Access Control Models for XML

Abdessamad Imine

Lorraine University & INRIA-LORIA Grand-Est

Nancy, France

Abdessamad.Imine@loria.fr

Outline

- Overview on XML
- Why XML Security?
- Querying Views-based XML Data
- Updating Views-based XML Data

Outline

- Overview on XML
- Why XML Security?
- Querying Views-based XML Data
- Updating Views-based XML Data

What is XML?

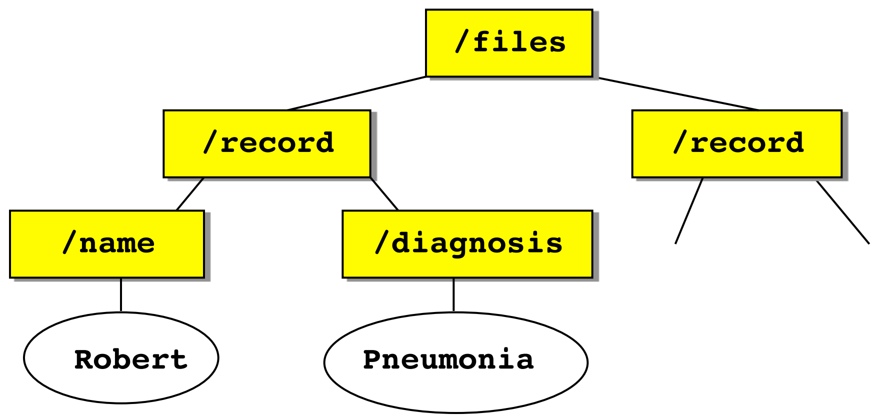
- eXtensible Markup Language [W3C 1998]

```
<files>
        <record>
                <name>Robert</name>
                <diagnosis>Pneumonia</diagnosis>
        </record>
        <record>
                <name>Franck</name>
                <diagnosis>Ulcer</diagnosis>
        </record>
</files>
```

What is XML?

- eXtensible Markup Language [W3C 1998]

```
<files>
 <record>
  <name>Robert</name>
  <diagnosis>
    Pneumonia
  </diagnosis>
 </record>
 <record …>
   …
 </record>
</files>
```



XML for Documents

- SGML
- HTML - hypertext markup language
- TEI - Text markup, language technology
- DocBook - documents -> html, pdf, ...
- SMIL - Multimedia
- SVG - Vector graphics
- MathML - Mathematical formulas

XML for Semi-Structured Data

- MusicXML
- NewsML
- iTunes
- DBLP http://dblp.uni-trier.de
- CIA World Factbook
- IMDB http://www.imdb.com/
- XBEL - bookmark files (in your browser)
- KML - geographical annotation (Google Maps)
- XACML - XML Access Control Markup Language

XML as Description Language

- Java servlet config (web.xml)
  - Apache Tomcat, Google App Engine, ...
- Web Services - WSDL, SOAP, XML-RPC
- XUL - XML User Interface Language (Mozilla/Firefox)
- BPEL - Business process execution language
- Other Web standards:
  - XSLT, XML Schema, XQueryX
  - RDF/XML
  - OWL - Web Ontology Language
  - MMI - Multimodal interaction (phone + car + PC)

XML Tools

- Standalone:
  - xsltproc, mxquery, calabash (XProc)
- Most Programming Languages have XML parsers
  - SAX (streaming), DOM (in-memory) interfaces
  - libxml2, expat, libxslt (C)
  - Xerces, Xalan (Java)
- XPath (path expressions) used in many languages
  - JavaScript/JQuery
  - XSLT, XQuery

Native XML Databases

- Offer native support for XML data & query languages
  - Galax
  - MarkLogic
  - eXist
  - BaseX
  - among others...
- Suitable for new or lightweight applications
  - but some lack features like transactions, views, updates

XML in the Industry

- Most commercial RDBMSs now provide some XML support
  - Oracle 11g - XML DB
  - IBM DB2 pureXML
  - Microsoft SQL Server - XML support since 2005
    - Language Integrated Query (LINQ) targets SQL & XML in .NET programs
- Data publishing, exchange, integration problems are very important
  - big 3 have products for all of these
  - SQL/XML standard for defining XML views of relational data

XML Terminology

Tags and Text

- ✓ XML consists of tags and text

  <course cno = “Eng 055”>

  <title> Spelling </title>

  </course>

- ✓ tags come in pairs: markups

  start tag: **<course>**

  end tag: **</course>**

- ✓ tags must be properly nested

<course> <title> … </title> </course> -- good

<course> <title> … </course> </title> -- ???

XML Terminology

Tags and Text

- ✓ XML consists of tags and text

  <course cno = “Eng 055”>

  <title> Spelling </title>

  </course>

- ✓ tags come in pairs: markups

  start tag: <course>

  end tag: </course>

- ✓ tags must be properly nested

<course> <title> … </title> </course> -- good

<course> <title> … </course> </title> -- bad

XML Terminology (cont.)

XML Elements

- ✓ Element: the segment between a start and its corresponding end tag
- ✓ Subelement: the relation between an element and its component elements.

```
<person>
        <name> Ali Baba </name>
        <tel> (33) 354595853 </tel>
        <email> Ali.Baba@nights.com </email>
        <email> ababa@tales.org </email>
</person>
```

XML Terminology (cont.)

XML Attributes

A start tag may contain attributes describing certain "properties" of the element:

```
<picture>
 <height dim="cm"> 2400</height>
 <width dim="in"> 96 </width>
 <data encoding="gif"> M05-+C$ … </data>
</picture>
```

References:

```
<person id = "011"   country ="UK">
 <name> Stan Laurel </name>
</person>
<person country="USA" id = "012">
 <name> Oliver Hardy </name>
</person>
```

XML Terminology (cont.)

Example: A relational database for school

Students:

id	name	sex
001	Joe	male
002	Mary	female
…	…	…

Course:

cno	title	credit
331	DB	3.0
350	Web	3.0
…	…	…

Enroll:

id	cno
001	331
001	350
002	331
…	…

XML Terminology (cont.)

Example: A relational database for school

```
<school>
        <student   id="001">
                <name>  Joe </name>
                <sex> male </sex>
        </student>
        …
        <course   cno="331">
                <title>  DB </title>
                <credit>    3.0  </credit>
        </course>
        …
        </course>
        <enroll>
                <id>  001 </id>
                <cno>    331  </cno>
        </enroll>
        …
</school>
```

Document Type Definition (DTD)

An XML document may come with an optional DTD – "schema"

```
<!DOCTYPE  db [
    <!ELEMENT   db  (book*)>
    <!ELEMENT   book   (title,  authors*, section*, ref*)>
    <!ATTLIST        book  isbn   ID   #required>
    <!ELEMENT   section  (text | section)*>
    <!ELEMENT   ref  EMPTY>
    <!ATTLIST          ref  to    IDREFS   #implied>
    <!ELEMENT      title  #PCDATA>
    <!ELEMENT      author  #PCDATA>
    <!ELEMENT      text  #PCDATA>
]>
```

Document Type Definition (DTD)

for each element type E, a declaration of the form:

<!ELEMENT E P> **E → P**

where P is a regular expression, i.e.,

P ::= EMPTY | ANY | #PCDATA | E' |

P1, P2 | P1 | P2 | P? | P+ | P*

- **E'** : element type
- **P1 , P2**: concatenation
- **P1 | P2**: disjunction
- **P?**: optional
- **P+**: one or more occurrences
- **P***: the Kleene closure

Document Type Definition (DTD)

- ✓ Extended context free grammar: **<!ELEMENT E P>**

  Why is it called extended?

  E.g., **book → title, authors*, section*, ref***
- ✓ single root: **<!DOCTYPE db […] >**
- ✓ subelements are ordered.

  The following two definitions are different. Why?

  <!ELEMENT section (text | section)*>

  <!ELEMENT section (text* | section*)>
- ✓ recursive definition, e.g., section, binary tree:

  <!ELEMENT node (leaf | (node, node))

  <!ELEMENT leaf (#PCDATA)>

Document Type Definition (DTD)

- ✓ Recursive DTDs

<!ELEMENT person (name, father, mother)>

<!ELEMENT father (person)>

<!ELEMENT mother (person)>

What is the problem with this? How to fix it?

Document Type Definition (DTD)

- ✓ Recursive DTDs

  <!ELEMENT person (name, father, mother)>

  <!ELEMENT father (person)>

  <!ELEMENT mother (person)>

  What is the problem with this? How to fix it?

  - optional (e.g., father?, mother?)
  - Attributes

- ✓ Ordering

  How to declare element E to be an unordered pair (a, b)?

Document Type Definition (DTD)

- ✓ Recursive DTDs

  <!ELEMENT person (name, father, mother)>

  <!ELEMENT father (person)>

  <!ELEMENT mother (person)>

  What is the problem with this? How to fix it?

  - optional (e.g., father?, mother?)
  - Attributes

- ✓ Ordering

  How to declare E to be an unordered pair (a, b)?

  <!ELEMENT E ((a, b) | (b, a)) >

Document Type Definition (DTD)

Attribute Declaration

<!ATTLIST **element_name**

attribute-name attribute-type default-declaration>

Example: “keys” and “foreign keys”

<!ATTLIST book

isbn ID #required>

<!ATTLIST ref

to IDREFS #implied>

Note: it is OK for several element types to define an attribute of the same name, e.g.,

<!ATTLIST person name ID #required>

<!ATTLIST pet name ID #required>

Document Type Definition (DTD)

Attribute Declaration

```
<!ATTLIST     person
              id          ID        #required
              father      IDREF     #implied
              mother      IDREF     #implied
              children    IDREFS    #implied>
```

e.g.,

```
<person  id="898"  father="332"  mother="336"
         children="982  984  986">
 ….
</person>
```

Valid XML Documents

A valid XML document must have a DTD.

- ✓ The document is well-formed
  - Tags have to nest properly
  - Attributes have to be unique
- ✓ It conforms to the DTD:
  - elements conform to the grammars of their type definitions (nested only in the way described by the DTD)
  - elements have all and only the attributes specified by the DTD
  - ID/IDREF attributes satisfy their constraints:
    - ID must be distinct
    - IDREF/IDREFS values must be existing ID values

XPath

W3C standard: www.w3.org/TR/xpath

- ✓ Navigating an XML tree and finding parts of the tree (node selection and value extraction)

  Given an XML tree T and a context node n, an XPath query Q returns

  - the set of nodes reachable via Q from the node n in T – if Q is a unary query
  - truth value indicating whether Q is true at n in T – if Q is a boolean query.

- ✓ Implementations: XALAN, SAXON, Berkeley DB XML, Monet XML – freeware, which you can play with
- ✓ A major element of XSLT, XQuery and XML Schema
- ✓ Version: XPath 3.0

XPath

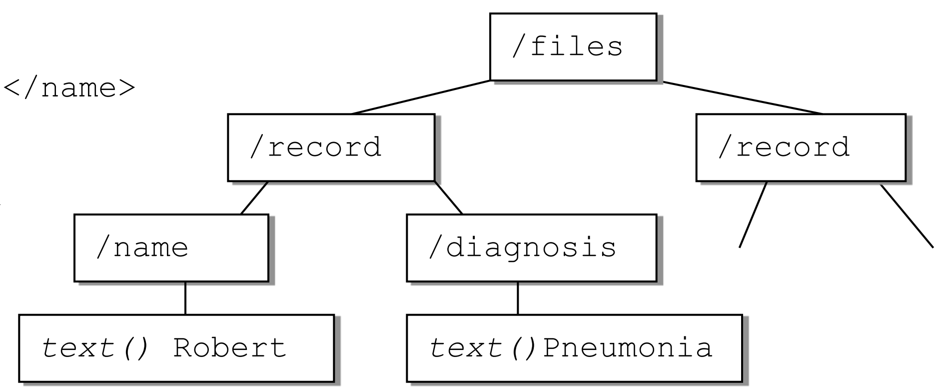
XPath query Q:

- Tree traversal: downward, upward, sideways
- Relational/Boolean expressions: qualifiers (predicates)
- Functions: aggregation (e.g., count), string functions

/files/record/name[text()="Ali Baba"]

/files/record[name="Toto"]/diagnosis | /files/record[name="Pascal"]/diagnosis

```
<files>
 <record>
  <name>Robert</name>
  <diagnosis>
    Pneumonia
  </diagnosis>
 </record>
 <record …>
   …
 </record>
</files>
```



/files/record

XPath

Downward Traversal

Syntax:

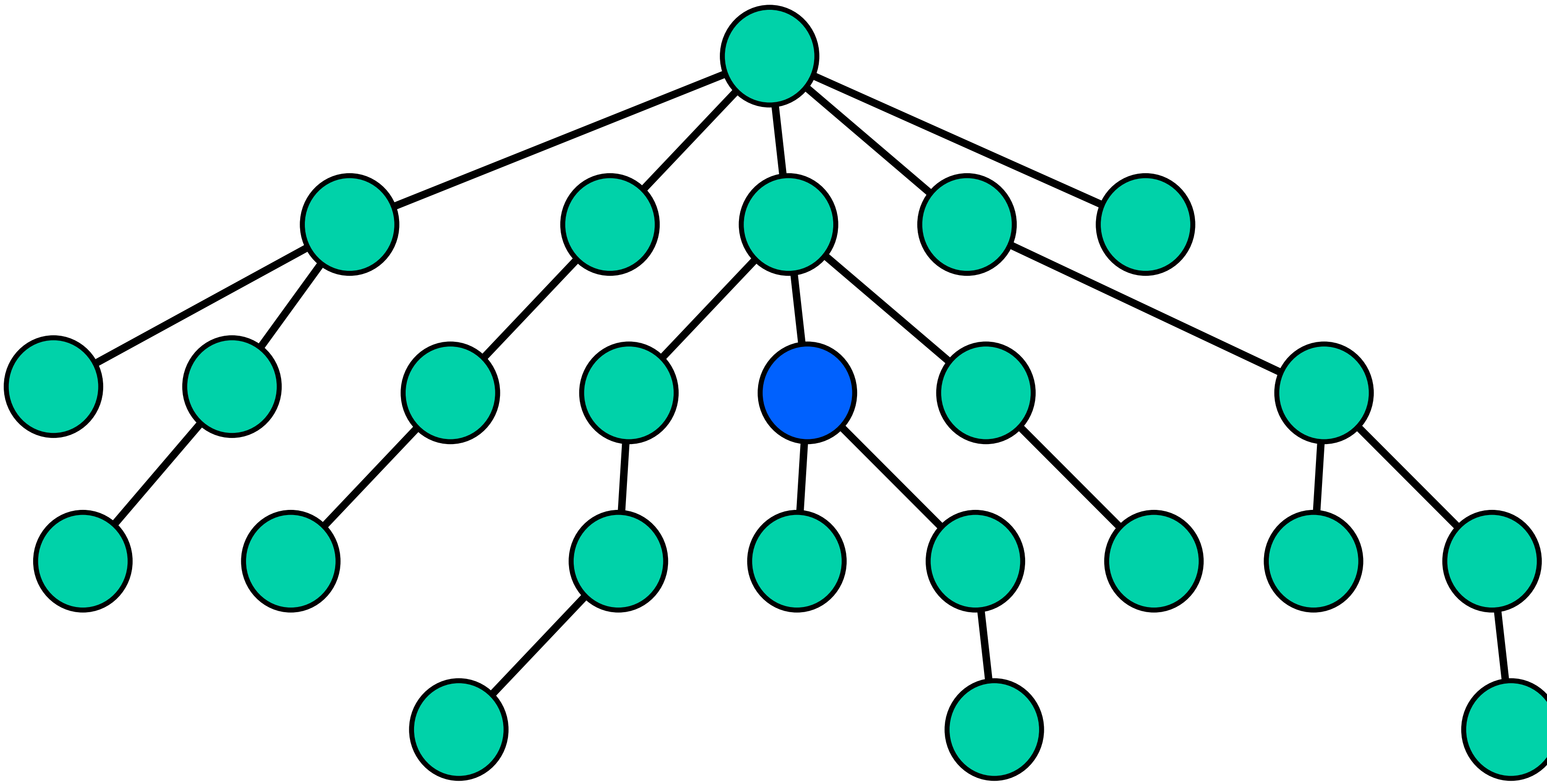
Q ::= l | @l | Q/Q | Q | Q | //Q | /Q | Q[q]

q ::= Q | Q op c | q and q | q or q | not(q)

- **l**: either a tag (label) or *: wildcard that matches any label
- **@l**: attribute
- **/**, **|**: concatenation (child), union
- **//**: descendants or self, "recursion"
- **[q]**: qualifier (filter, predicate)
  - **op**: =, !=, <=, <, >, >=, >
  - **c**: constant
  - **and**, **or**, **not()**: conjunction, disjunction, negation

XPath

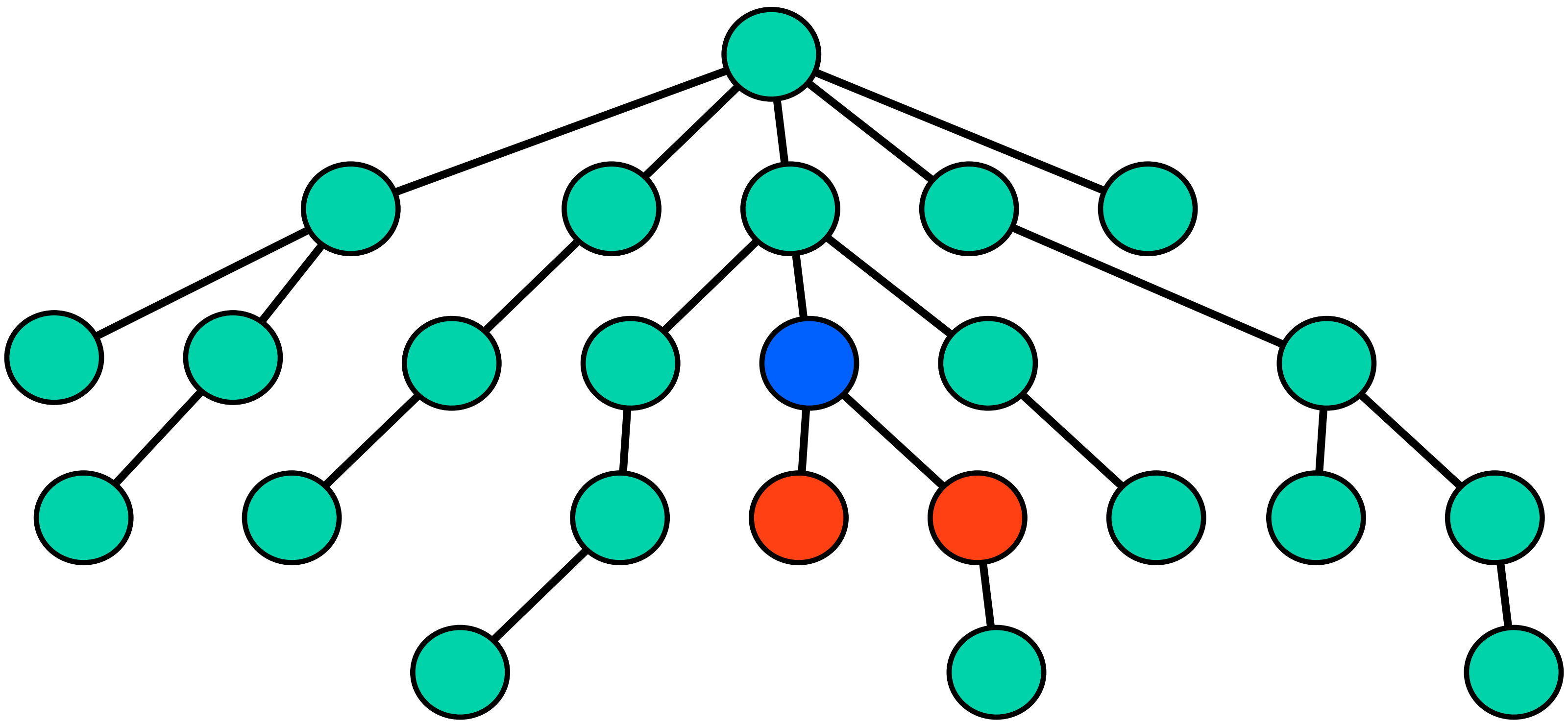
Context node: starting point



XPath

Child

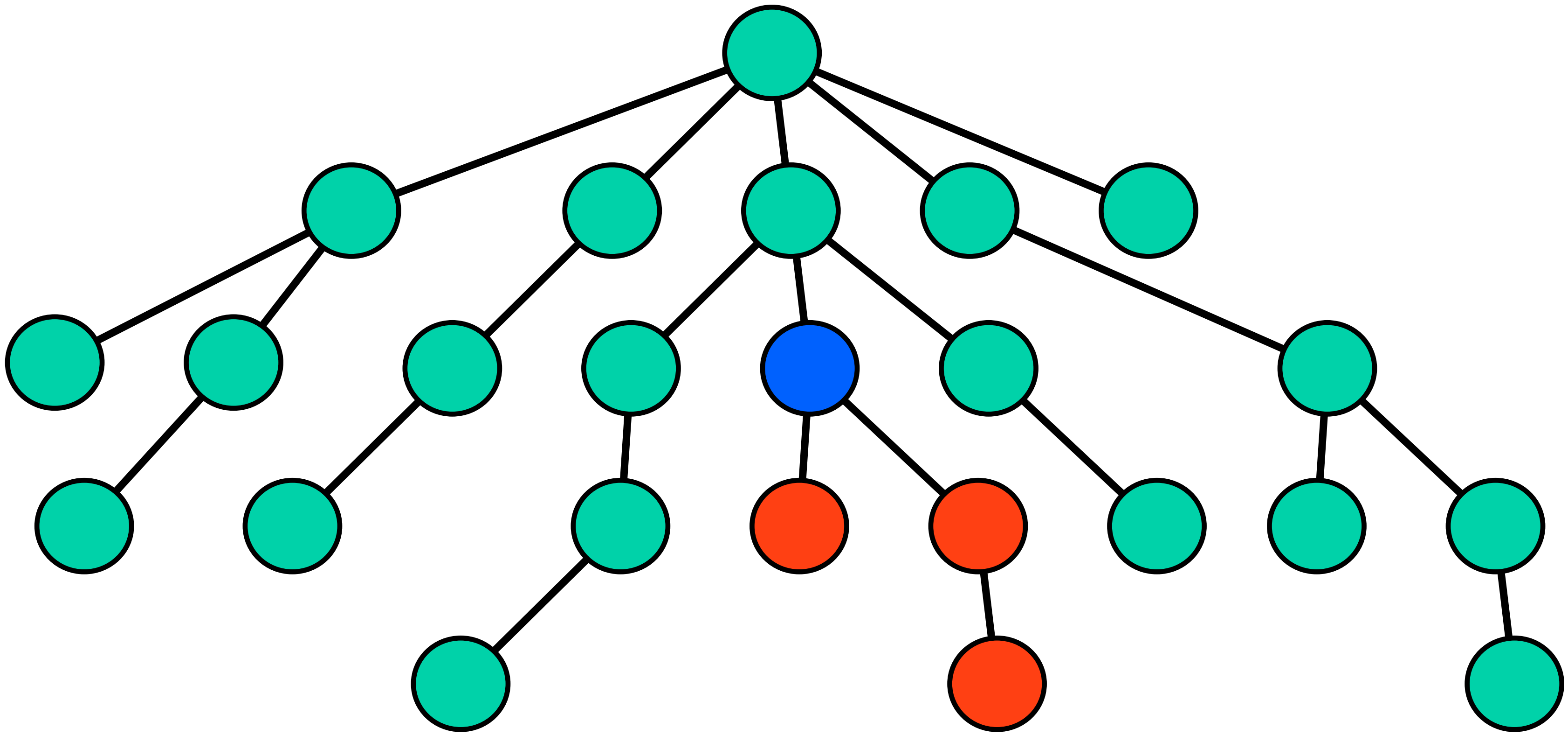
/a is equivalent to child::a



XPath

Descendant

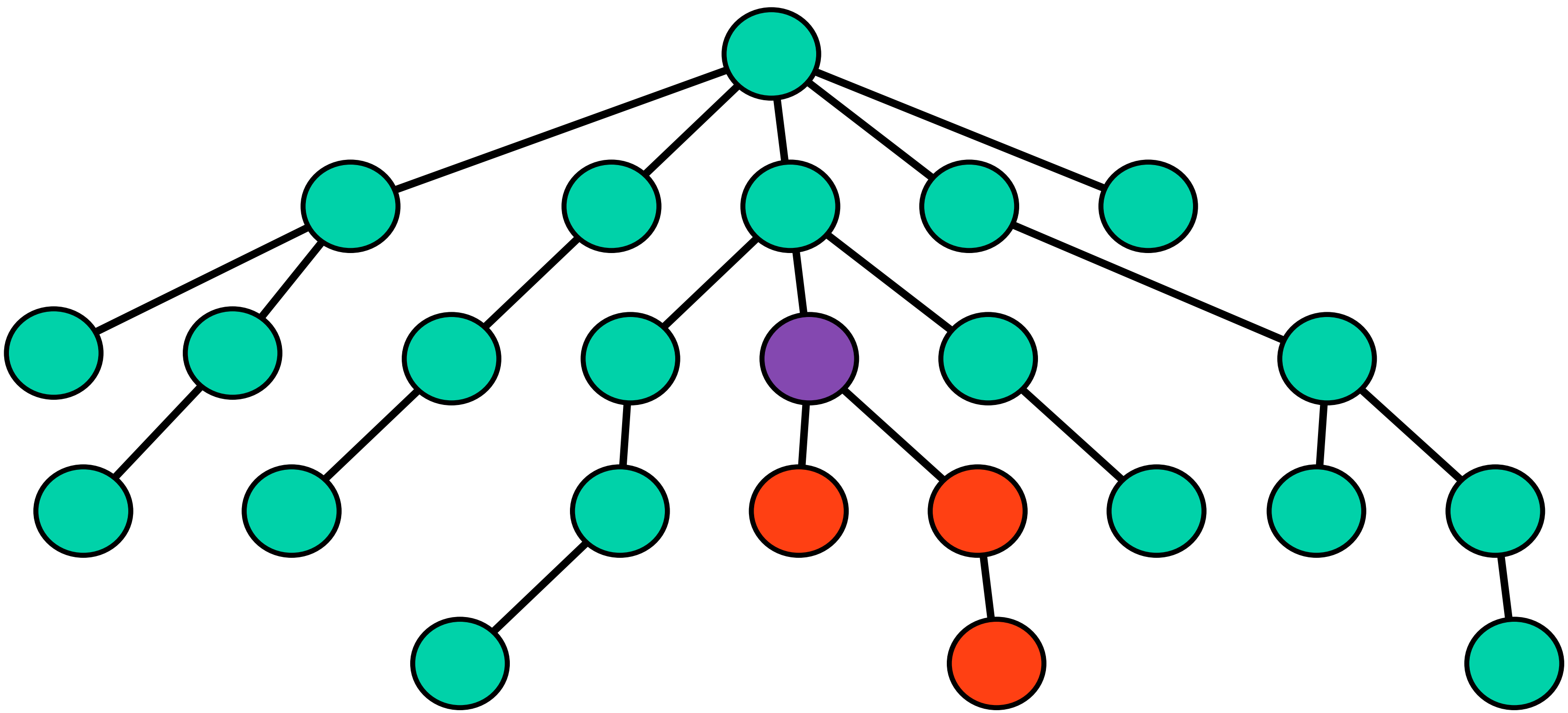
/descendant::*



XPath

Descendant-or-self

//a is equivalent to descendant-or-self::*/child::a



XPath

Upward Traversal

Syntax:

