq i} F_j$ is replaced by F_i

TargetPathMiner$_{t,X,N}(c)$ (cont'd)

Let U be the set of all selector expressions of length at most $k_{\max}$.

TargetPathMiner$_{t,X,N}(c)$ uses partial order $\preceq$:

$C_0 := \{.//*\}$
$i := 0$
while $C_i \neq \emptyset$ **do**
$F_i := \{\tau \in C_i \mid \text{supp}(c, \tau, t) > N\}$
$C_{i+1} := \{\tau \in U \mid \forall \tau' \in U : \tau' \prec \tau \Rightarrow \tau' \in \bigcup_{j \leq i} F_j\} \setminus \bigcup_{j \leq i} C_j$
$i := i + 1$
return $\bigcup_{j \leq i} F_j$

Some remarks:

- Prefix tree $\mathrm{PT}(t)$ is used when checking $\text{supp}(c, \tau, t) > N$
- For more specific results: $\bigcup_{j \leq i} F_j$ is replaced by F_i
- A more efficient implementation uses $\prec_1$ instead of $\preceq$

OneKeyPathMiner$_{t,X}(c, \tau)$

It returns the set of all key paths p of length at most $k_{\max}$ for which $(c, \tau, (p))$ is consistent with X

It uses the consistency algorithm presented in the previous slides.

- To reduce the number of consistency tests, it only considers the candidates p for which $(c, \tau, (p))$ qualifies in t

MinimalKeyPathSetMiner$_{t,X}(c, \tau, S)$

It returns a set $\mathcal{P}$ of minimal subsets P of S for which $t \models (c, \tau, P)$

- We capitalize on existing relational techniques for mining functional dependencies

Assume that $S = \{p_1, \ldots, p_k\}$, and let R be a relation such that:

- The schema of R is $(\text{CID}, \text{TID}, p_1 \ldots, p_k)$
- $(v, u, a_1, \ldots, a_k) \in R$ iff $v \in \text{CNodes}_t(c)$, $u \in \tau(t, v)$ and $\text{record}_S(t, u) = [a_1, \ldots, a_k]$

MinimalKeyPathSetMiner$_{t,X}(c, \tau, S)$ (cont'd)

For every $P = \{p_{i_1}, \ldots, p_{i_n}\}$, with $1 \leq i_1 < \cdots < i_n \leq k$, we have that:

$$t \models (c, \tau, (p_{i_1}, \ldots, p_{i_n}))$$
$$\text{iff}$$
$$\text{CID}, p_{i_1}, \ldots, p_{i_n} \rightarrow \text{TID holds in } R$$

Thus, we can now plug in any existing functional dependency discovery algorithm.

Outline

- A bit of notation: XML trees and XSDs
- XSD keys: Formal definition
- XSD key mining problem: Formal definition
- The consistency problem: Complexity
- An XSD key mining algorithm
- Experimental evaluation: A few words
- Take-home message

Experimental evaluation: Setting

We use a corpus of 90 XML trees and associated XSDs from the University of Amsterdam XML Web Collection.

- Maximal and average number of elements occurring in trees is 91K and 5K, respectively
- Maximal and average number of elements occurring in XSDs is 532 and 52, respectively

We search for keys with target path length at most 4 and key path length at most 2.

Experimental evaluation: A few words

- In ContextMiner$_{t,X}$, the use of prefix trees allows to reduce the number of contexts: 52% do not need to be considered

- In TargetPathMiner$_{t,X,N}(c)$, the framework of levelwise search allows a significant reduction in the number of target paths considered:

possible TPs	2.4×10^{11}
candidate TPs	6.7×10^{6}
supported TPs	8.4×10^{4}

Experimental evaluation: A few words (cont'd)

- In OneKeyPathMiner$_{t,X}(c, \tau)$, start by checking consistency on trees allows a significant reduction on the number of expensive consistency tests:

candidate KPs	48144
consistent KPs on tree	484
consistent KPs on XSD	288

- Resulting keys from MinimalKeyPathSetMiner$_{t,X}(c, \tau, S)$:

consistency test	no		yes	
support	10	100	10	100
derived keys	107	54	43	16
trees with keys	27	15	19	10
average nr. of keys per tree	4	3.6	2.2	1.6
max nr. of keys per tree	23	23	9	4

Outline

- A bit of notation: XML trees and XSDs
- XSD keys: Formal definition
- XSD key mining problem: Formal definition
- The consistency problem: Complexity
- An XSD key mining algorithm
- Experimental evaluation: A few words
- Take-home message

Take-home message

- We consider the problem of discovering XSD keys from a given XML tree that conforms to a given XSD.
- We consider the semantics for XSD keys proposed by the W3C, and we incorporate in our discovering algorithm some quality criteria like consistency, which was shown to be decidable in polynomial time.

Thank you!

Bibliography

[MT97] H. Mannila, H. Toivonen: Levelwise Search and Borders of Theories in Knowledge Discovery. Data Min. Knowl. Discov. 1(3): 241–258, 1997

[S90] H. Seidl: Deciding Equivalence of Finite Tree Automata. SIAM J. Comput. 19(3):424–437, 1990

[SH85] R. E. Stearns, H. B. Hunt III: On the Equivalence and Containment Problems for Unambiguous Regular Expressions, Regular Grammars and Finite Automata. SIAM J. Comput. 14(3):598–611, 1985

Backup slides

The complexity of $\forall^{=k,\mathcal{SE}}_{\text{string}}$: The general case

Theorem

$\forall^{=k,\mathcal{SE}}_{\text{string}}$ *is in* PTIME *for every* k

The complexity of $\forall^{=k,\mathcal{SE}}_{\text{string}}$: The general case

Theorem

$\forall^{=k,\mathcal{SE}}_{\text{string}}$ *is in* PTIME *for every* k

Proof idea: $\forall^{=0,\mathcal{SE}}_{\text{string}} = \forall^{<1,\mathcal{SE}}_{\text{string}}$, so let's consider the case $k \geq 1$

Polynomial-time algorithm to verify whether $(A, p) \in \forall^{=k,\mathcal{SE}}_{\text{string}}$:

1. Check whether $(A, p) \in \forall^{<k+1,\mathcal{SE}}_{\text{string}}$. If this condition holds, then go to step 2. Otherwise return **no**.

2. Construct an NFA B^k_p such that:
   - $s \in \mathcal{L}(B^k_p)$ iff $|p(s)| \geq k$
   - if $s \in \mathcal{L}(B^k_p)$, then the number of accepting runs of B^k_p on s is

$$\frac{|p(s)|!}{(|p(s)| - k)!}$$

The complexity of $\forall^{=k,\mathcal{SE}}_{\text{string}}$: The general case (cont'd)

3. Compute $A \times B_p^k$

4. Check whether $\mathcal{L}(A) \subseteq \mathcal{L}(A \times B_p^k)$. If this condition holds, then return **yes**, otherwise return **no**

Key observations:

- Containment problem for c-unambiguous finite automata (c-UFAs) can be solved in polynomial time [SH85].
- $A \times B_p$ is a $k!$-UFA:
  - If $(A, p) \in \forall^{<k+1,\mathcal{SE}}_{\text{string}}$ and $s \in \mathcal{L}(A \times B_p^k)$, then $|p(s)| = k$ and the number of accepting runs of $A \times B_p^k$ on s is bounded by:

$$\frac{|p(s)|!}{(|p(s)| - k)!} = \frac{k!}{(k-k)!} = k!$$

□