Q ::= . . . | ../Q | ancestor ::Q | ancestor-or-self::Q

- ✓ **../: parent**
- ✓ **ancestor, ancestor-or-self: recursion**

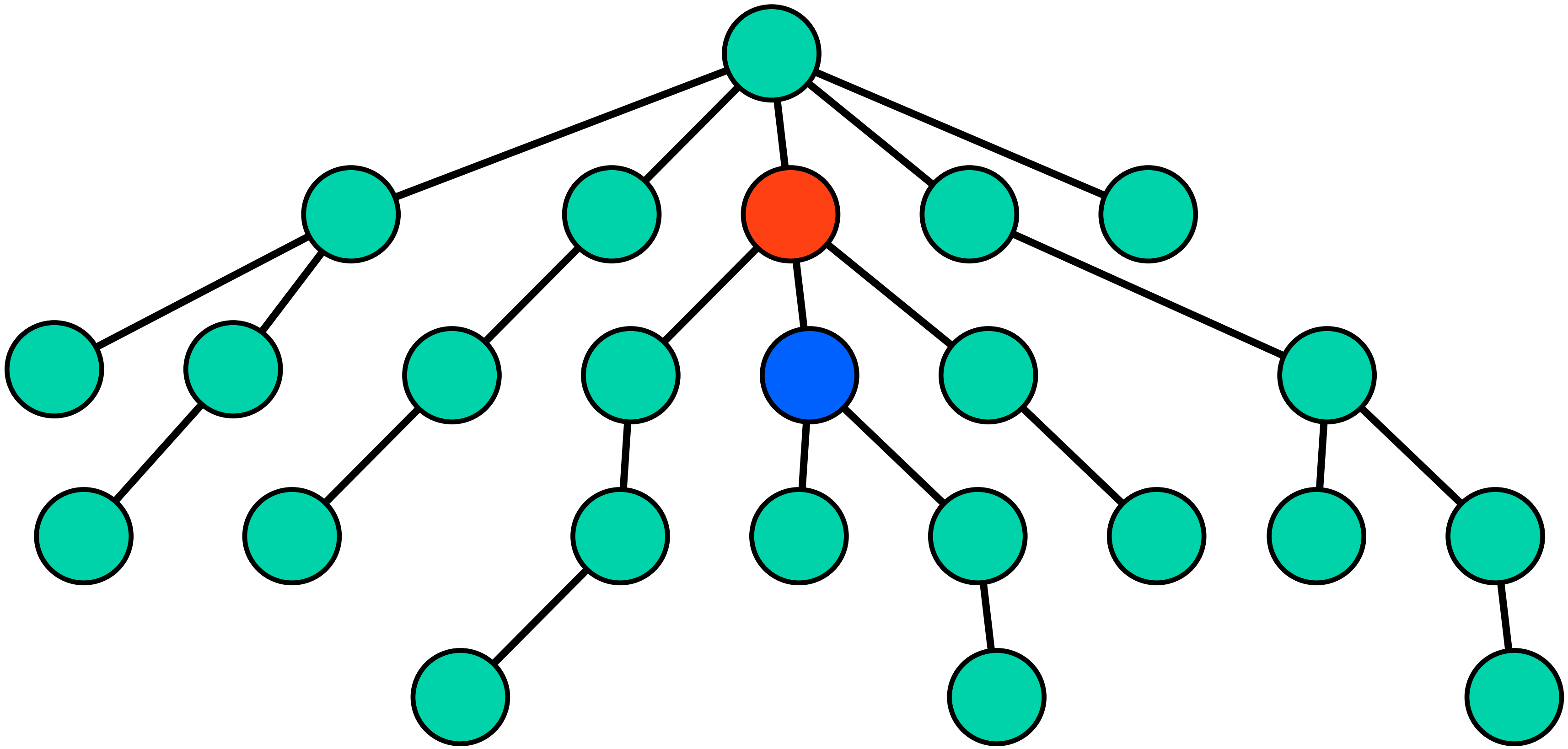
Abreviations:

. is equivalent to **self::***

.. is equivalent to parent::*

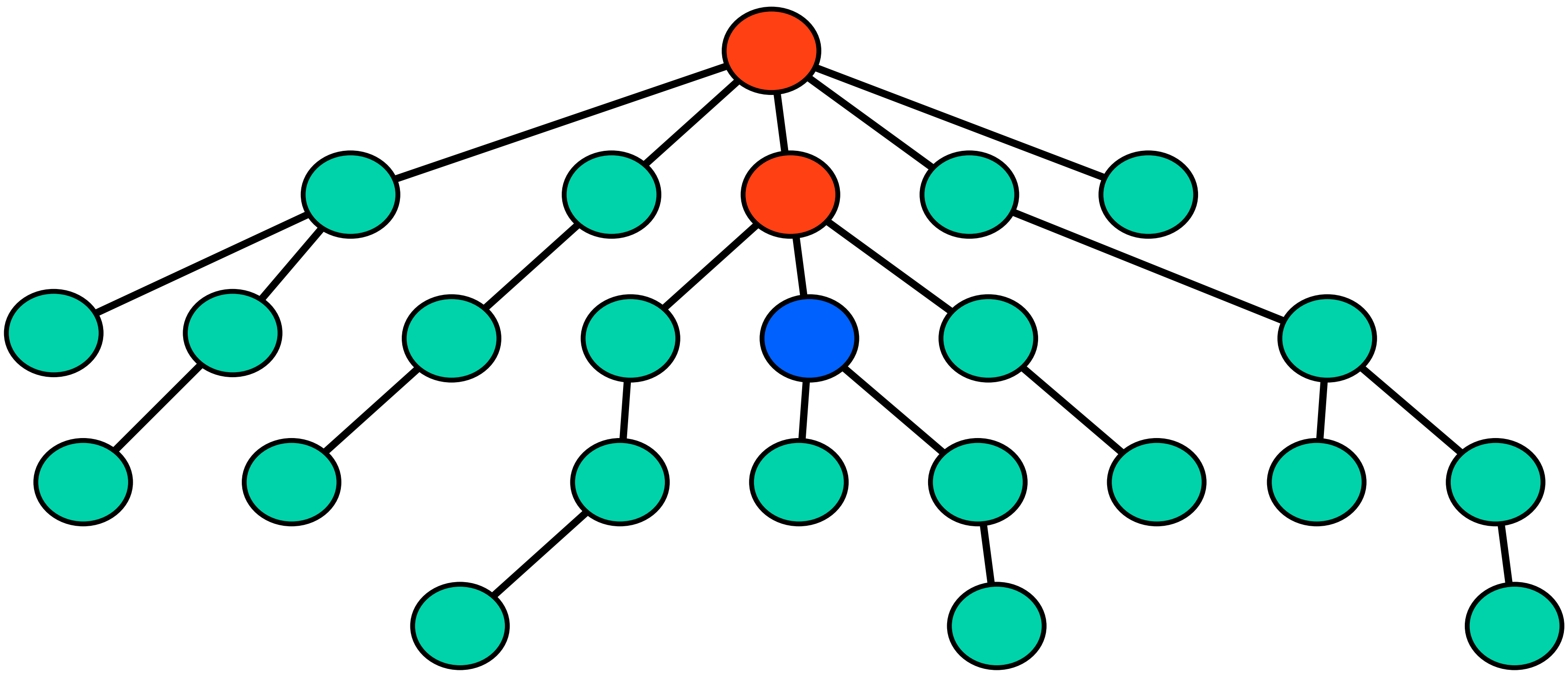
XPath

Parent



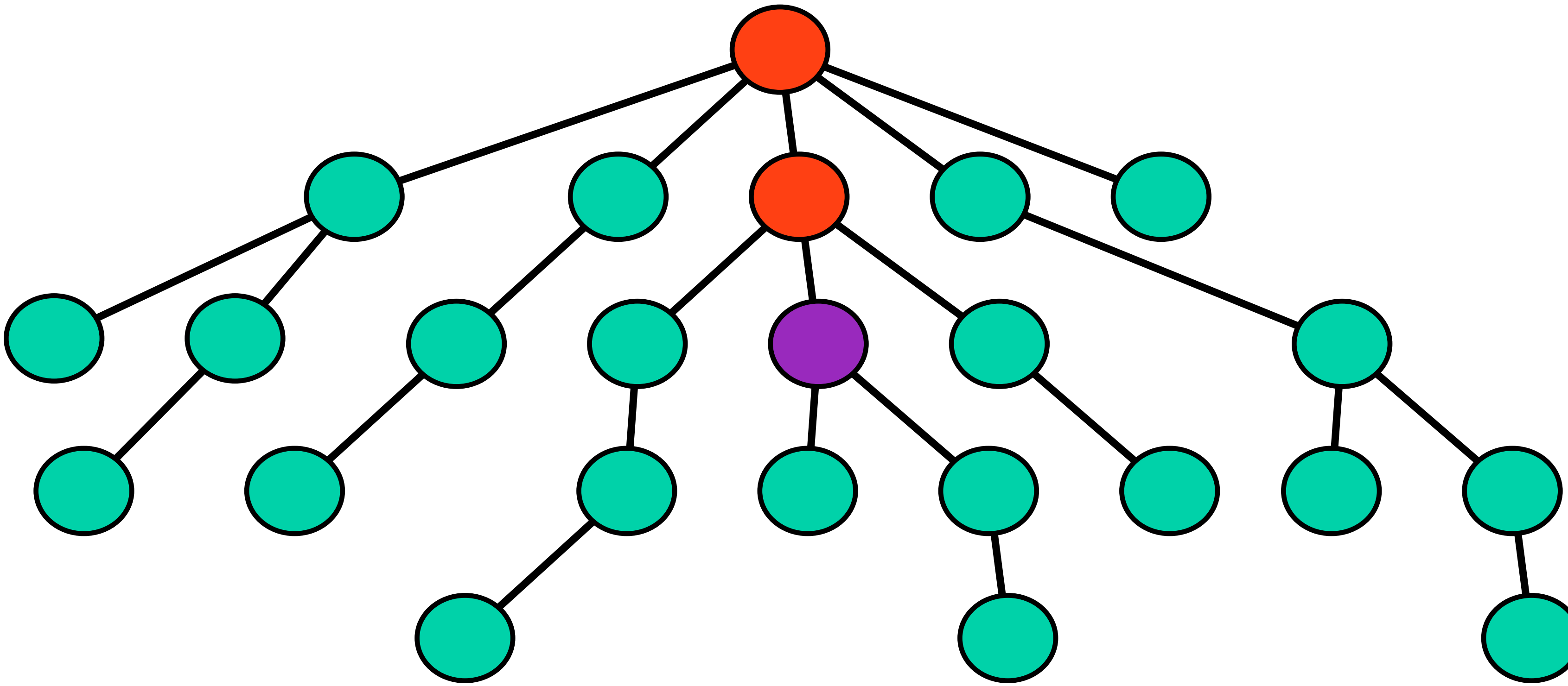
XPath

Ancestor



XPath

Ancestor-or-self



XPath

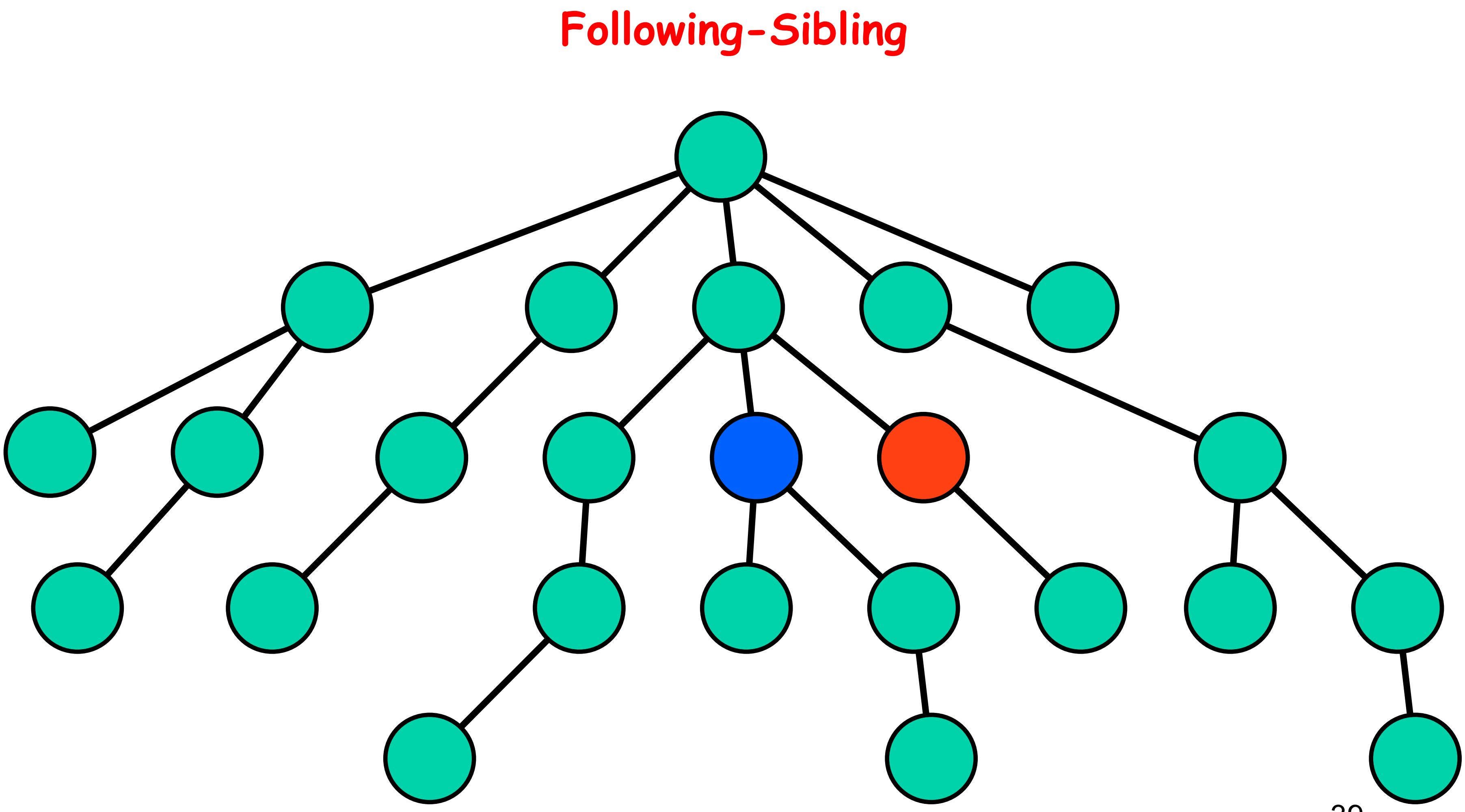
Sideways

Syntax:

Q ::= . . . | following::Q | preceding::Q |
following-sibling ::Q | preceding-sibling::Q |
[p] (p is integer)

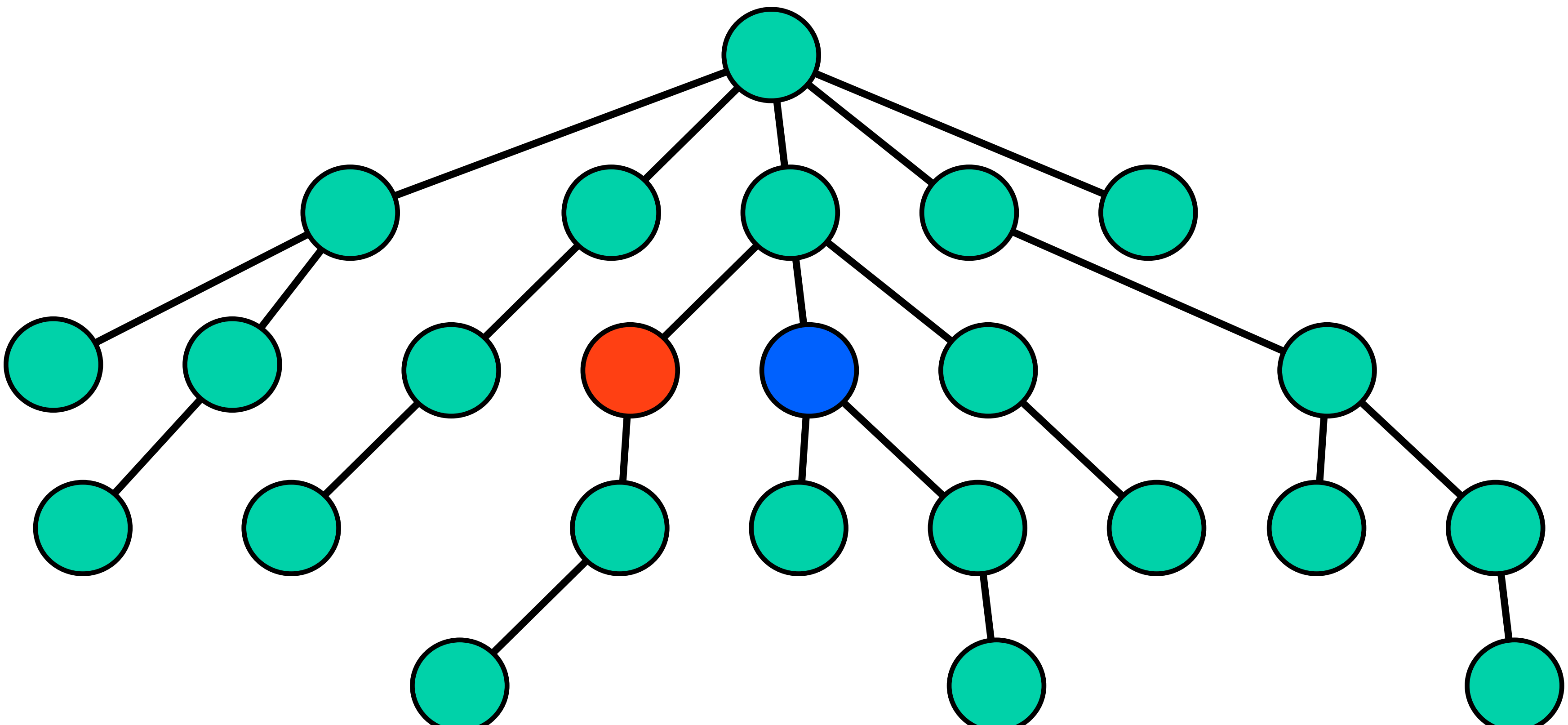
- ✓ **following-sibling**: the right siblings
- ✓ **preceding-sibling**: the left siblings
- ✓ **position function** (starting from 1): e.g., **//author[position( ) < 2]**

XPath



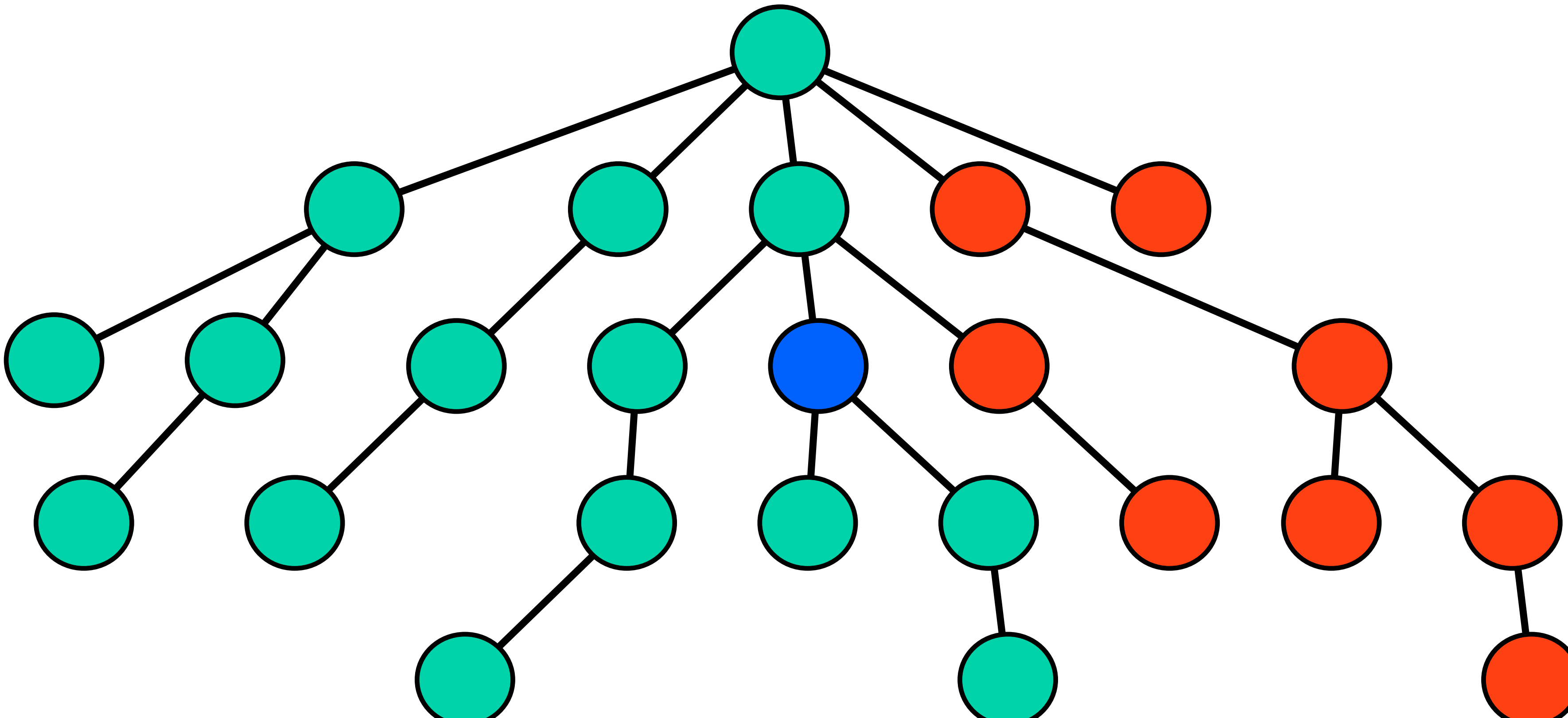
XPath

Preceding-Sibling



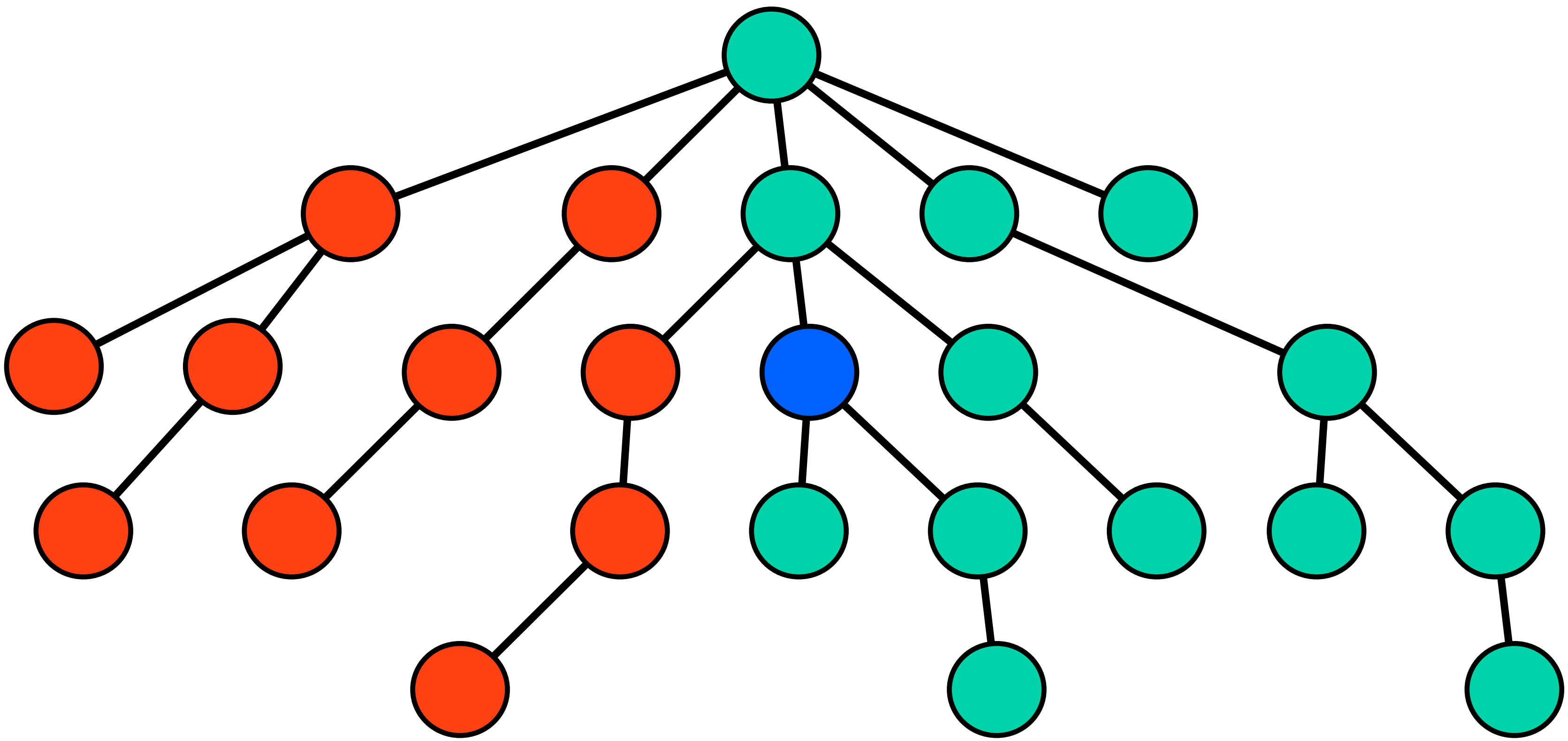
XPath

Following



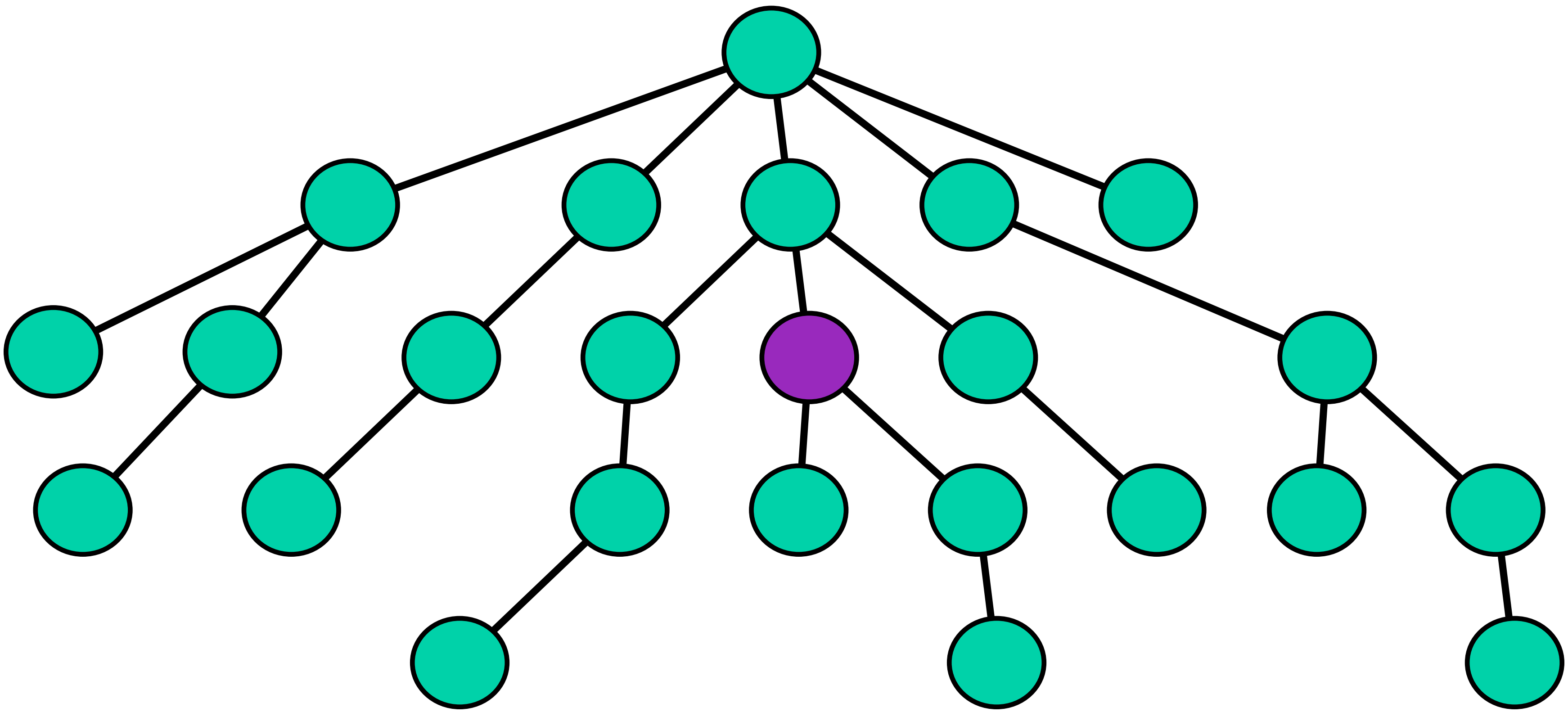
XPath

Preceding



XPath

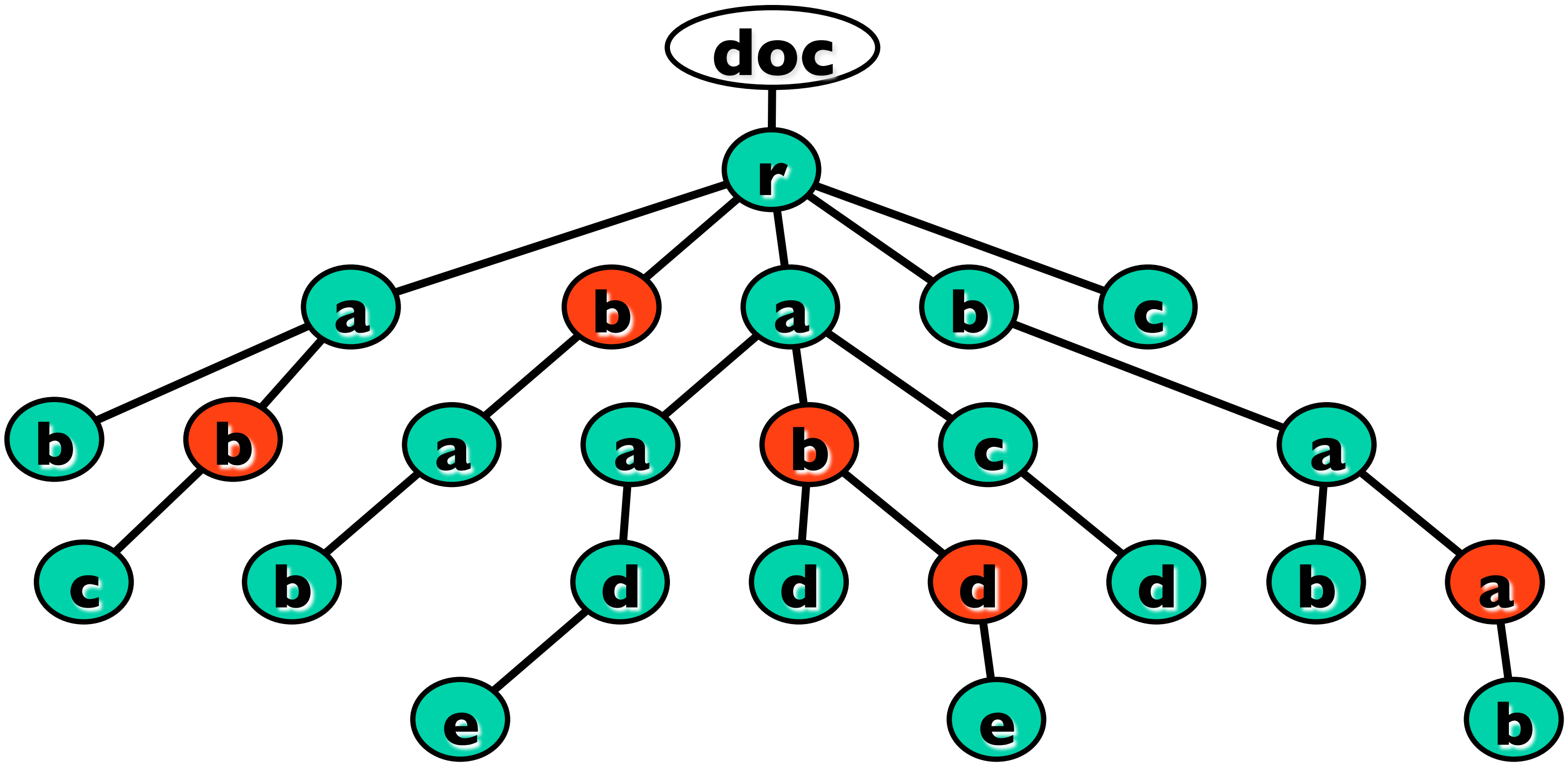
Self



XPath

Positional Tests

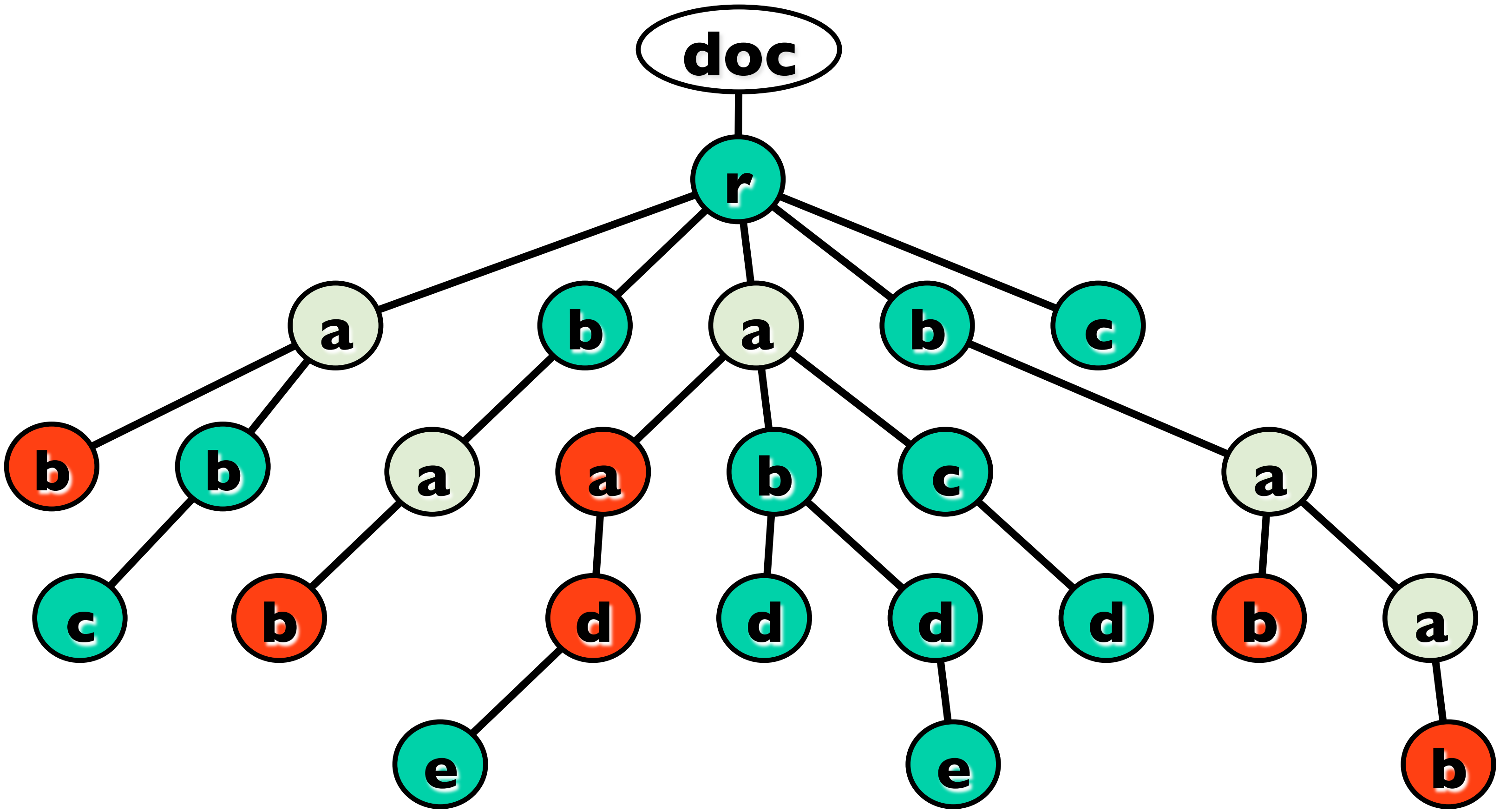
//*[position()=2] (or just **//*[2]**)



XPath

Positional Tests

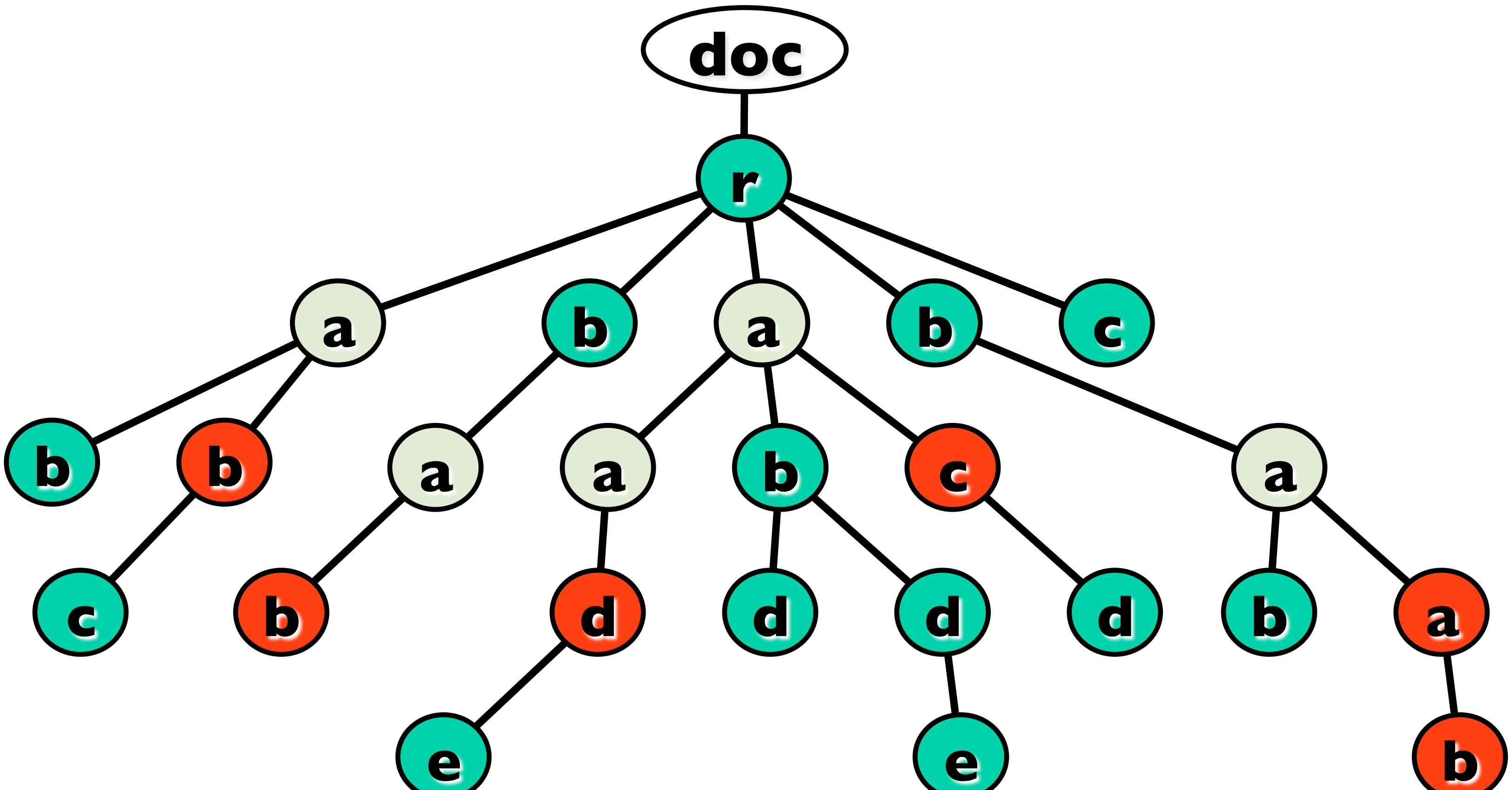
//a/*[first()]



XPath

Positional Tests

//a/*[last()]



Outline

- Overview on XML
- Why XML Security?
- Querying Views-based XML Data
- Updating Views-based XML Data

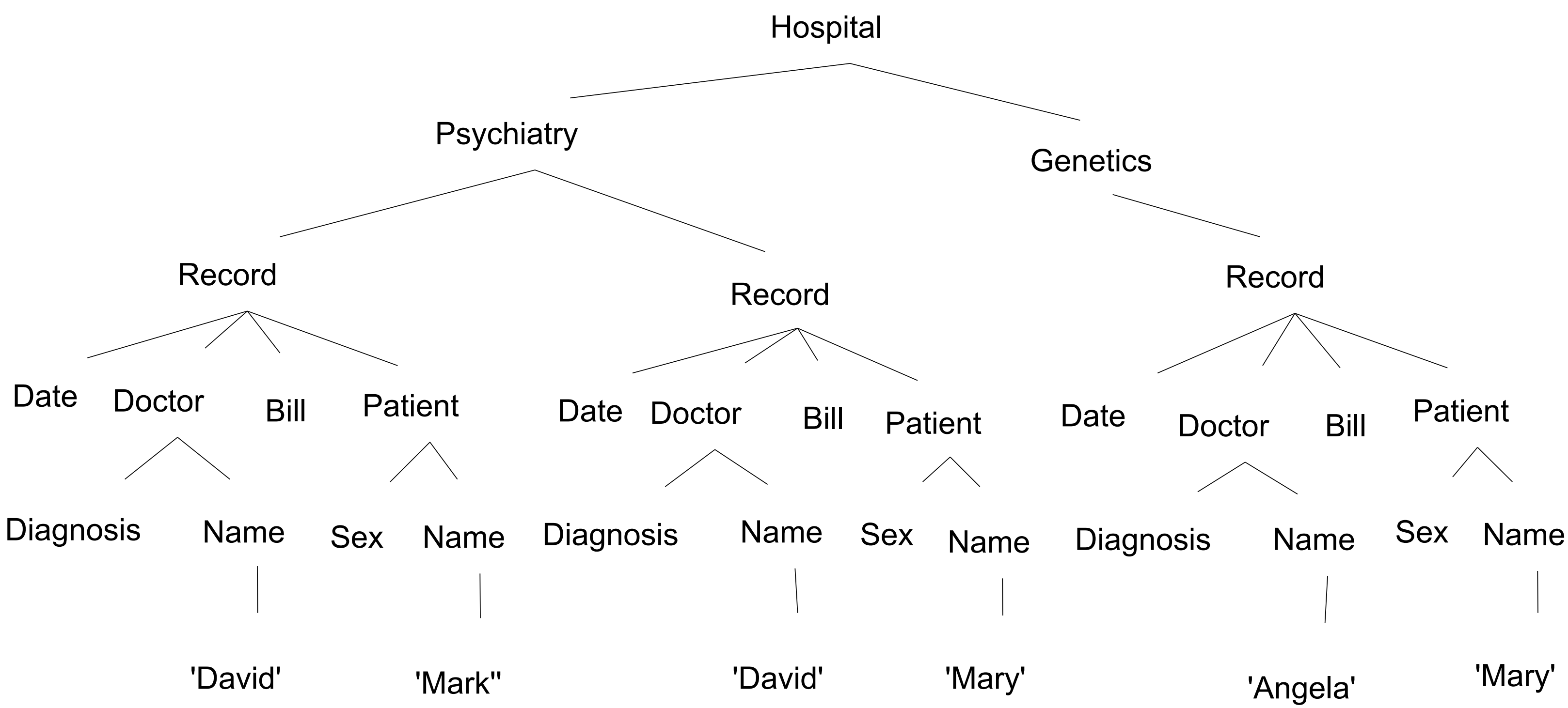
XML Security

- XML data management
  - Business information: Confidential
  - Health-care data: the Patient Privacy Act

- Selective divulgation of XML data
  - A major concern for data providers and consumers
  - Preserving data confidentiality, privacy and intellectual property

Example

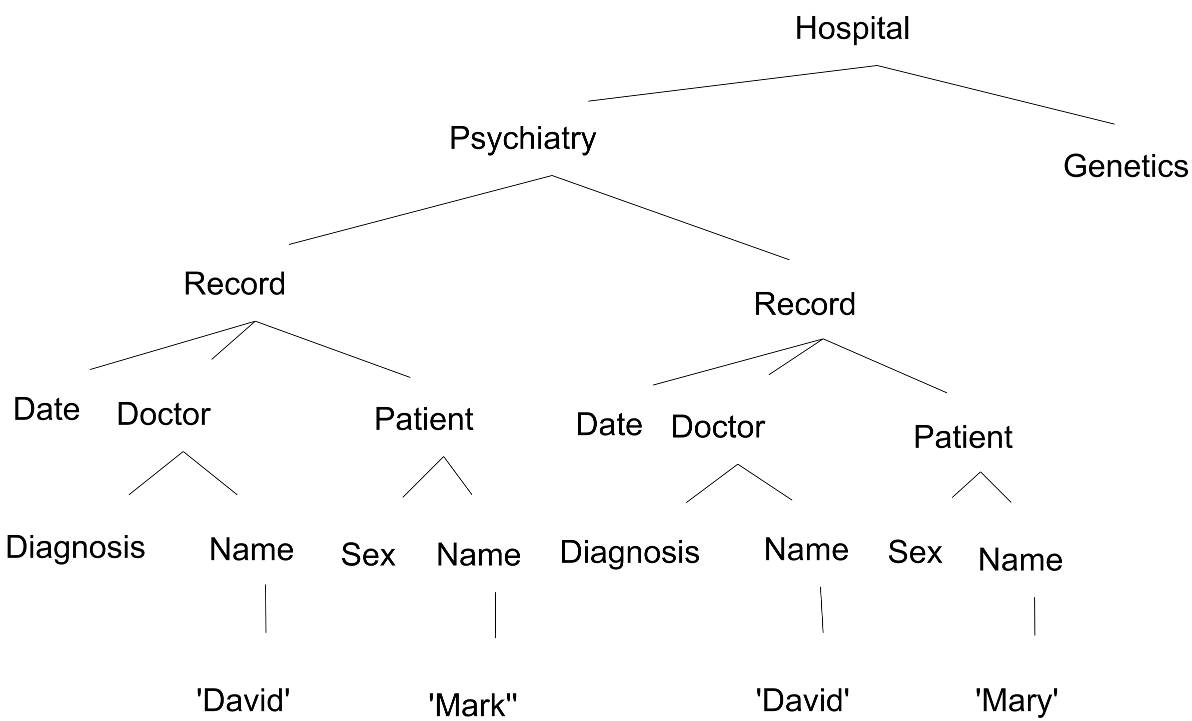
XML database containing medical records



The Administrator could see the whole database

Example

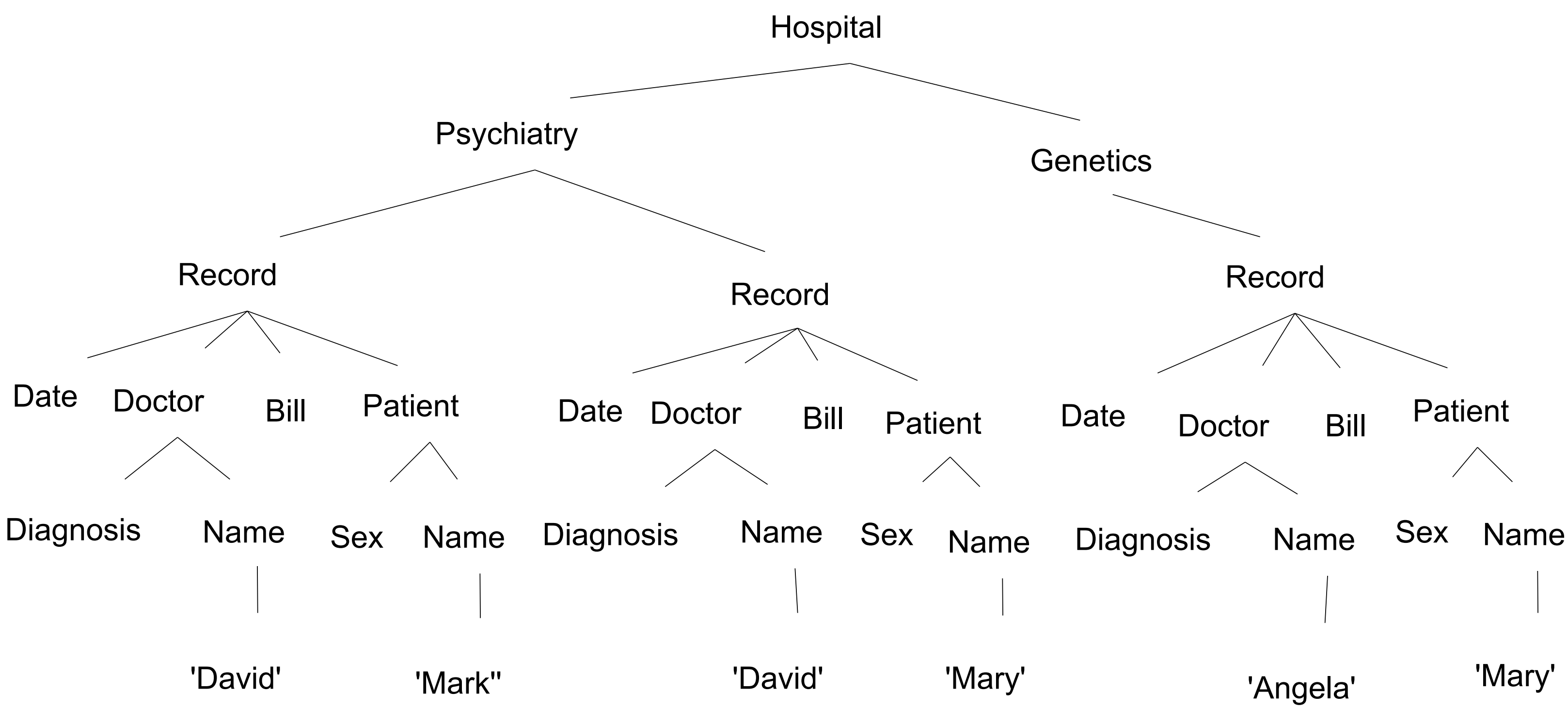
XML database containing medical records



Doctor David can only access the records of his patients

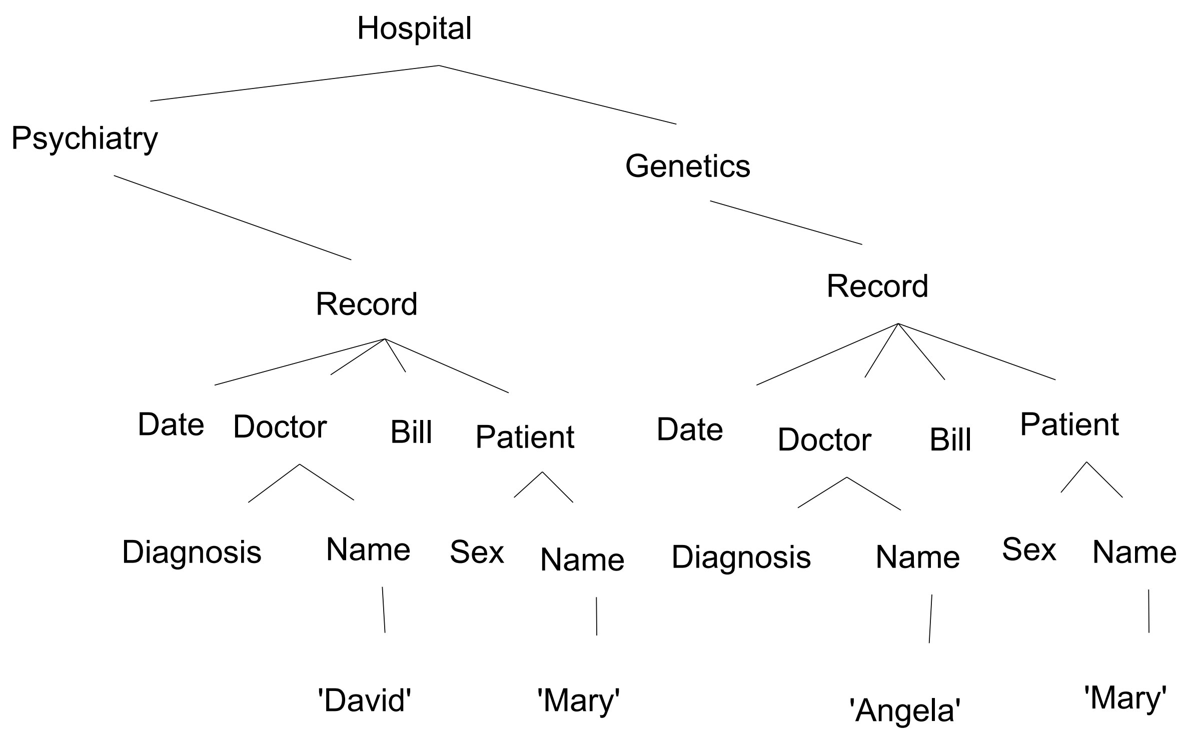
Example

XML database containing medical records



Example

XML database containing medical records

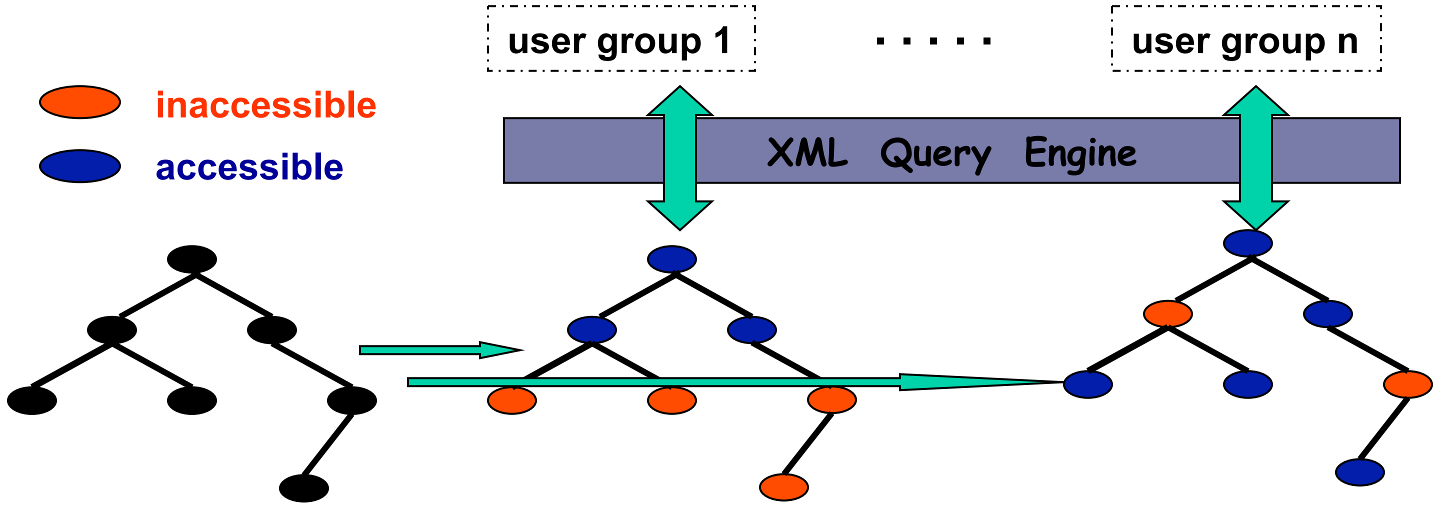


Patient Mary can access his own medical records

XML Access Control

- ✓ Access control
  - multiple groups simultaneously query the same XML document
  - each user group has a different access-control policy

- ✓ Enforcement of access-control policies:



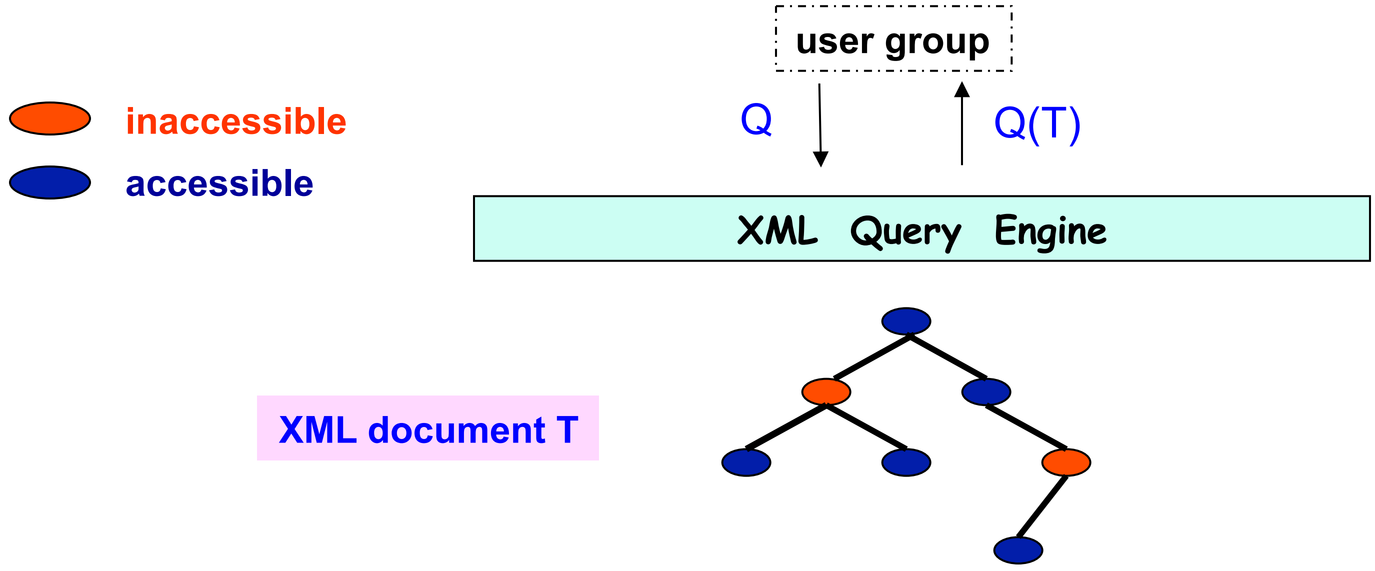
XML Access Control

For each user group of an XML document T,

- ✓ specify a access-control policy S,
- ✓ enforce S: for any query Q posted by the group over the document T, Q(T) consists of only data accessible w.r.t S

Problems with access control for XML:

- ✓ How to specify access policies at various levels of granularity?
- ✓ How to efficiently enforce those access policies?



Models for XML Security

Several models have been proposed for XML: XACML, XACL, …

- ✓ Specifying and enforcing access-control at a physical level
  - annotate data nodes in an XML document with accessibility, and check accessibility at runtime (with optimizations for tree-pattern queries and tree/DAG DTDs)
- ✓ Problems:
  - costly (time, space): multiple accessibility annotations
  - error-prone: integrity maintenance becomes a problem when the underlying data or access policy is updated

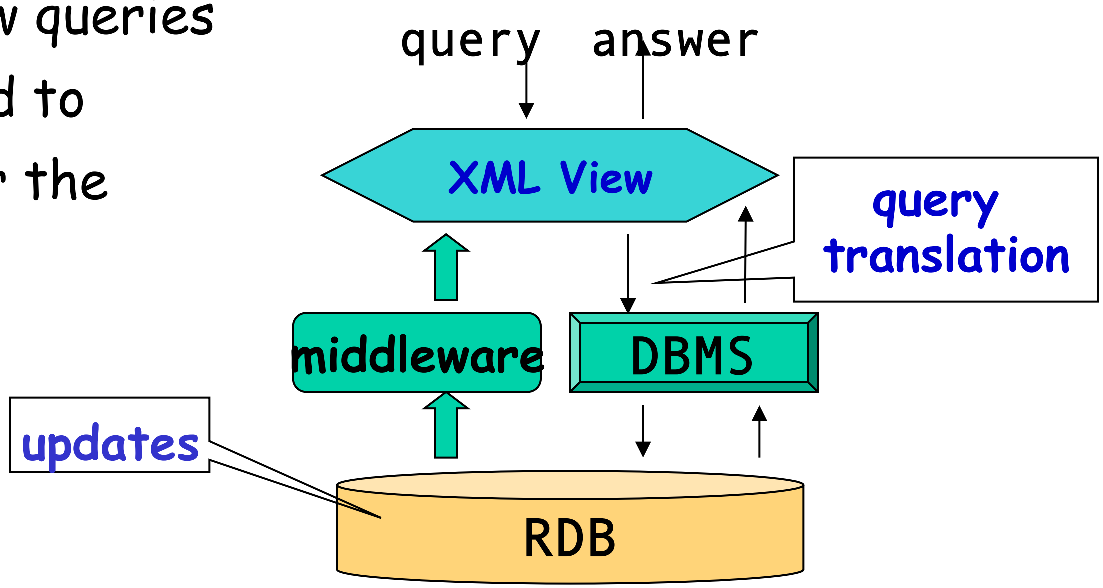
Models for XML Security

Several models have been proposed for XML: XACML, XACL, …

- ✓ Using at a Security Views: multiple user groups
  - who wish to query the same XML document
  - different access policies may be imposed, specifying **the portions of the document** the users are **granted** or **denied** access to.
- ✓ Two types of security views are used
  - Virtual views
  - Materialized views

XML Views

- Materialized views: store data in the views
  - Query support: straightforward and efficient
  - Consistency: the views should be updated in response to changes to the underlying database
- Virtual views: do not store data
  - Query support: view queries should be translated to equivalent ones over the underlying data
  - Updates: not an issue



Virtual vs. Materialized

XML views are important for data exchange, Web services, access control (security), Web interface for scientific databases, …

- ✓ Materialized views: publishing
  - sometimes necessary, e.g., XML publishing
  - when response time is critical, e.g., active system
  - "static": the underlying database is not frequently updated
- ✓ Virtual views: shredding
  - "dynamic": when the underlying data source constantly changes and/or evolves
  - Web interface: when the underlying database and the views are large
  - Access control: multiples views of the same databases are supported simultaneously for different user groups

Access Control Specification

Definition of rules for restricting access in XML data using various levels of granularity (entire subtrees or specific elements).

Each rule is a tuple of:

- ✓ Requestor
  - ✓ The user of set of users concerned by the authorization
- ✓ Resource
  - ✓ The data that the requestor is (or not) granted to access
- ✓ Action
  - ✓ The action (read, write, etc) is (or not) allowed on the resource
- ✓ Effect
  - ✓ It grants (sign '+') or denies (sign '-') access to the resource
- ✓ Propagation
  - ✓ It defines the scope of the rule

Language for Access Control

XPath language is used to specify the XML nodes concerned by an access rule.

Each rule's resource is defined as a XPath expression:

- ✓ Accessible /Inaccessible nodes
- ✓ Conditional accessible nodes

- ✓ XPath is a navigation language that returns a subset of nodes
  - ✓ It is used by XML-related technologies (XQuery, XSLT, etc)

- ✓ Different XPath fragments are used
  - ✓ Navigational axis (e.g. child, descendent, attributes, etc)
  - ✓ Comparison operators (e.g. testing only equality)
  - ✓ Expressions are absolute or relative

Scope for Access Control Rule

Due to the hierarchical nature of XML: how to apply the access rule?

The access rule is local if the scope can be:

- ✓ The node only
- ✓ The node and its attributes
- ✓ The node and its text value

The access rule is recursive if the scope can be:

- ✓ The node, its attributes, all its descendants and their attributes
- ✓ Entire sub-trees
- ✓ inheritance: some nodes inherit the accessibility of their ancestors

Default Semantics

Given an access control policy, there is a question:

What happens to the node if there exists no access control rule that neither grants nor denies access to it?

The default semantics of the access control policy gives an implicit rule in this case. There are two semantics:

- Deny
  - The node is non-accessible
- Grant
  - The node is accessible

Conflict Resolution

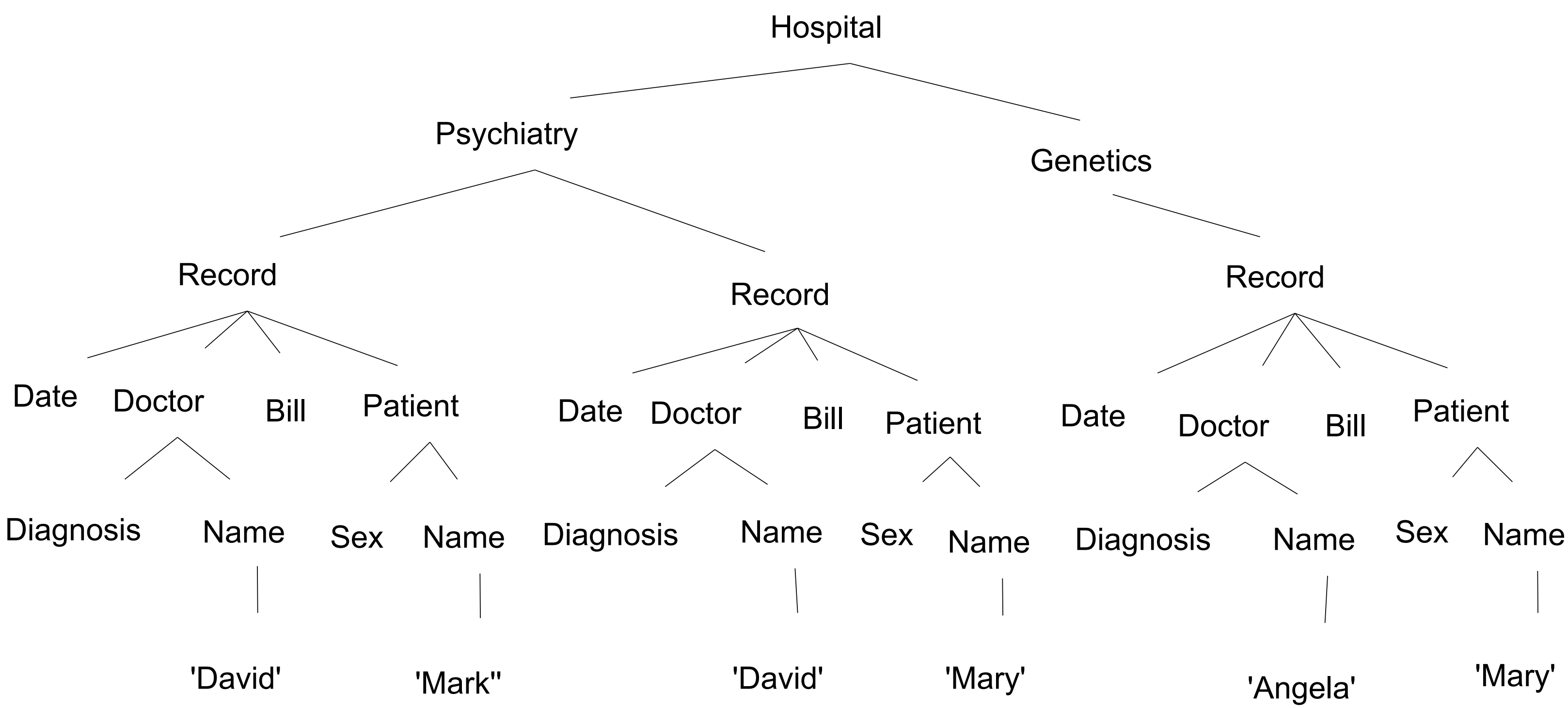
A conflict occurs when a node is granted access (by a positive rule) and denied access (by a negative rule) at the same time.

There are different approaches to perform conflict resolution:

- ✓ Priorities
  - ✓ Each rule is assigned a priority and the rule with highest priority is considered
- ✓ Deny overwrites
  - ✓ Negative rule takes precedence over positive rule
- ✓ Grant overwrites
  - ✓ Positive rule takes precedence over negative rule

Example

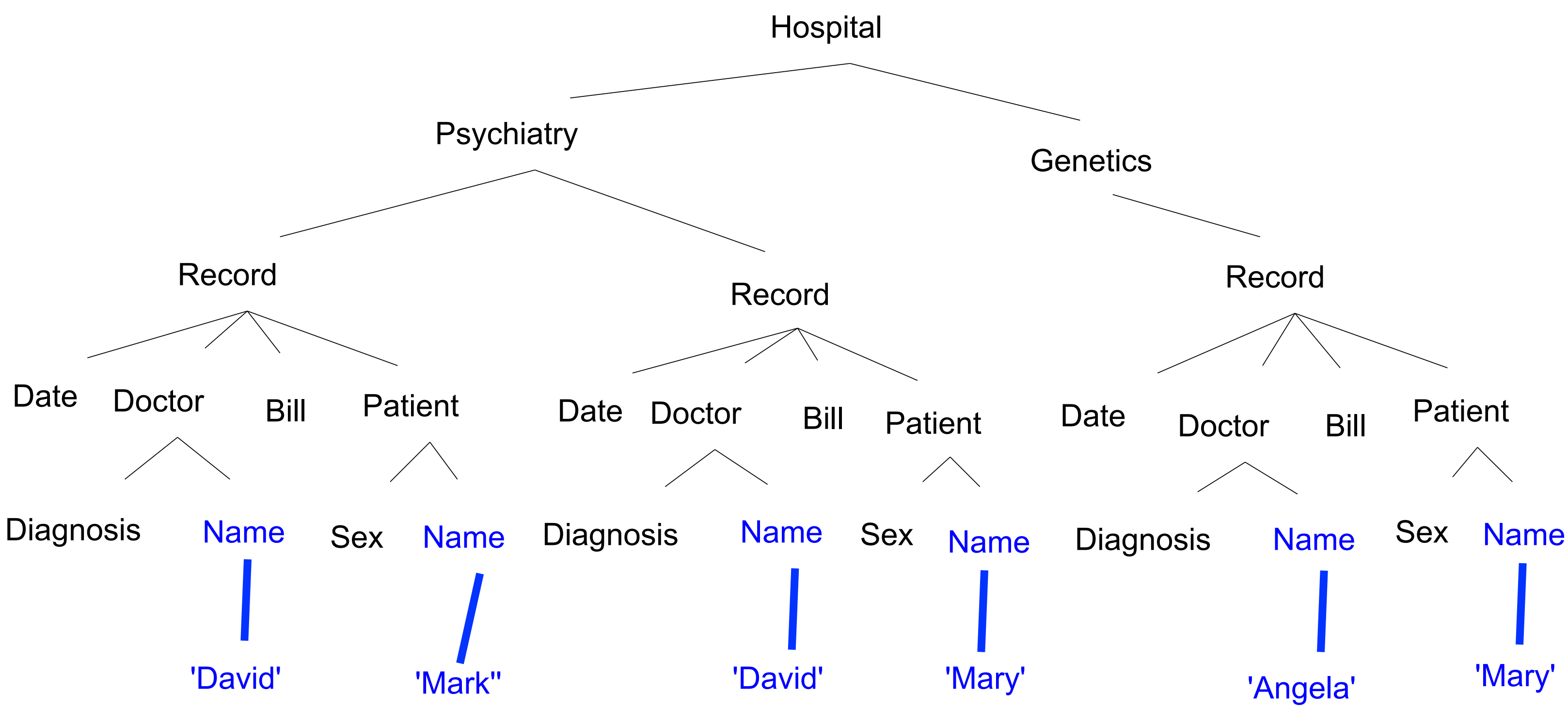
XML database containing medical records



Rule: (Toto, //Name, Read, +, local)
Default semantics: Deny

Example

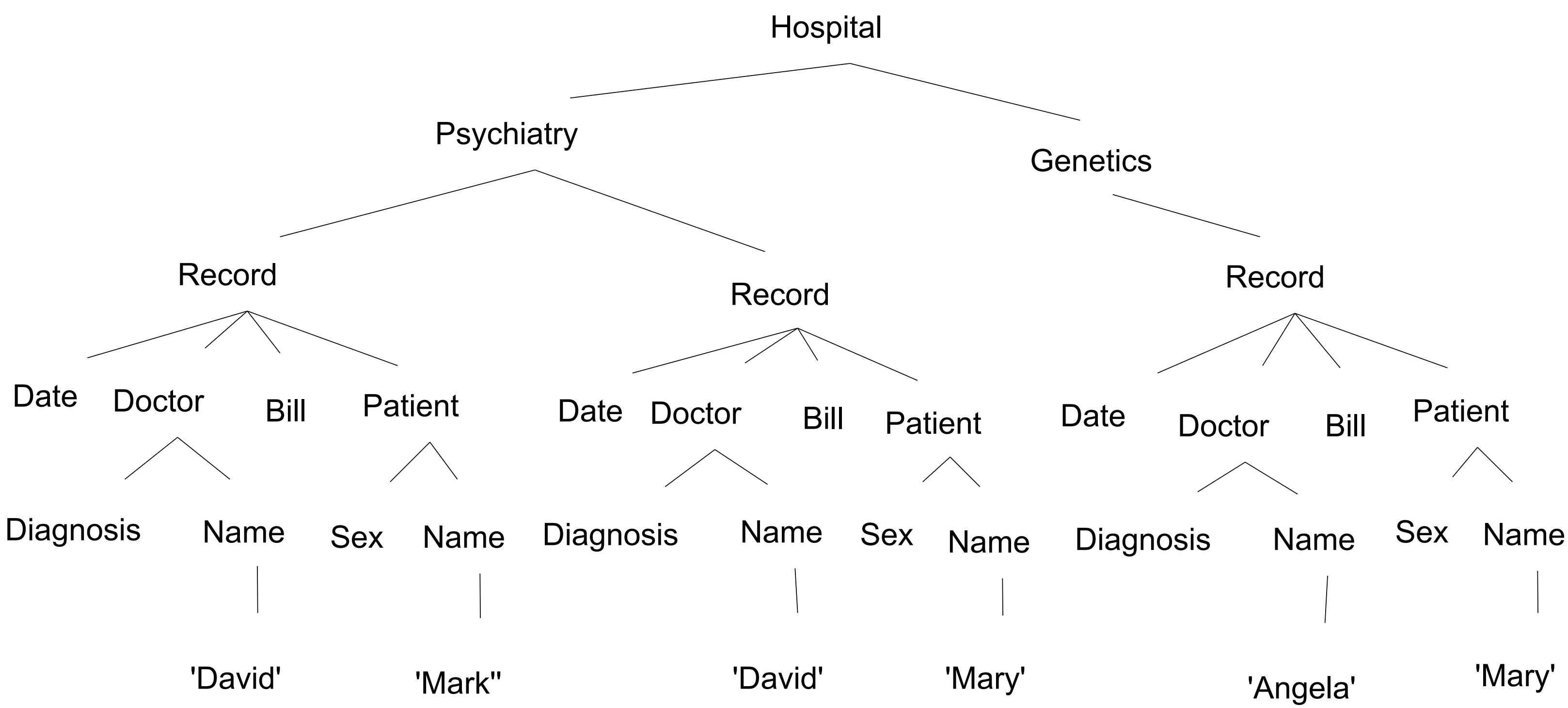
XML database containing medical records



Rule: (Toto, //Name, Read, +, local)
Default semantics: Deny

Example

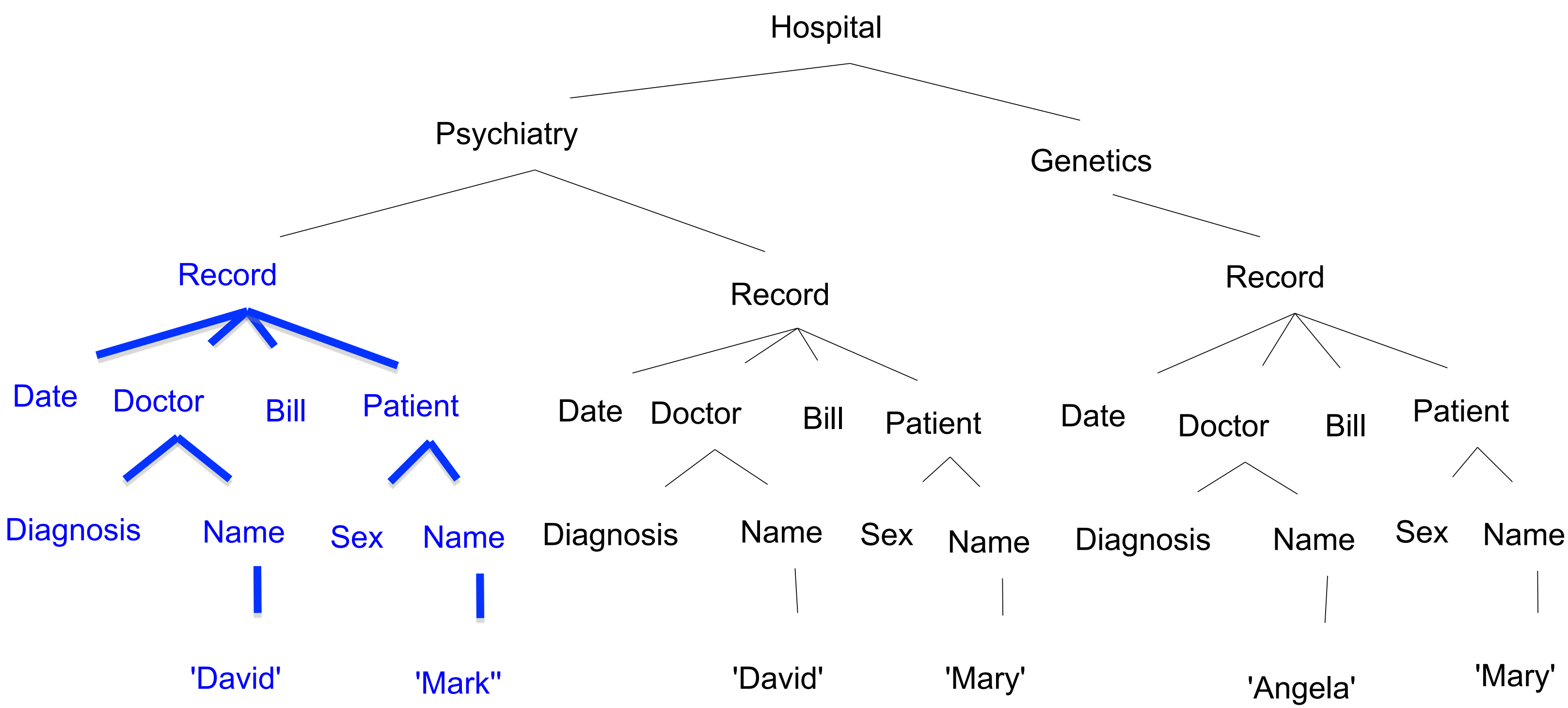
XML database containing medical records



Rule: (Toto, //Record[./Patient/Name='Mark'], Read, +, recursive)
Default semantics: Deny

Example

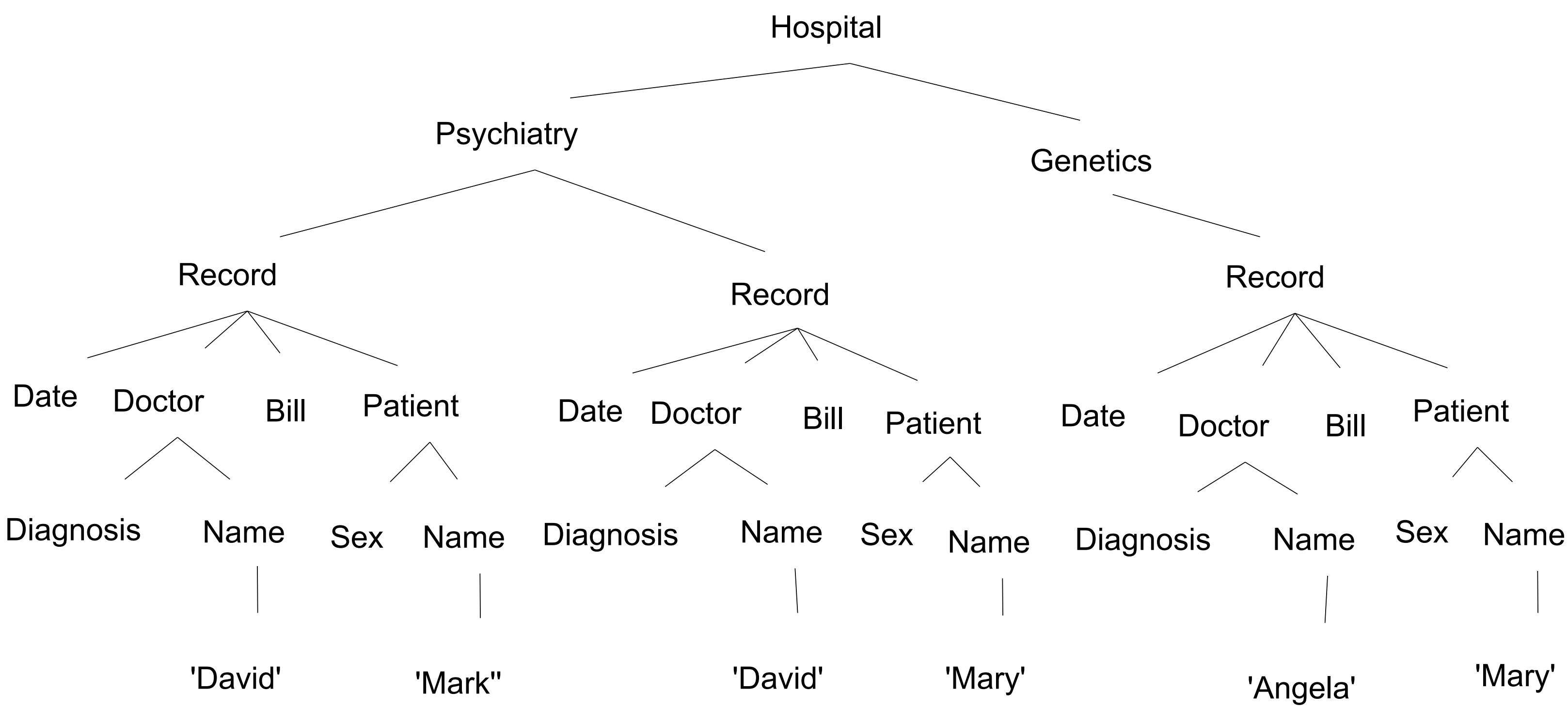
XML database containing medical records



Rule: (Toto, //Record[./Patient/Name='Mark'], Read, +, recursive)
Default semantics: Deny
Date, Doctor, Bill, Diagnosis, … inherit the accessibility of Record

Example

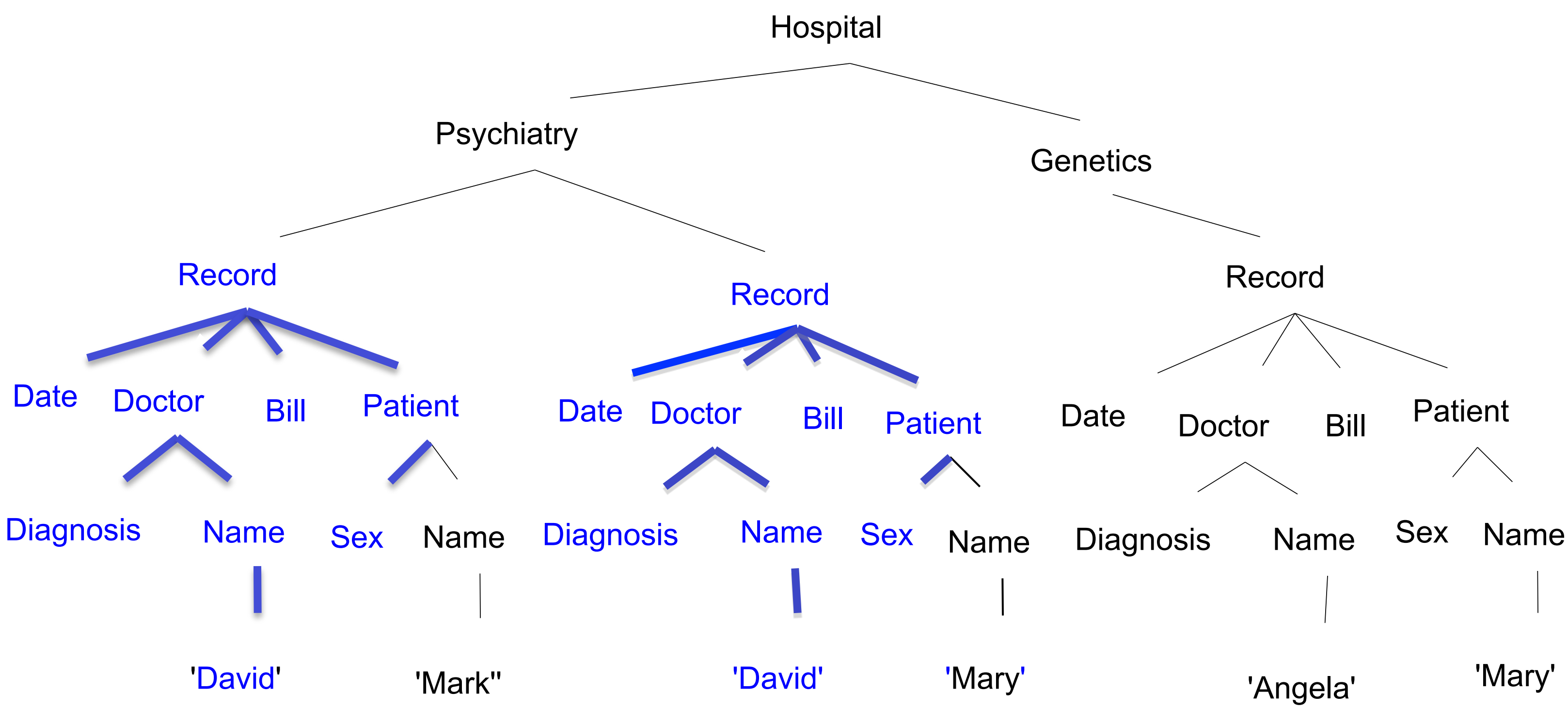
XML database containing medical records



Rule1: (Toto, //Patient/Name, Read, - , local)
Rule2: (Toto, //Record[./Doctor/Name='David'], Read, +, recursive)
Default semantics: Deny

Example

XML database containing medical records



Rule1: (Toto, //Patient/Name, Read, - , local)
Rule2: (Toto, //Record[./Doctor/Name='David'], Read, +, recursive)
Default semantics: Deny
Conflict resolution policy: Deny

Outline

- Overview on XML
- Why XML Security?
- Querying Views-based XML Data
- Updating Views-based XML Data

XML without Schema

Access control for XML Data proposed by Fundulaki *et* al. [Iri 2004].

- ✓ XPath fragment

  locapath ::= axis '::' ntst '[' expr ']' | '/' locapath | localpath '/' localpath
  expr ::= localpath | not expr | expr and expr | expr or expr
  | locapath *op v*

  ntst is a node label, * or function text()
  op is comparison operator (e.g. <=)
  v is a value

- ✓ Access Control Policy
  - ✓ Defined by four sets of XPath filter expressions

    P_l, P_r: positive local and recursive rules
    N_l, N_r: negative local and recursive rules

XML without Schema

Example: XML database containing medical records

1. Grant access to all nodes: P_r = {*}

2. Only Name nodes are accessible: P_l = {Name}

3. All nodes are accessible except Diagnosis: P_r = {*}, N_l = {Diagnosis}

4. Grant access to the Record nodes and all its descendant nodes, except if they are below a Patient node whose Name node has the value 'Mark':
   P_r = {Record}
   N_r = {Patient[./Name='Mark']}

XML without Schema

Enforcement of Access Control

- A XML document: D
- A query as a XPath expression: q
- An access control policy: ACP
- The query q is rewritten into q[expr] where expr is XPath expression, obtained from ACP in such a way the answer set of q must be filtered to obtain **only the accessible node**

XML without Schema

Access Control Policies with Only Local Rules

A node is accessible if there exists:

1. at least one positive rule that grants access to it, and
2. no negative rule that denies access to it

q[expr]

✓q targets element nodes

[expr] is $[\bigvee_{p \in \mathbf{P}_l} \text{self} :: p \bigwedge_{f \in \mathbf{N}_l} \text{not self} :: f]$

✓q targets attribute/text nodes

[expr] is $[\bigvee_{p \in \mathbf{P}_l} \text{parent} :: p \bigwedge_{f \in \mathbf{N}_l} \text{not parent} :: f]$

XML without Schema

Access Control Policies with Only Recursive Rules

A node is accessible if:

1. there exists a positive rule that grants access to one of its ancestors, or the node itself, and

2. no negative rule that denies access to one of its ancestors or the node itself

q[expr]

$$
\begin{aligned}
(1): &\quad [\bigvee_{p \in \mathbf{P}_r} \text{ancestor-or-self} :: p \\
(2): &\quad \bigwedge_{f \in \mathbf{N}_r} \text{not ancestor-or-self} :: f]
\end{aligned}
$$

XML without Schema

Access Control Policies with Local and Recursive Rules

A node is accessible if:

1. there exists at least one positive recursive rule that grants access to it, or
2. there exists at least one positive local rule that grants access to it, and
3. there is no negative recursive rule, and
4. there is no negative local rule that denies access to it

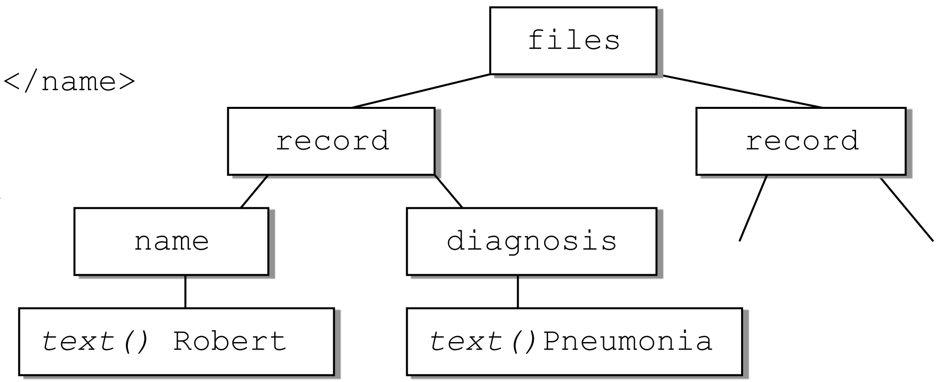
q[expr]

$$
\begin{array}{ll}
(1) & [(\bigvee_{p \in \mathbf{P}_r} \text{ancestor -or -self} :: p \text{ or} \\
(2) & \bigvee_{p \in \mathbf{P}_l} \text{self} :: p) \text{ and} \\
(3) & \bigwedge_{f \in \mathbf{N}_r} \text{ not ancestor -or -self} :: f \text{ and} \\
(4) & \bigwedge_{f \in \mathbf{N}_l} \text{ not self} :: f]
\end{array}
$$

XML without Schema

Problem: Security Breaches

```
<files>
 <record>
  <name>Robert</name>
  <diagnosis>
    Pneumonia
  </diagnosis>
 </record>
 <record …>
    …
 </record>
</files>
```



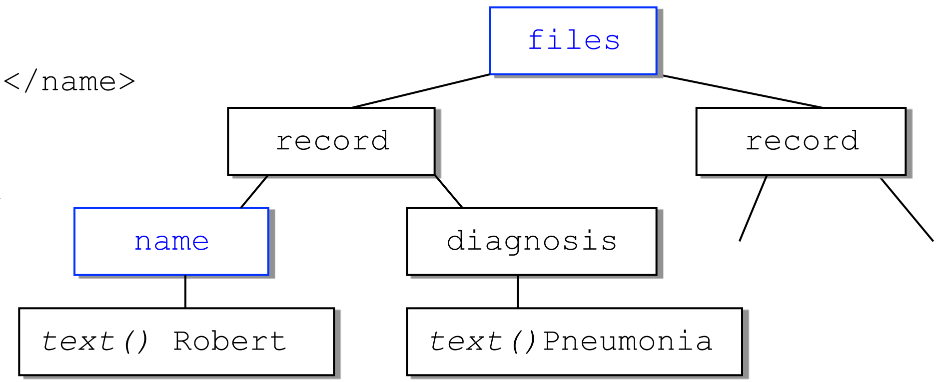
Only nodes files and name are accessible: P_l = {files}, P_l = {name}

Query /files/record/name is rewritten in /files/record/name[self::name]

XML without Schema

Problem: Security Breaches

```
<files>
 <record>
  <name>Robert</name>
  <diagnosis>
    Pneumonia
  </diagnosis>
 </record>
 <record …>
   …
 </record>
</files>
```



Only nodes files and name are accessible: P_l = {files}, P_l = {name}

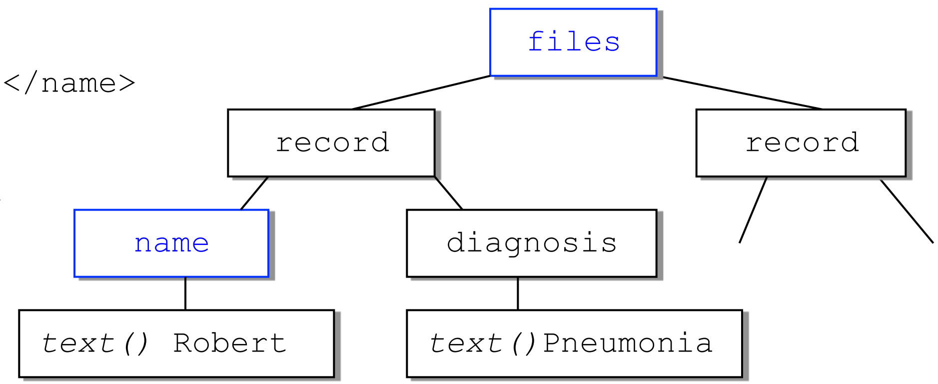
Query /files/record/name is rewritten in /files/record/name[self::name]

➔ Discloses the existence of hidden node

XML without Schema

Problem: Security Breaches

```
<files>
 <record>
  <name>Robert</name>
  <diagnosis>
    Pneumonia
  </diagnosis>
 </record>
 <record …>
   …
 </record>
</files>
```



Only nodes files and name are accessible: P_l = {files}, P_l = {name}

Query /files/record/name is rewritten in /files/record/name[self::name]

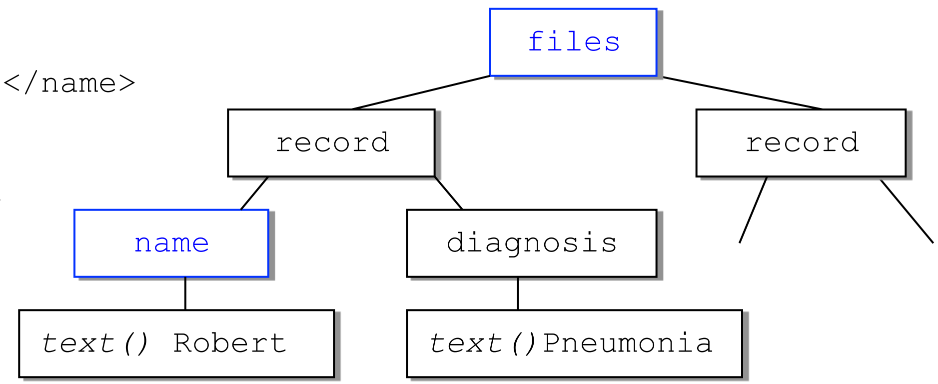
➔ Discloses the existence of hidden node

Solution: examining all nodes parsed in the query

XML without Schema

Problem: Rewriting may be impossible

```
<files>
 <record>
  <name>Robert</name>
  <diagnosis>
    Pneumonia
  </diagnosis>
 </record>
 <record …>
   …
 </record>
</files>
```



Only nodes files and name are accessible: P_l = {files}, P_l = {name}

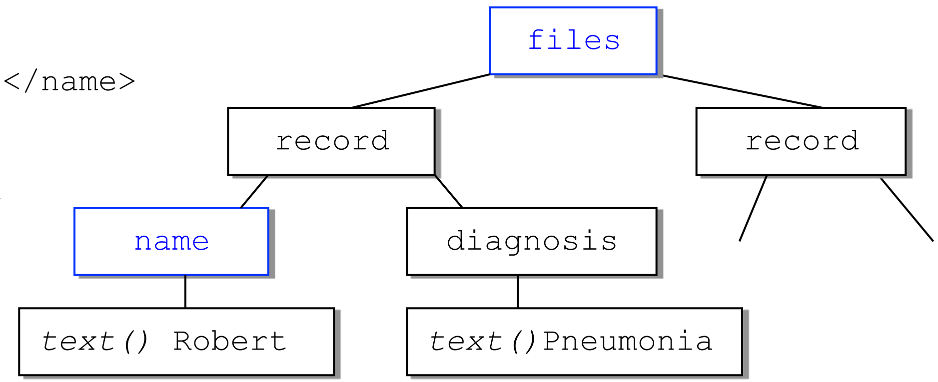
Query /files/name is rewritten in /files/name[self::name]

➔ This query will be rejected

XML without Schema

Problem: Rewriting may be impossible

```
<files>
 <record>
  <name>Robert</name>
  <diagnosis>
    Pneumonia
  </diagnosis>
 </record>
 <record …>
   …
 </record>
</files>
```



Only nodes files and name are accessible: P_l = {files}, P_l = {name}

Query /files/name is rewritten in /files/name[self::name]

➔ This query will be rejected

Solution: Denial Downward Consistency Property

if a node is inaccessible then all its descendants are inaccessible

XML with Schema

Access control for XML Data proposed by Fan *et* al. [Fan 2004].

- ✓ Security administrator: specifies a access-control policy for each group by extending the document DTD with XPath qualifiers
- ✓ Derivation module: automatically derives a security-view definition from each policy: view DTD and mapping via Xpath
- ✓ Query translation module: rewrite and optimize queries over views to equivalent queries over the underlying document

XML with Schema

Access control for XML Data proposed by Fan *et* al. [Fan 2004].

- ✓ Specification and enforcement: at the conceptual (schema) level
  - no need to update the underlying XML data
  - no need to materialize views or perform runtime check
- ✓ Schema availability: view schema is automatically derived
  - characterizing accessible data
  - exposing necessary schema information only

XML with Schema

Access control Specification

DTD D : element type definitions $A \rightarrow \alpha$

α ::= PCDATA | ε | A1, …, Ak | A1 + … + Ak | A*

Annotations are added in the DTD document to define the access control policy

$\boxed{\text{Access policy}} = \boxed{\text{Document DTD}} + \boxed{\text{XPath qualifiers}}$

XML with Schema

Access control Specification

✓Specification S = (D, access( )): a mapping access( ) from the edges in the DTD document D → { Y, N, [q] }.

For each $A \rightarrow \alpha$, for each B in α, define Access(A, B) as

- Y: accessible (true)
- N: inaccessible (false)
- [q]: XPath qualifier, conditional: accessible iff [q] holds

XPath fragment:

$$p ::= \varepsilon \mid A \mid * \mid // \mid p/p \mid p \cup p \mid p[q]$$

$$q ::= p \mid p = \text{“c”} \mid q1 \wedge q2 \mid q1 \vee q2 \mid \neg q$$

XML with Schema

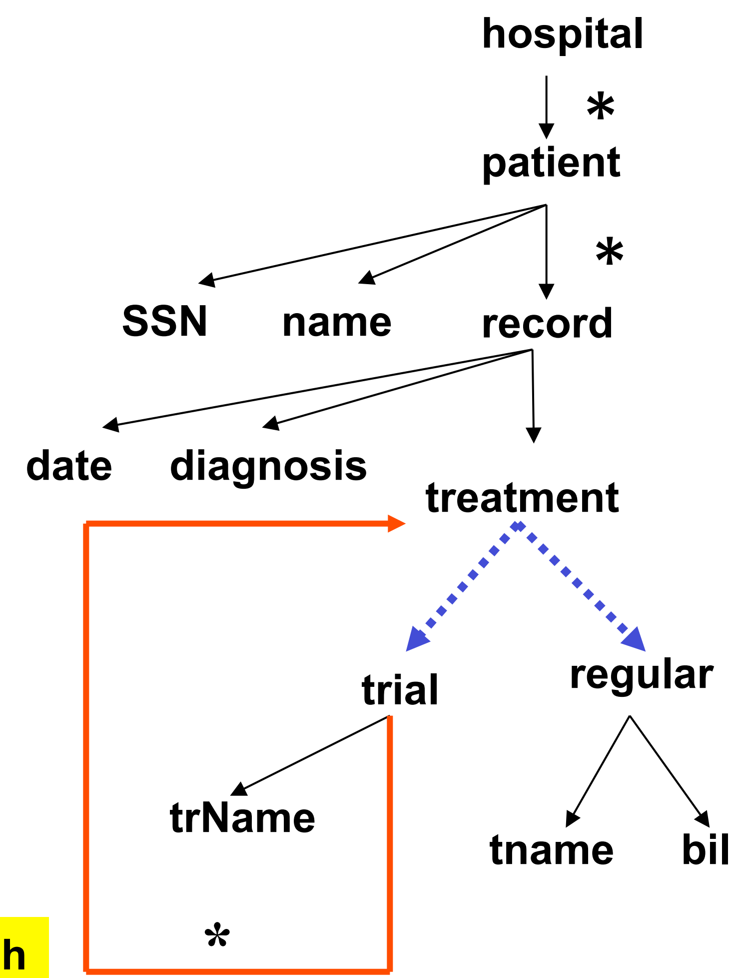
Example: an XML document of patients

Document DTD D

hospital	→	patient*
patient	→	SSN, name, record*
record	→	date, diagnosis, treatment
treatment	→	(trial + regular)
trial	→	trName, treatment*
regular	→	tname, bill

Access-control policies over docs of D:

- ✓ Doctors in the hospital are granted access to all the data in the docs
- ✓ Insurance company is allowed to access billing information only



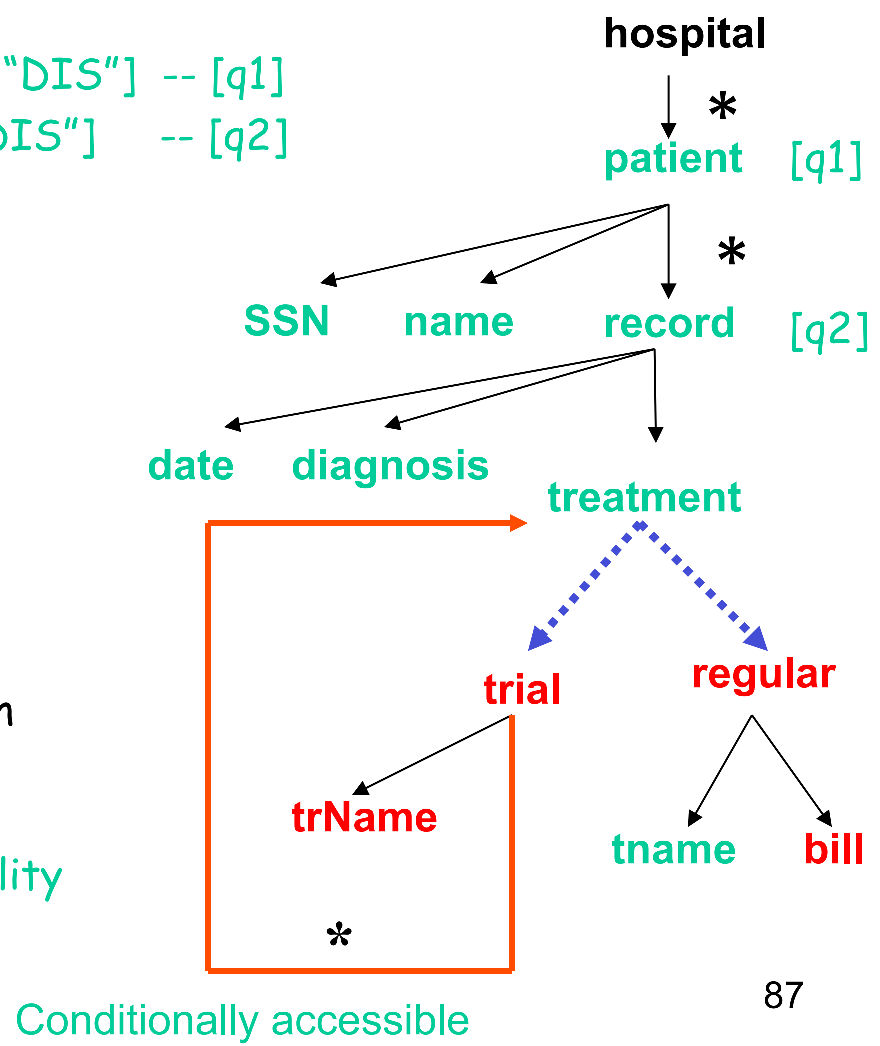
DTD graph

XML with Schema

Example: an XML document of patients

access(hospital, patient) = [//diagnosis = "DIS"] -- [q1]
access(patient, record) = [diagnosis = "DIS"] -- [q2]
access(treatment, trial) = N
access(treatment, regular) = N
access(regular, tname) = Y

- ✓ **overriding**: if access(A, B) = Y (N), then the B children of A override the accessibility of A
- ✓ **inheritance**: if access(A, B) is not explicitly defined, then the B children of A inherit the accessibility of A
- ✓ **content-based**: conditional accessibility via XPath qualifiers



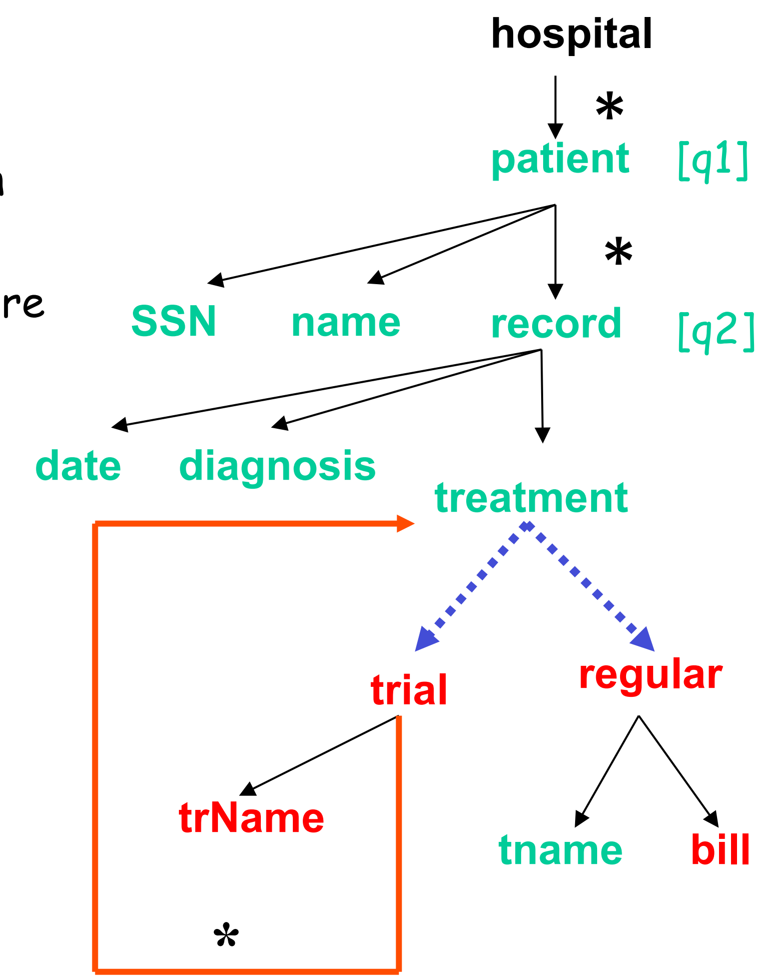
XML with Schema

Properties of the specification language

- ✓ XML tree of the document DTD:
  the accessibility of each data node
  is uniquely defined by an access specification
  - relative to the path from root
  - a qualifier at a node **a** constrains the entire subtree rooted at **a**,
    - e.g., [q2] constrains tname

- ✓ various levels of granularity: entire subtrees or specific elements

- ✓ schema level: the underlying XML data is not touched; efficient, easy to specify and maintain



Conditionally accessible

XML with Schema

Enforcing Access Control – Security Views

XML security view: σ = (Dv, xpath( )) with respect to an access policy S = (D, access( )),

- ✓ Dv: view DTD, exposed to the user and characterizing the accessible information (of document DTD D) w.r.t S

  Schema availability: to facilitate query formulation

- ✓ xpath( ): mapping from instances of D to instances of Dv defined in terms of XPath queries and view DTD Dv
  - for each $A \rightarrow \alpha$ in Dv, for each B in α, xpath(A, B) = p
  - p: generates B children of an A element in a view

$$p ::= \varepsilon \mid A \mid * \mid // \mid p/p \mid p \cup p \mid p[q]$$

$$q ::= p \mid p = \text{“c”} \mid q1 \wedge q2 \mid q1 \vee q2 \mid \neg q$$

XML with Schema

Derivation of Security Views

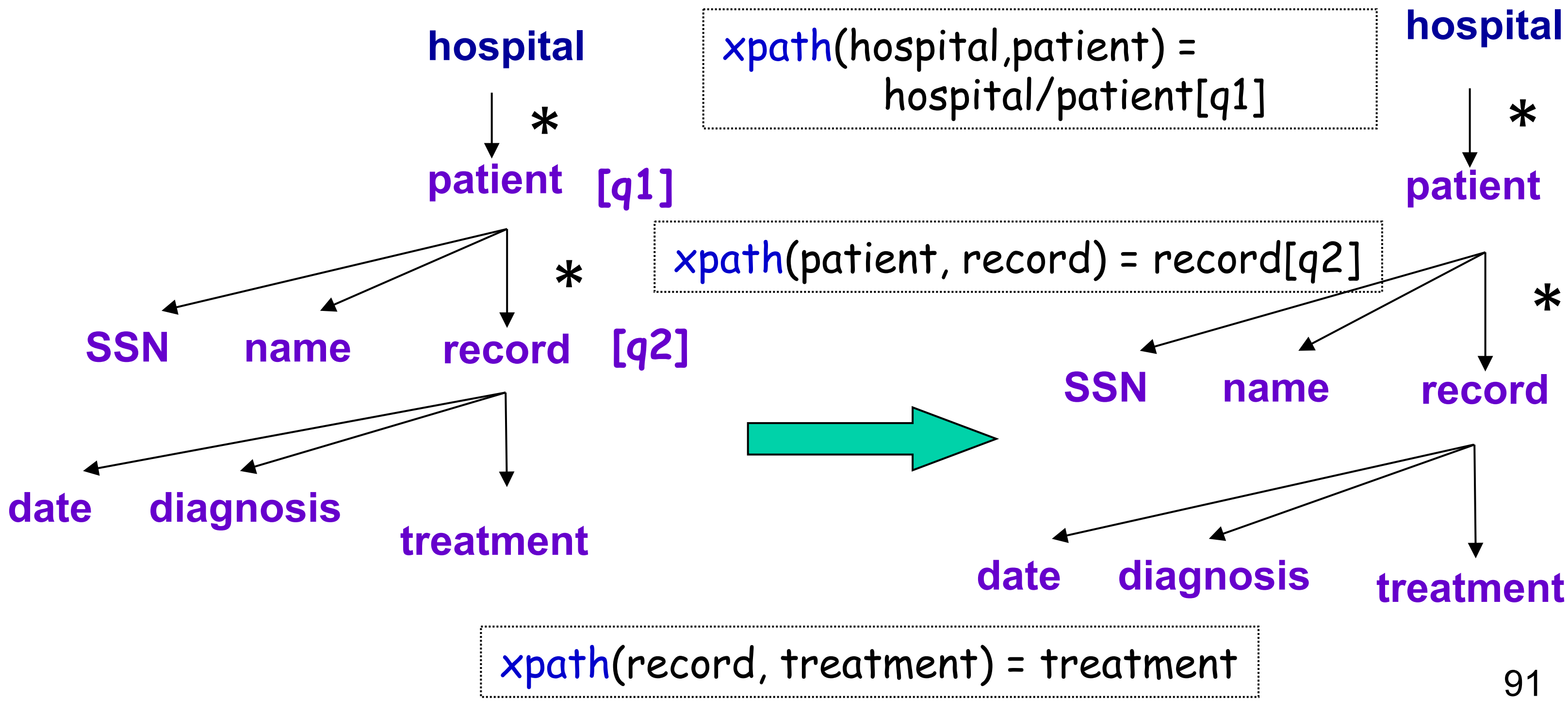
One needs an algorithm to compute a security-view definition:

- ✓ Input: an access policy S = (D, access( ))
- ✓ Output: a security-view definition σ = (Dv, xpath( ))
  - sound: accessible information only
  - complete: all the accessible data (structure preserving)
  - DTD-conformant: conforming to the view DTD
- ✓ efficient: $O(|S|^2)$ time (proposed in [Fan2004])
- ✓ generic: recursive/nondeterministic document DTDs

XML with Schema

Example: an XML document of patients

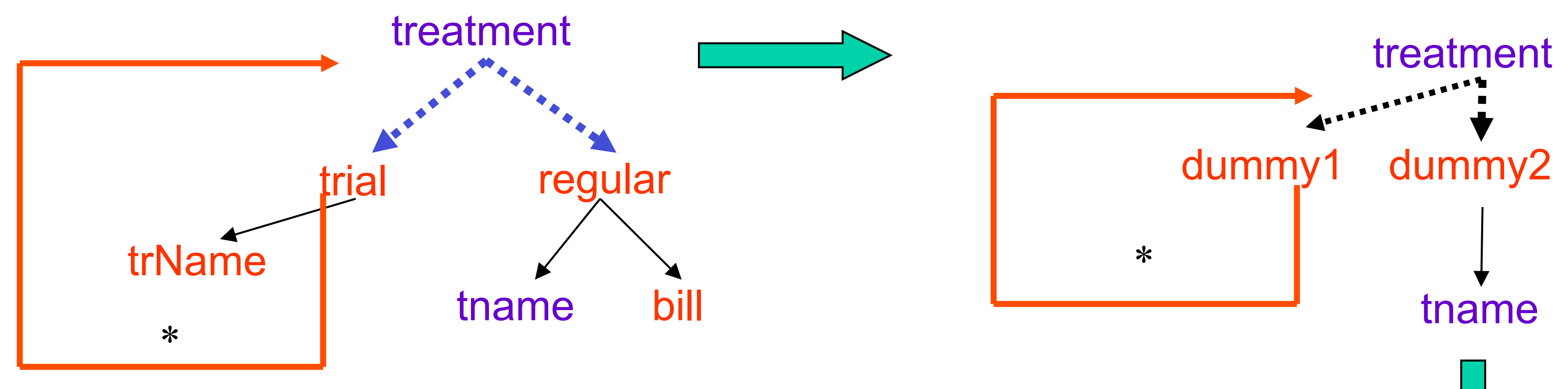
- ✓ Top-down traversal of the document DTD D
- ✓ short-cutting/renaming (via dummy) inaccessible element types
- ✓ normalizing the view DTD Dv and reducing dummy types



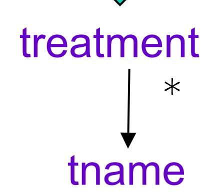
XML with Schema

Example: an XML document of patients

- ✓ recursive and non-deterministic productions



xpath(treatment, dummy2) = regular
xpath(treatment, dummy1) = trail



- ✓ reducing dummy element types:

(dummy1/treatment)* / dummy2 / tname ∪ dummy2/tname)

⇒ (dummy1/treatment)* / dummy2 / tname ⇒ tname*

xpath(treatment, tname) = //tname

XML with Schema

Rewriting Algorithm

✓Input:

- σ = (Dv, xpath( )) (security view wrt S = (D, access( ))), and
- an XPath query Qv over the view (Dv)

✓Output: an equivalent XPath query Qt over the document

- for any XML document T of D, Qt(T) = Qv(σ(T))

Dynamic programming:

✓for any subquery Qv' of Qv, any node A in view-DTD graph Dv

rewrite Qv' at A by incorporating xpath(A, _) $\Rightarrow$ Qt' (A)

✓efficient: O(|Qv| | σ |^2) time

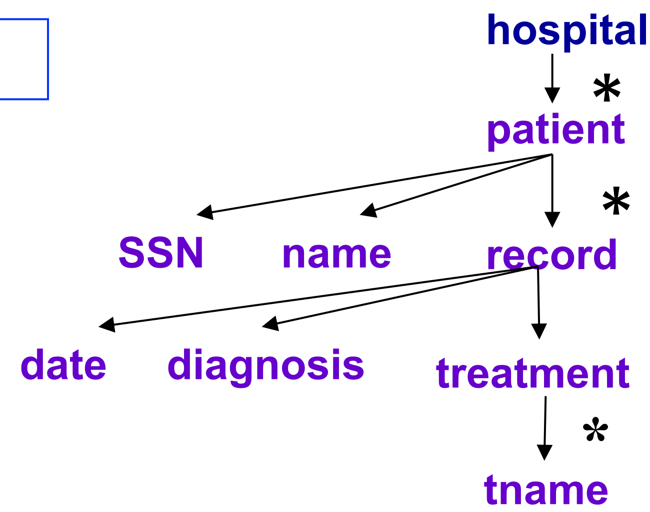
✓a practical class of XPath (with union, descendant, qualifiers) vs. tree-pattern queries studied in previous security models

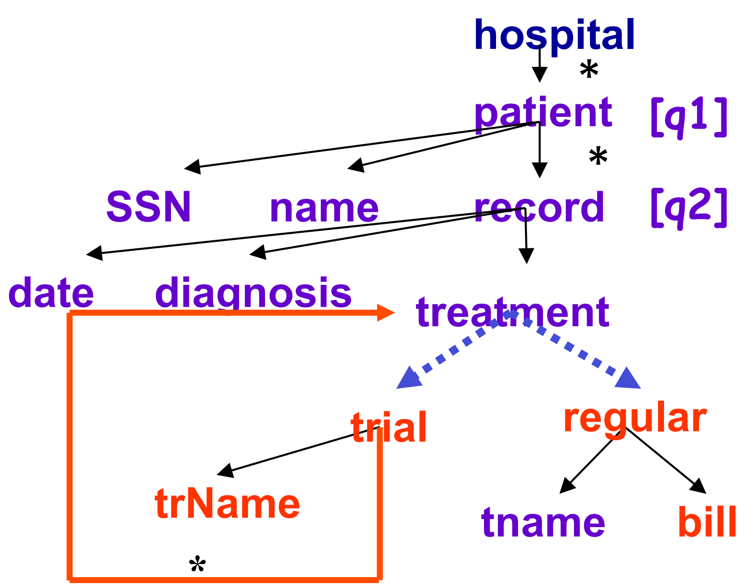
XML with Schema

Example: an XML document of patients

Qv = // patient[name="Joe"] // tname over the view

xpath(hospital, patient) [name = "Joe"] /
    xpath(patient, record) /
    xpath(record,treatment) /
    xpath(treatment, tname)





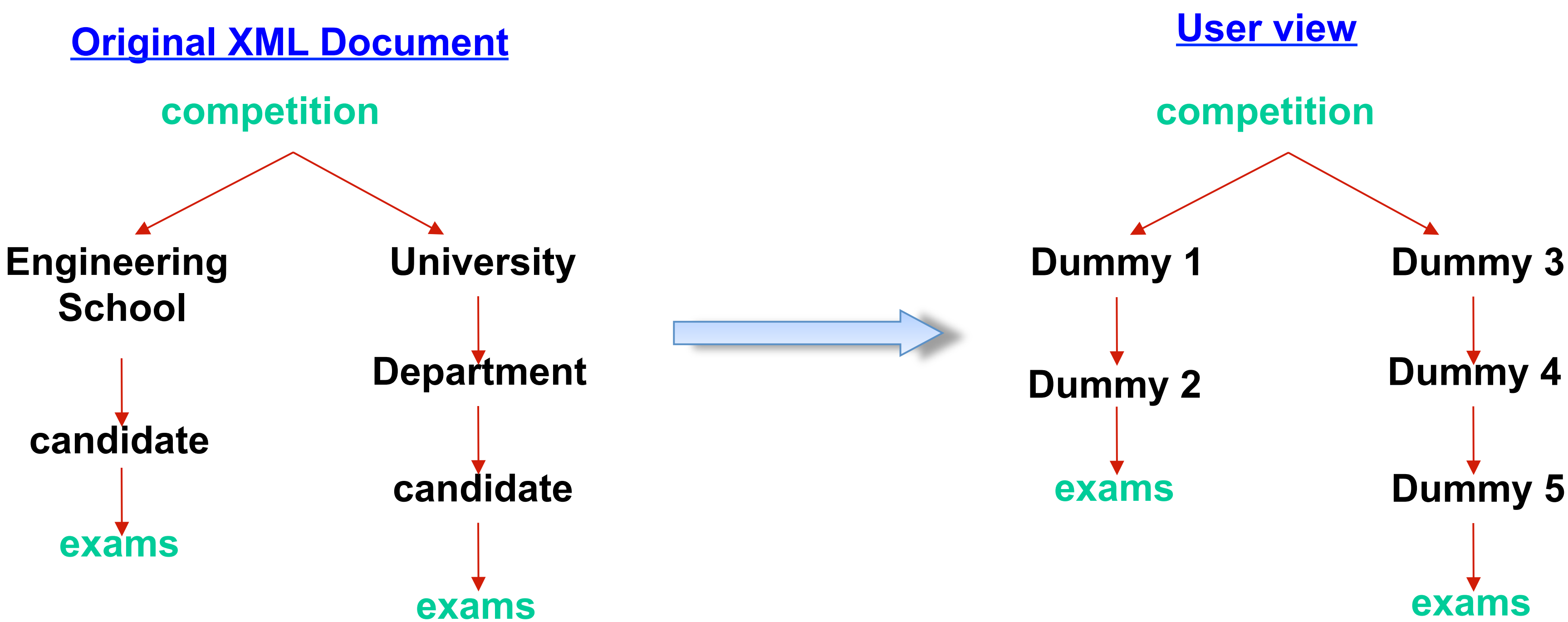
Qt = /hospital/patient[name = "Joe"
    and //diagnosis = "DIS"]
  /record[diagnosis = "DIS"]
  /treatment // tname

equivalent query over document

XML with Schema

Problems when using “dummy” nodes

Replacing inaccessible nodes with anonymous nodes

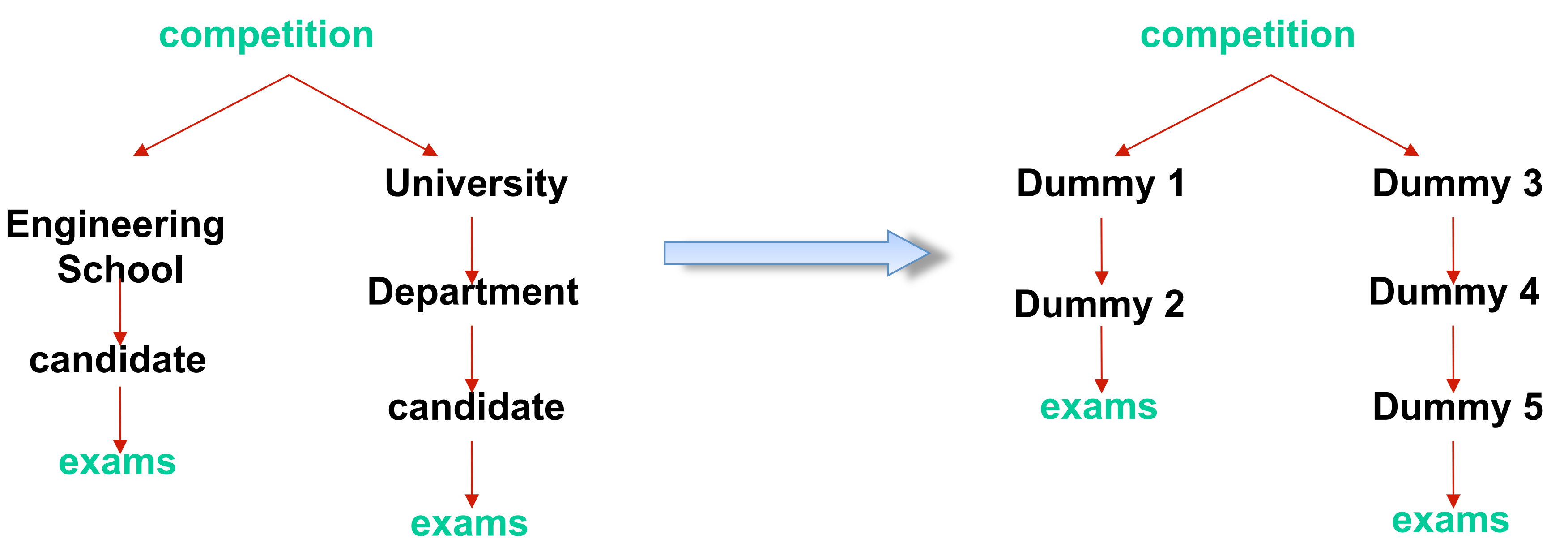


XML with Schema

Problems when using "dummy" nodes

User queries may contain "dummy" nodes

Original XML Document | **User view**

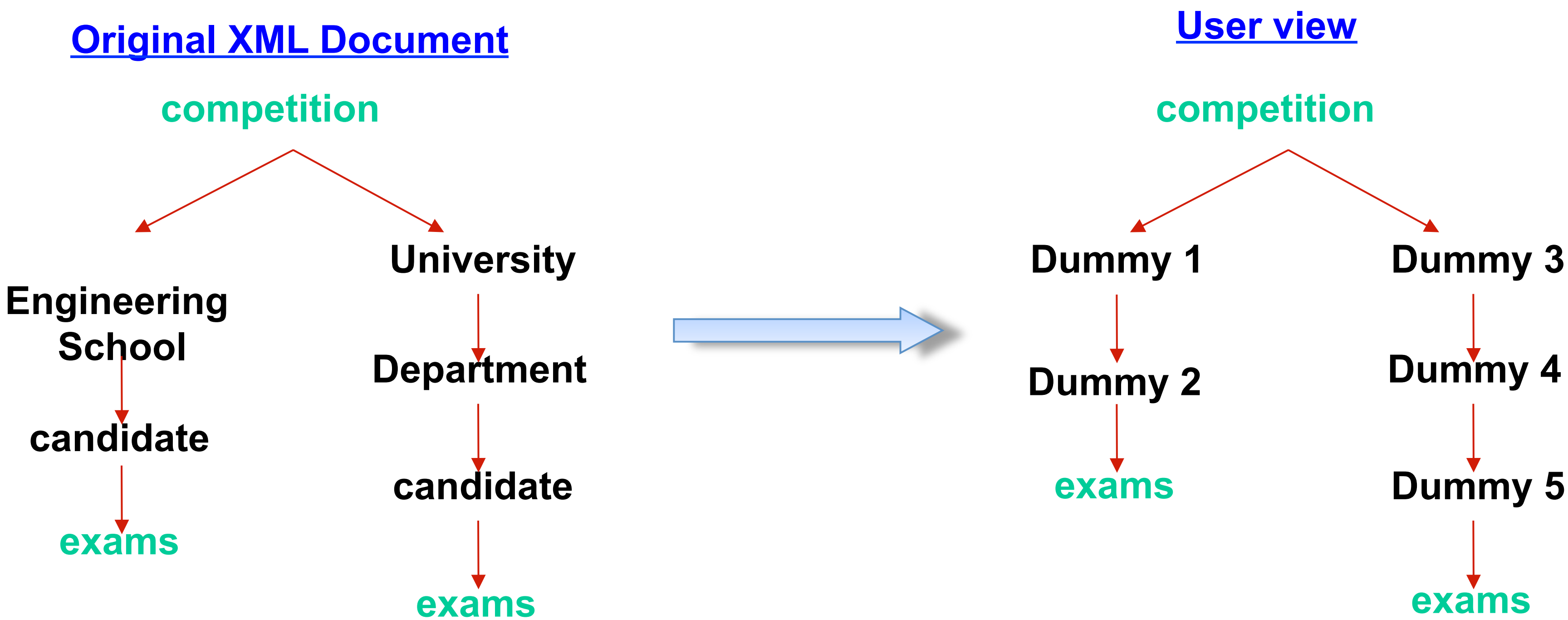


//candidate/exams ← **pre-processing** — //dummy2/exams

XML with Schema

Problems when using “dummy” nodes

User queries may disclose some confidential information



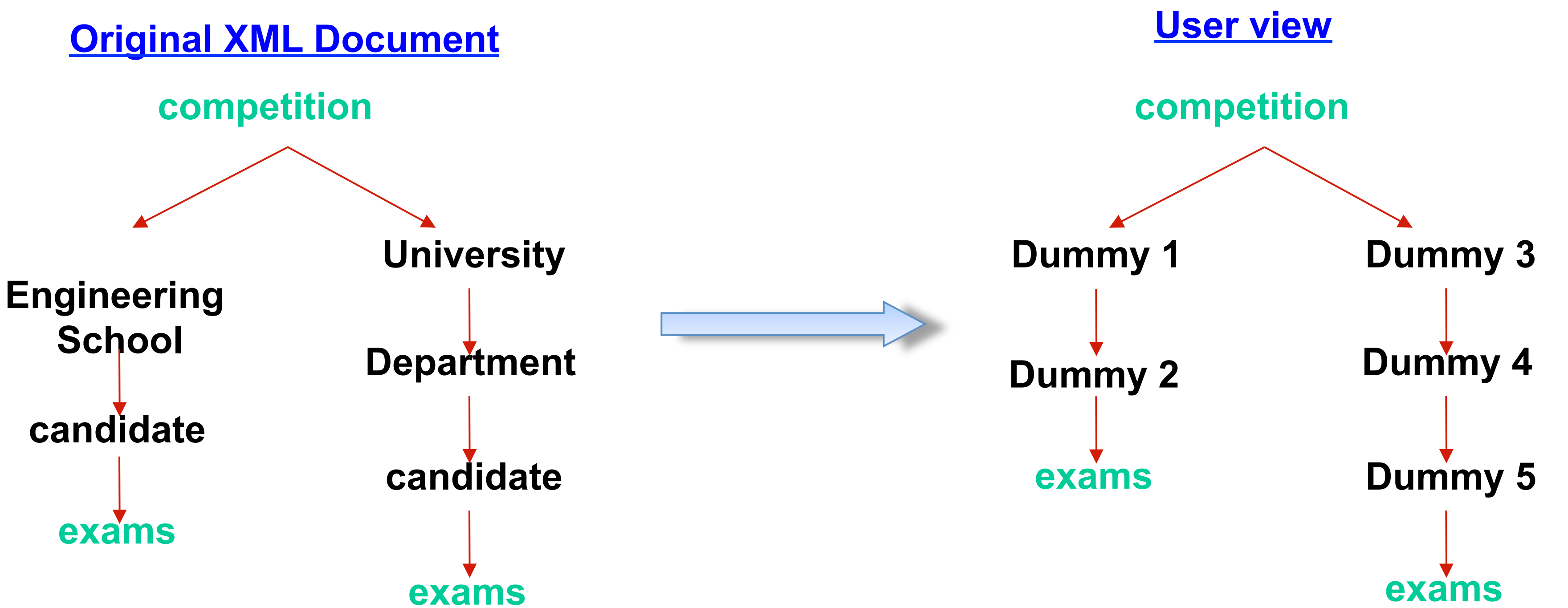
//*={university, department,...} → {dummy3, dummy4,...}

post-processing

XML with Schema

Problems when using "dummy" nodes

User queries do not contain "dummy" nodes …



… Difficult to express some queries (e.g. exams under Dummy 2)

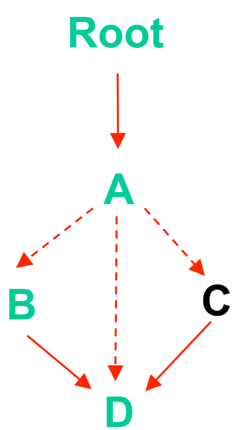
XML with Schema

Problem: the XPath fragment is not closed un rewriting

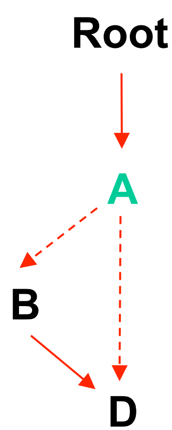
XPath fragment:

p ::= ε | A | $*$ | $//$ | p/p | $p \cup p$ | $p[q]$

q ::= p | p = "c" | $q1 \wedge q2$ | $q1 \vee q2$ | $\neg q$



DTD D1



View Dv

Qv = //D over Dv

xpath(Root,A) = A

xpath(A,D) = D $\cup$ B/D $\cup$ C/D

Qv is rewritten into

Q = /Root/xpath(Root,A)/(D $\cup$ B/D)

over D1

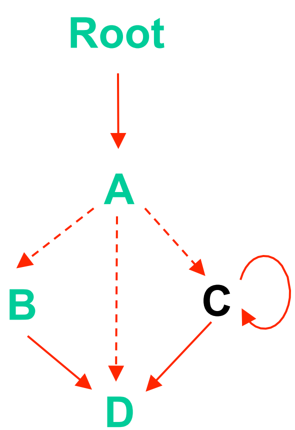
XML with Schema

Problem: the XPath fragment is not closed un rewriting

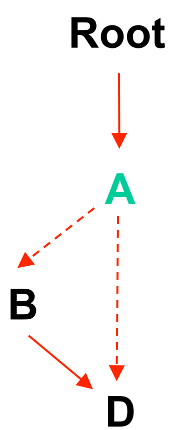
XPath fragment:

p ::= ε | A | $*$ | $//$ | p/p | $p \cup p$ | $p[q]$

q ::= p | p = "c" | $q1 \wedge q2$ | $q1 \vee q2$ | $\neg q$



DTD D2



View Dv

Qv = //D over Dv

xpath(Root,A) = A

xpath(A,D) = D ∪ B/D ∪ C/D ∪ C/C/D … ∪ C/C/C/C/D …

Qv cannot be rewritten as xpath(A,D) leads to infinitely many paths

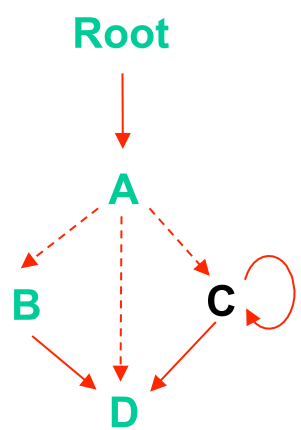
XML with Schema

Problem: the XPath fragment is not closed un rewriting

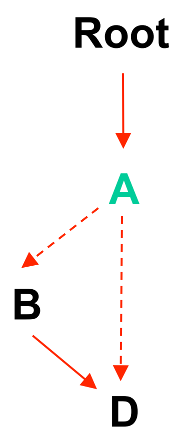
XPath fragment:

p ::= ε | A | $*$ | $//$ | p/p | $p \cup p$ | $p[q]$

q ::= p | p = "c" | $q1 \wedge q2$ | $q1 \vee q2$ | $\neg q$



DTD D2



View Dv

Qv = //D over Dv

xpath(Root,A) = A

xpath(A,D) = D ∪ B/D ∪ C/D ∪ C/C/D

… ∪ C/C/C/C/D …

Qv cannot be rewritten as xpath(A,D) leads to infinitely many paths

XPath does not contain the Kleene Star

XML with Schema

Solution 1: Using Regular XPath for rewriting [Fan 2007]

Capture DTD recursion and XPath recursion in a uniform framework

- ✓ Regular XPath:

  $Q ::= \varepsilon \mid A \mid Q/Q \mid Q \cup Q \mid Q^* \mid Q[q]$

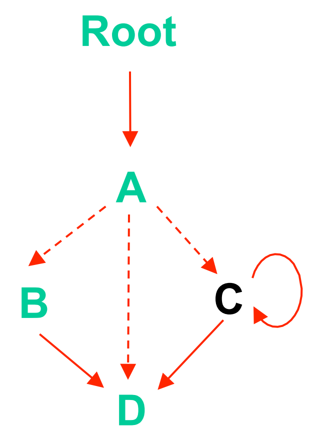
  $q ::= Q \mid Q = \text{'c'} \mid q \wedge q \mid q \vee q \mid \text{not } q$

- ✓ The child-axis, Kleene closure, union
- ✓ An XPath fragment: Q//Q instead of Q*

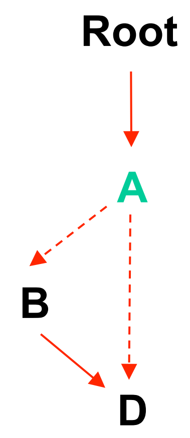
Example:

/Root/A /C//C/D is translated into

/Root/A/(C) */D



DTD D2



View Dv

XML with Schema

Solution 1: Using Regular XPath for rewriting [Fan 2007]

Drawback of Regular XPath Query

- the size of the rewritten query Q_T, if directly represented in Regular XPath, may be exponential in the size of input query Q_V.
- Regular XPath remains a theoretical achievement (it is not yet accepted as a standard)
- There are no translation and evaluation tools

XML with Schema

Solution 2: Extending the fragment for rewriting [Mah 2012]

Using two XPath fragments in a uniform framework

- ✓ XPath fragment **F** for expressing queries:

  $p ::= \varepsilon \mid A \mid * \mid // \mid p/p \mid p \cup p \mid p[q]$

  $q ::= p \mid p = \text{“c”} \mid q1 \wedge q2 \mid q1 \vee q2 \mid \neg q$

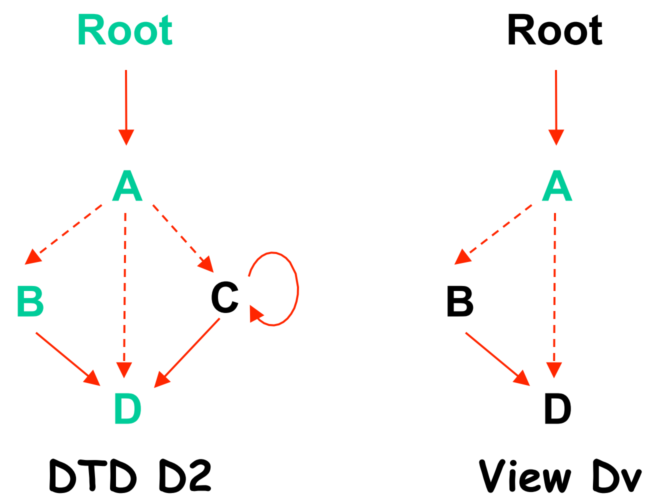
- ✓ Extended XPath fragment for rewriting queries:

  F + ../Q | ancestor ::Q | ancestor-or-self::Q | p[n]

  - ../: parent
  - ancestor, ancestor-or-self: ascendant axis
  - p[n]: Position function

XML with Schema

Solution 2: Extending the fragment for rewriting [Mah 2012]



Qv = //D over Dv such that xpath(Root,A) = A

xpath(A,D) = D ∪ B/D ∪ C/D ∪ C/C/D … ∪ C/C/C/C/D …

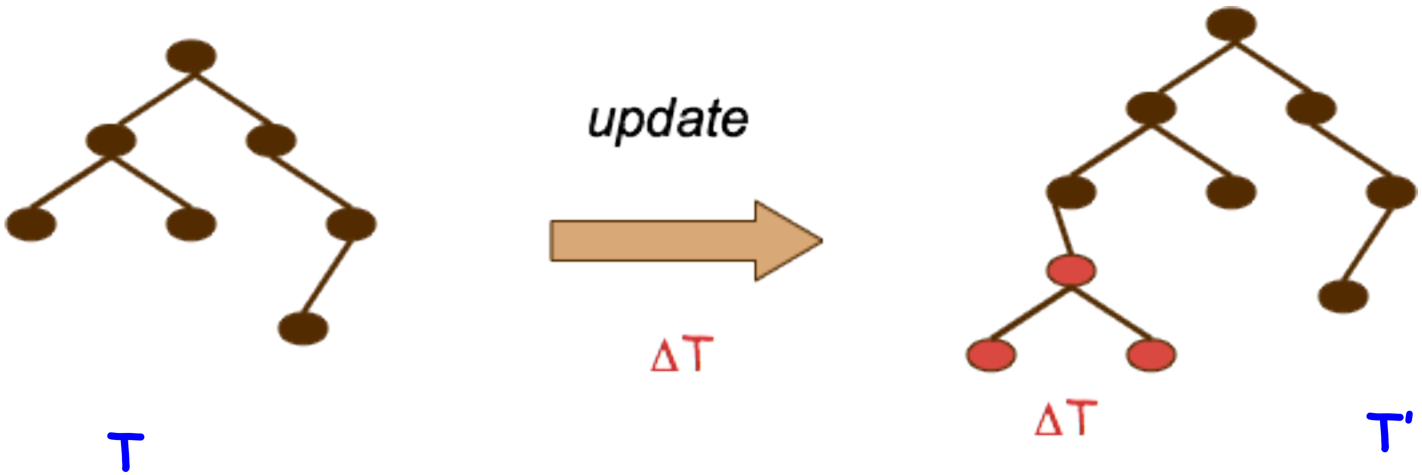
= D ∪ B/D ∪ D[ancestor::C[1]]

Qv is rewritten into //D[not ancestor::C[1]]

Outline

- Overview on XML
- Why XML Security?
- Querying Views-based XML Data
- Updating Views-based XML Data

XML Updates



Input: an XML tree T and XML update ΔT

Output: updated XML tree T' = T + ΔT

Atomic Updates

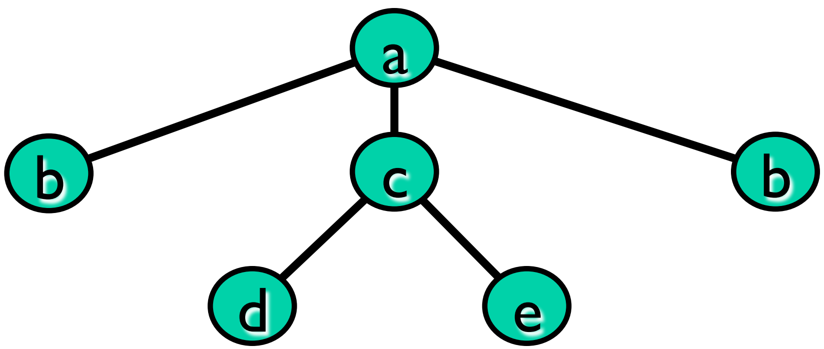
Basic changes that can be applied to tree

```
u ::= insertInto(n,t)
    |  insertAsFirstInto(n,t)
    |  insertAsLastInto(n,t)
    |  insertBefore(n,t)
    |  insertAfter(n,t)
    |  delete(n)
    |  replace(n,t)
    |  replaceValue(n,s)
    |  rename(n,a)
```

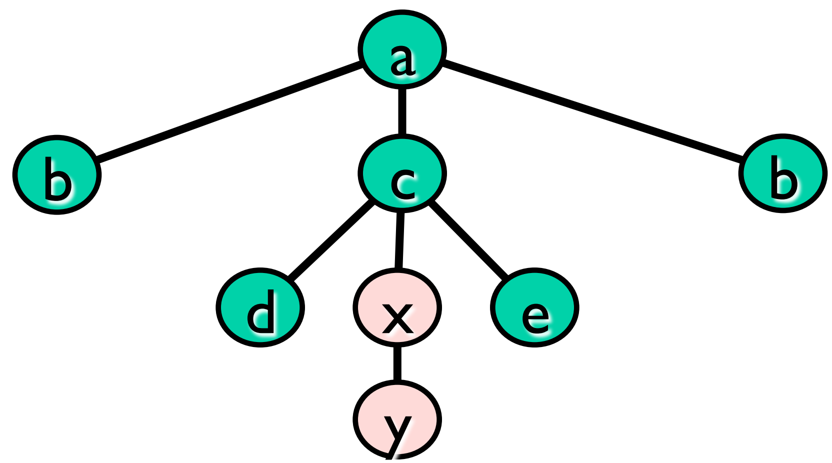
Atomic Updates

Insertion

- `InsertInto (c,<x><y/></x>)`



Before

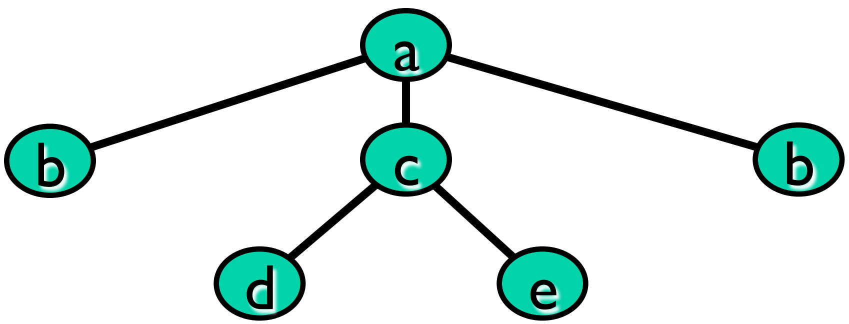


After

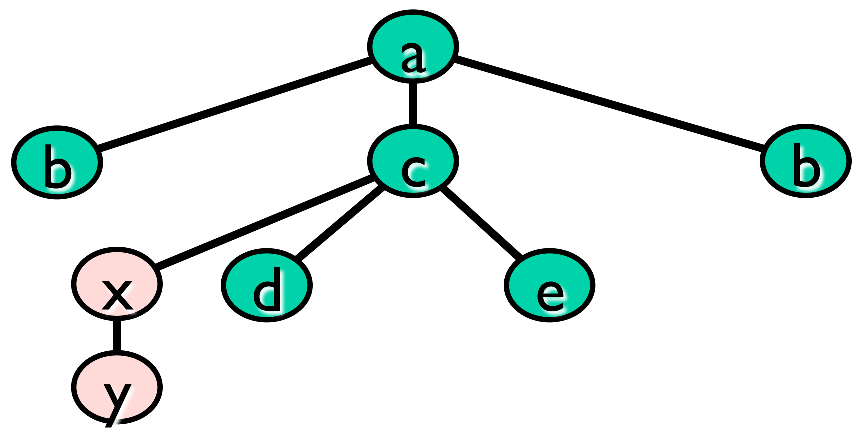
Atomic Updates

Insertion

- InsertAsFirstInto (c,<x><y/></x>)



Before

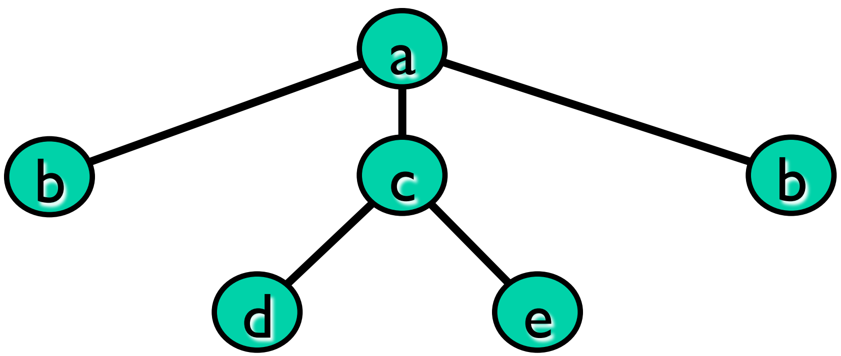


After

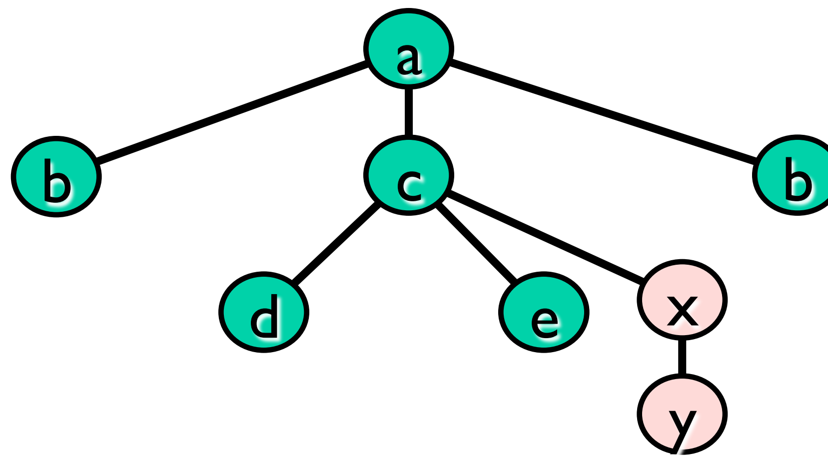
Atomic Updates

Insertion

- InsertAsLastInto (c,<x><y/></x>)



Before

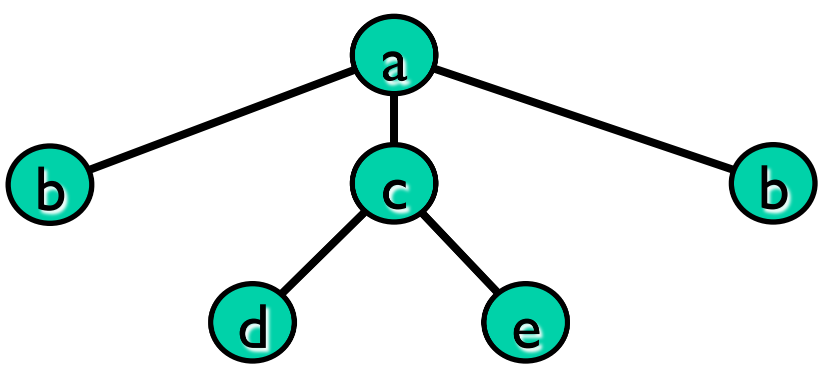


After

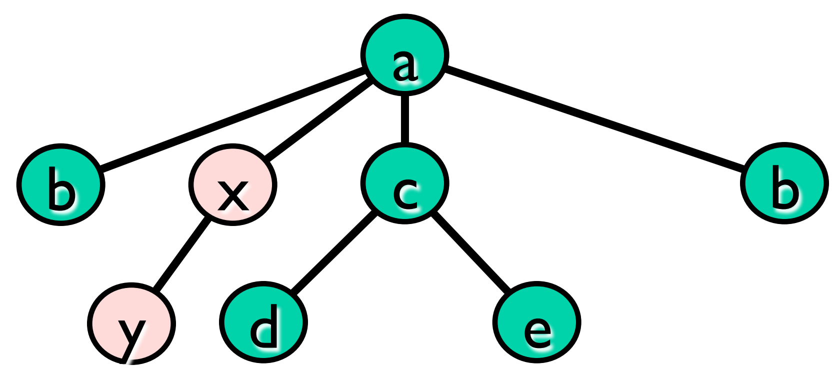
Atomic Updates

Insertion

- InsertBefore (c,<x><y/></x>)



Before

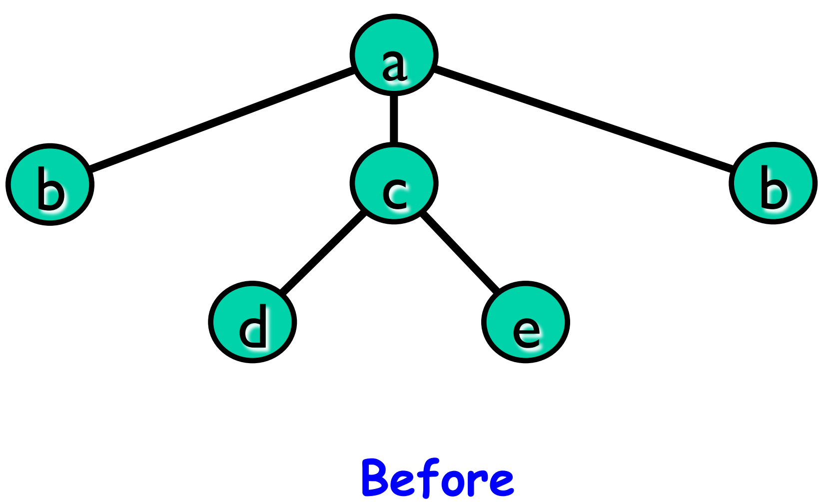


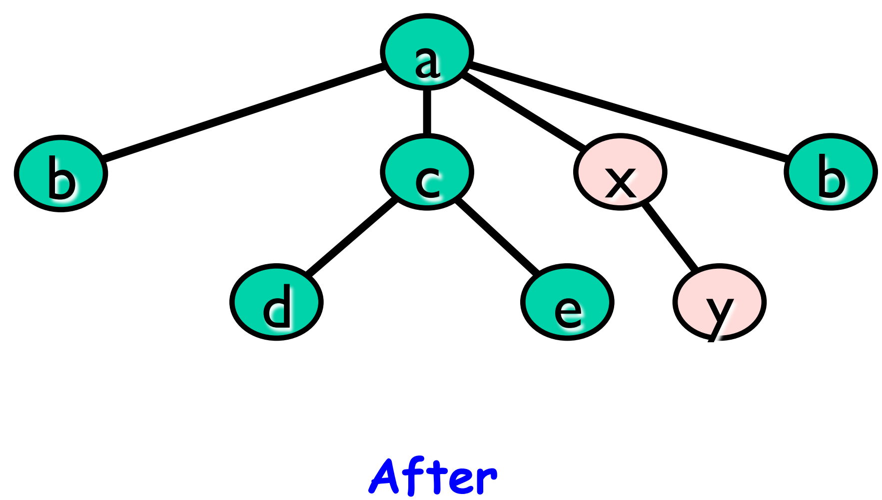
After

Atomic Updates

Insertion

- `InsertAfter (c,<x><y/></x>)`

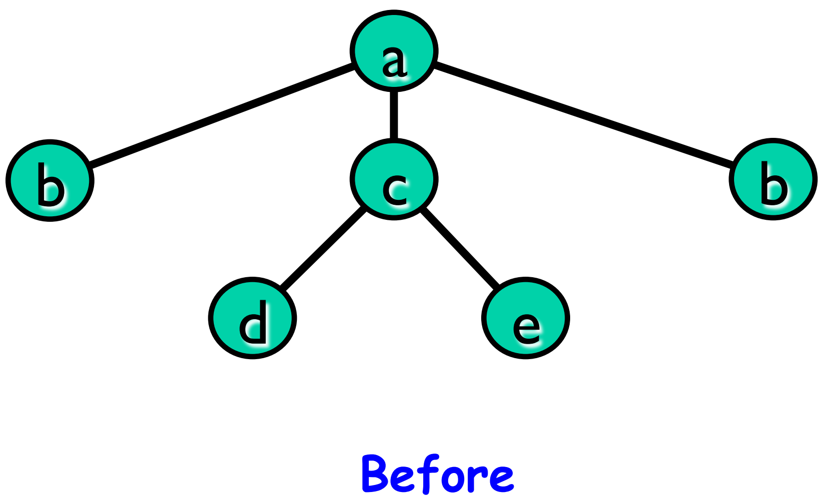


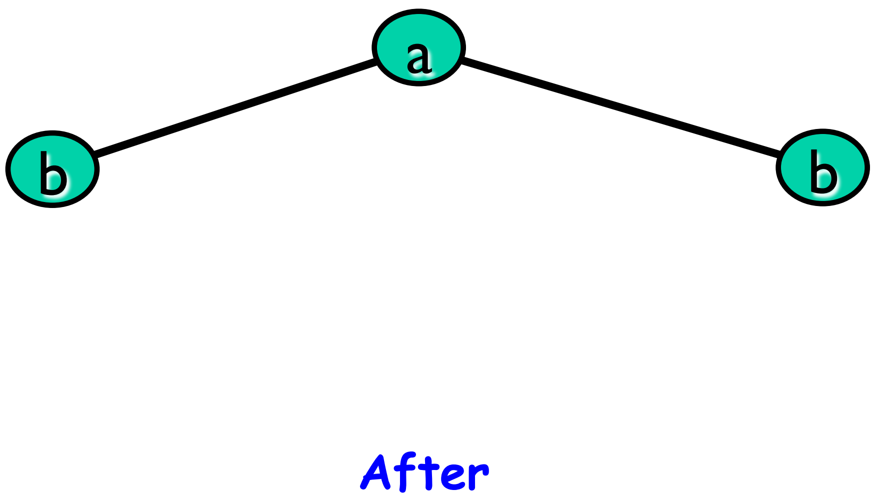


Atomic Updates

Deletion

- `Delete (c)`

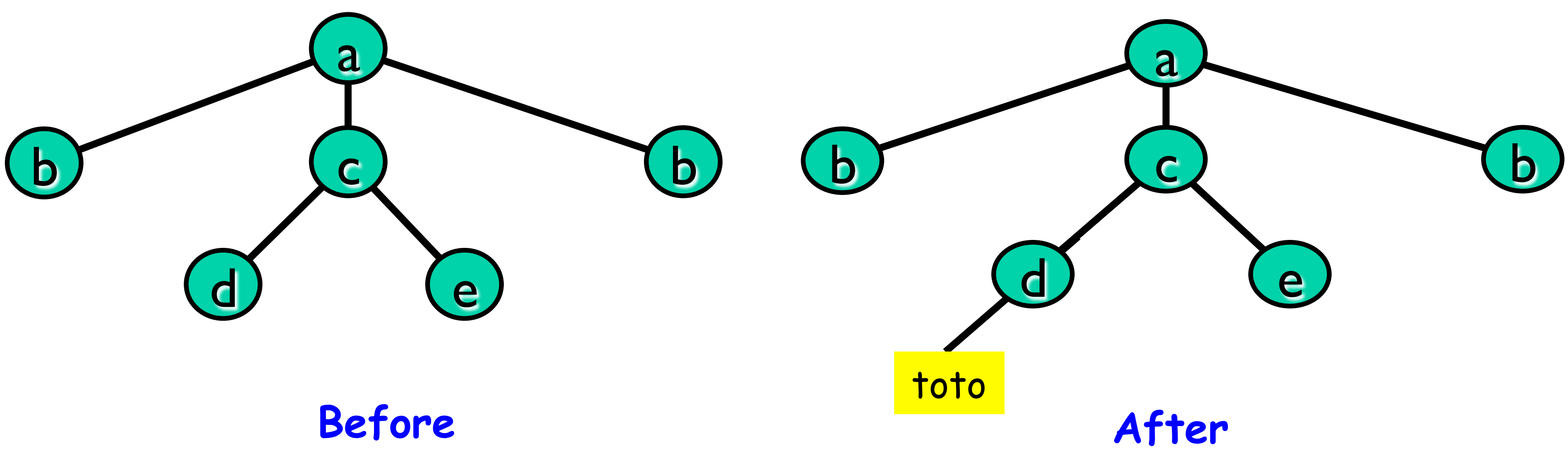




Atomic Updates

Replace Text Value

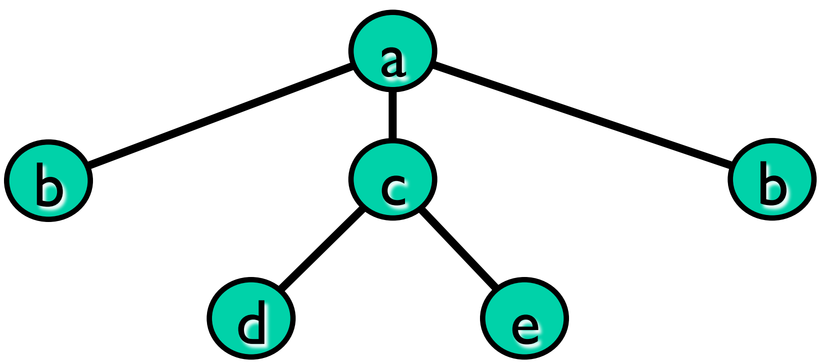
- `ReplaceValue (d, "toto")`



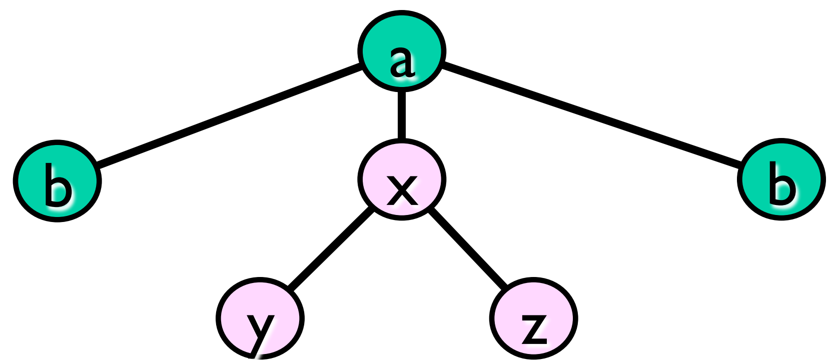
Atomic Updates

Replace Subtree

- `Replace (c, <x><y/><z/></x>)`



Before

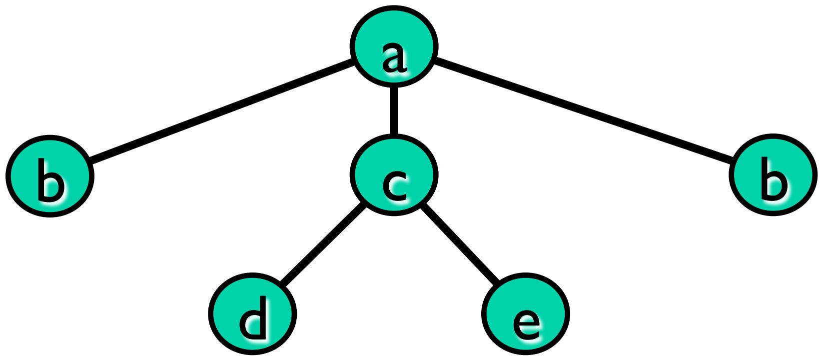


After

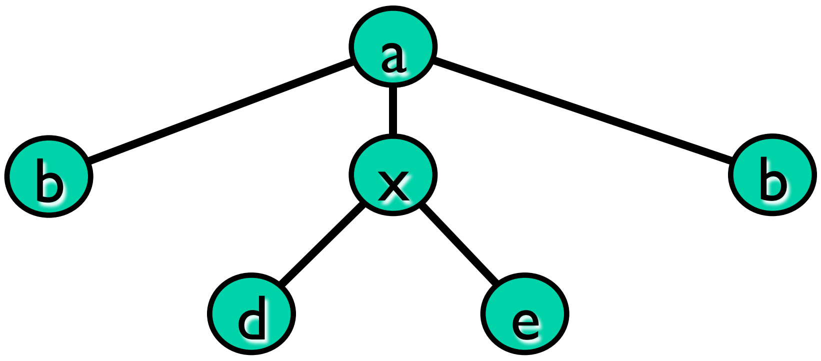
Atomic Updates

Rename

- `Rename (c, x)`



Before



After

Access Control with Updates

Existing access control approaches

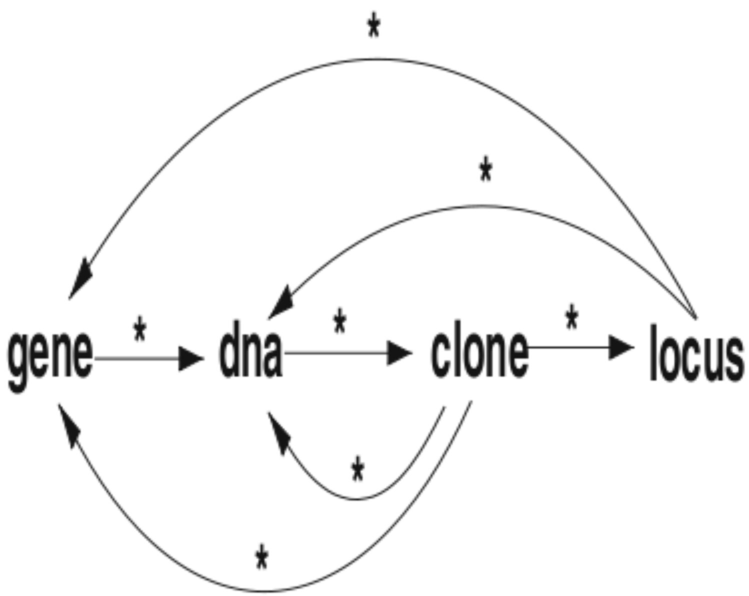
- Most of XML access control approaches deal only with *read access rights*
- Access control considering *update rights* has not received more attention
- The XQuery Update *Facility*: a recommendation of W3C providing facility to modify XML documents

Drawbacks

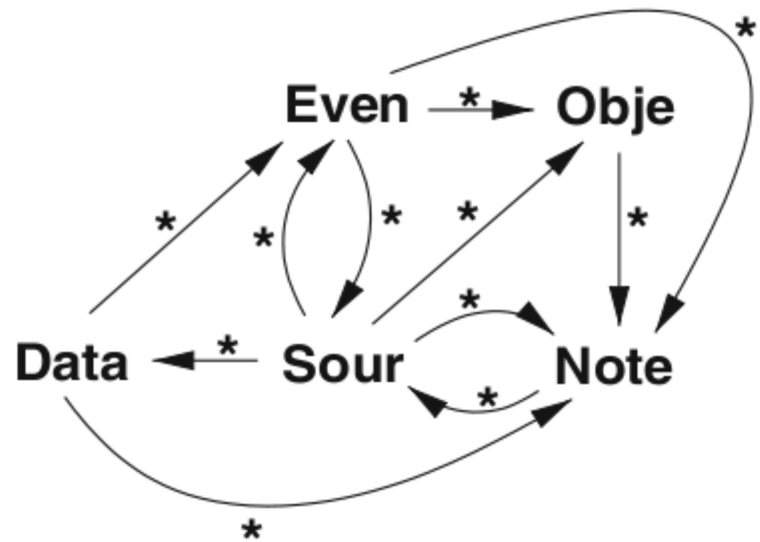
- Existing update access control languages are *unable* to specify some update policies in case of recursive DTDs
- *No practical tool exists* for securely querying and updating XML data over recursive DTDs

Access Control with Updates

Example of DTD: Biopolymer and Genealogical Data



(a) BIOML

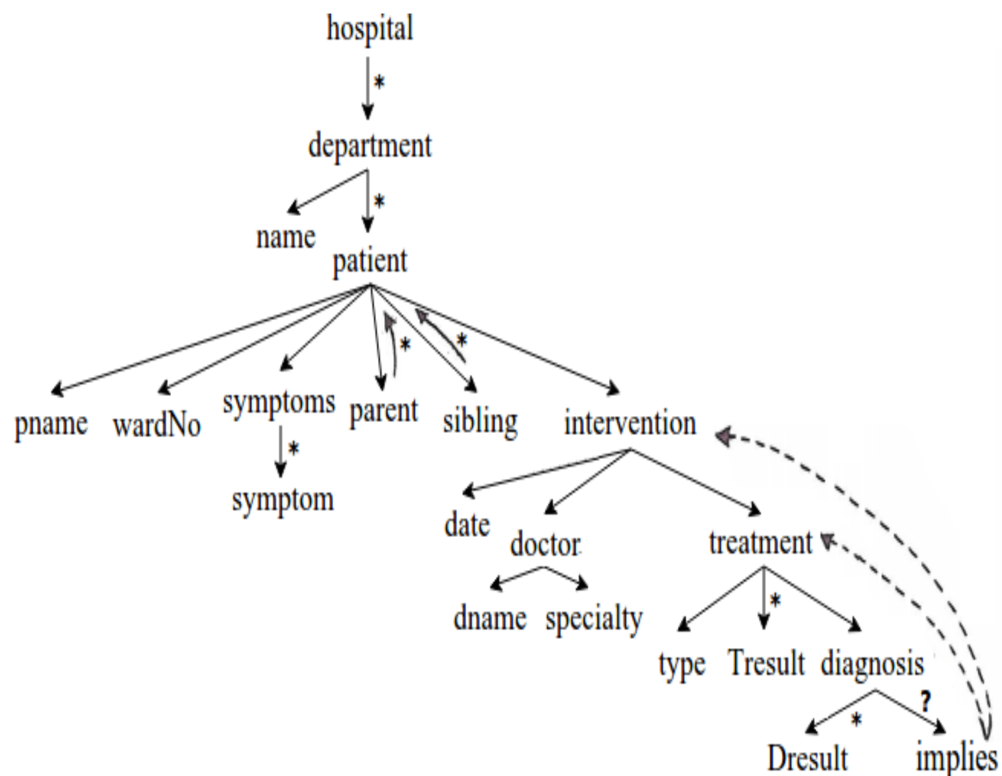


(b) GedML

Access Control with Updates

Example of DTD: Hospital Data

(a) Patient Treatment
Life-cycle Management



(b) Corresponding DTD

Basic Notions

DTD (Document Type Definition)

A DTD *D* is a triple (*Ele, Rg, root*) where:

- *Ele* is a set of element types;
- *root* is a distinguished element type, called the *root type;*
- *Rg* is a function such that for any *A* in *Ele*, *Rg(A)* is a regular expression of the form:

$$\alpha \ \texttt{:=}\ \ \texttt{str} \mid \epsilon \mid B \mid \alpha\text{'},\text{'}\alpha \mid \alpha\text{'}|\text{'}\alpha \mid \alpha* \mid \alpha+ \mid \alpha?$$

- $A \longrightarrow$ *Rg(A)* is the production of *A;*
- *B* is a *child type* of *A*, and *A* is a *parent* type of *B;*
- *D* is *recursive* if there is an element type *A* defined in terms of itself directly or indirectly.

Basic Notions

Xquery Update Operations

In the following, *source* is a set of XML nodes, and *target* is an XPath expression which returns a single node in case of *Insert* and *Replace* operations.

- *Insert source into target*: insert nodes in source as children of target's node.
- *Insert source as first/as last into target*: insert nodes in source as first (resp. as last) children of target's node.
- *Insert source before/after target*: insert nodes in source as preceding (resp. following) sibling nodes of target's node.
- *Replace target with source*: replace target's node with the nodes in source.
- *Replace value of target with string-value*: replace the text-content of target's node with the new value *string-value*.
- *Delete target*: delete nodes returned by target along with their descendant nodes.
- *Rename target with string-value*: rename the label of target's node with the new label *string-value*.

Access Control Policy

Goals

For each user group of an XML document T:

- Specifying an update-access policy S
- Enforcing S at update time: any update *op* must be performed only at nodes that are updatable w.r.t. S.

Challenges

- How to specify update policies at various granularity levels?
- How to specify update policies over arbitrary DTDs?
- How to efficiently enforce those update policies ?

Access Control Policy

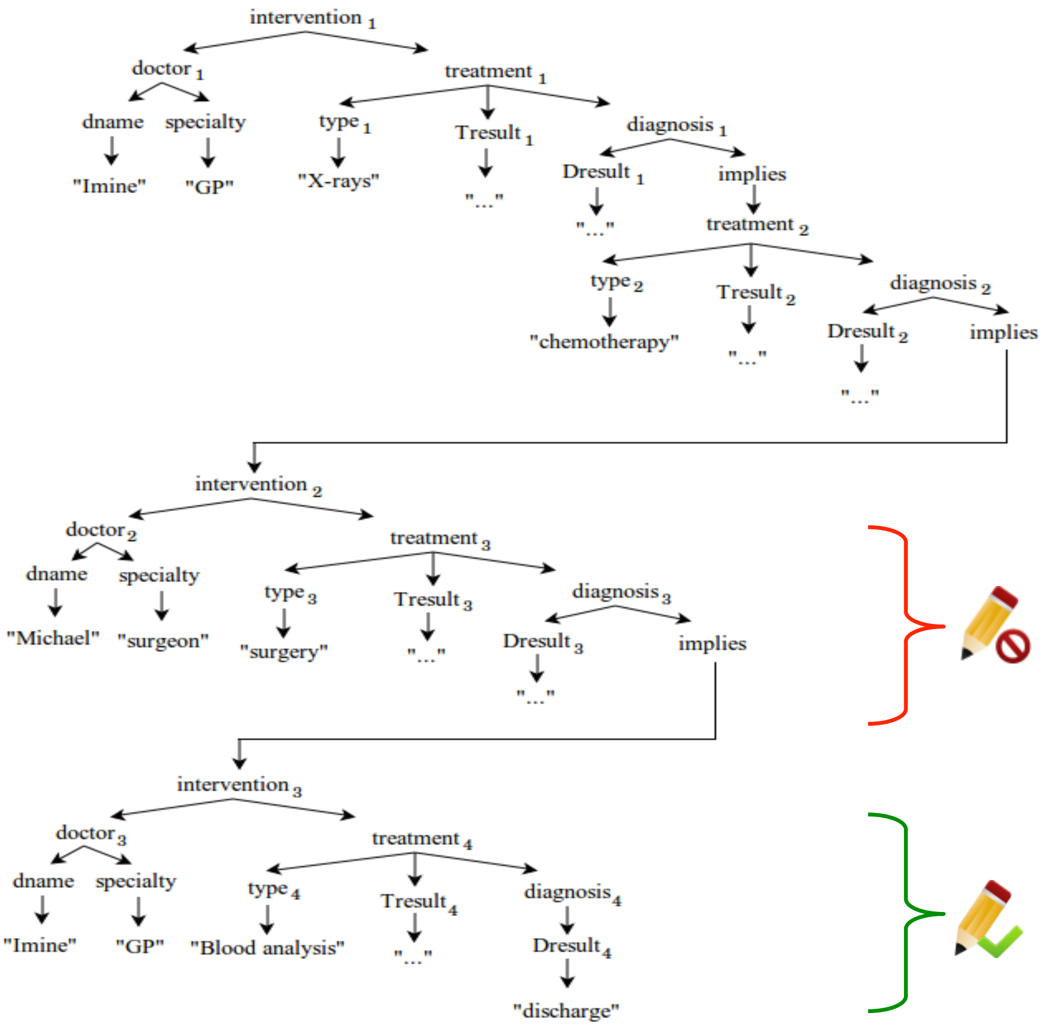
Example: Doctor Update Policy

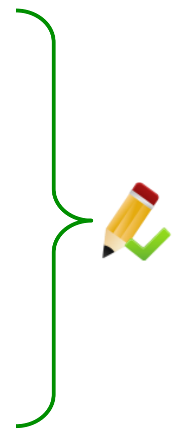
Update Policy:

Each doctor can update only data of treatments that she/he has done.

Access Control Policy

Example: Update rights of Dr Imine





User update:

Delete //treatment [type='surgery']/Tresult

ERROR: insufficient privilege

Existing Access Control Models

Model of Fundulaki et *al.* [Fun2007]

- An XPath-based rules language (**XACU**)is proposed to specify update policies.
- An *XACU* rule has the form: (*object, action, effect*).
- An XACU rule can be *positive/negative*, *local/recursive*.
- *Grant/Deny* overrides as conflict resolution policy.

Drawbacks

- The **XACU** language can be used only for non-recursive DTDs.

Existing Access Control Models

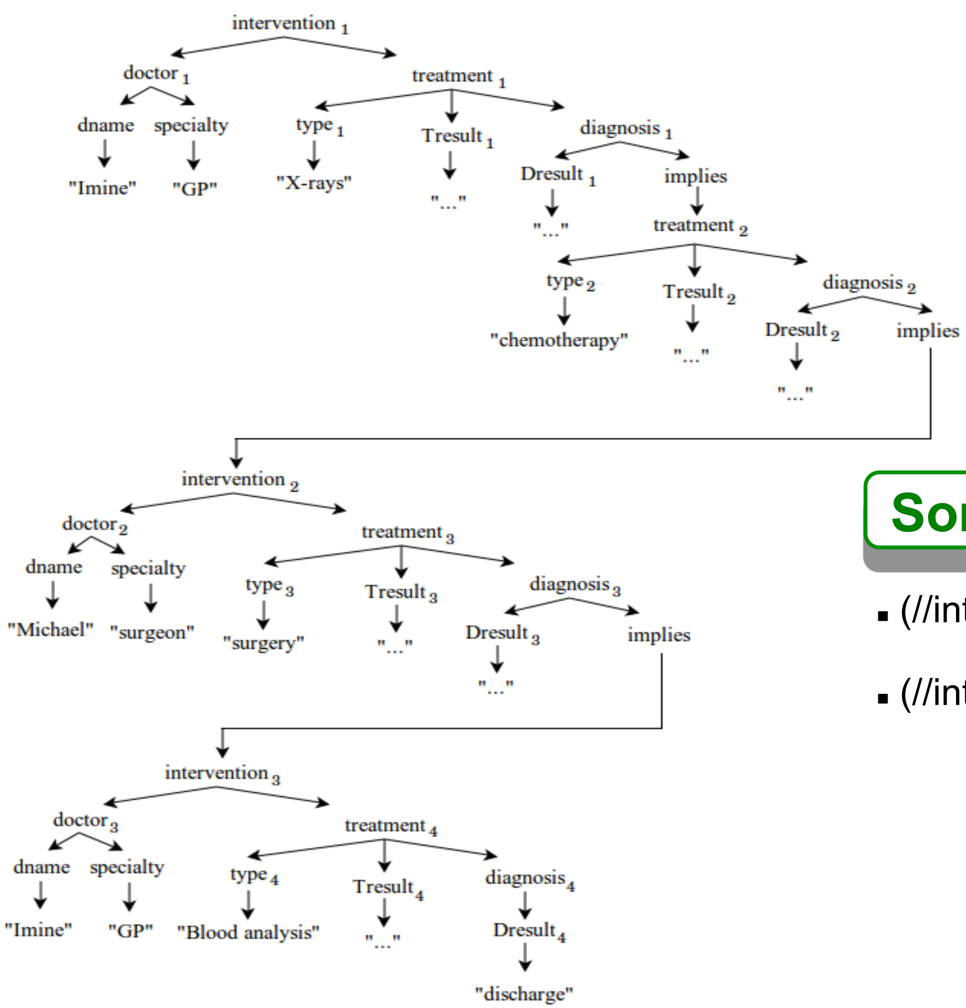
Model of Fundulaki et *al.* [Fun2007]

Update Policy:

Each doctor can update only data of treatments that she/he has done.

Existing Access Control Models

Model of Fundulaki et *al.* [Fun2007]



Update Policy:

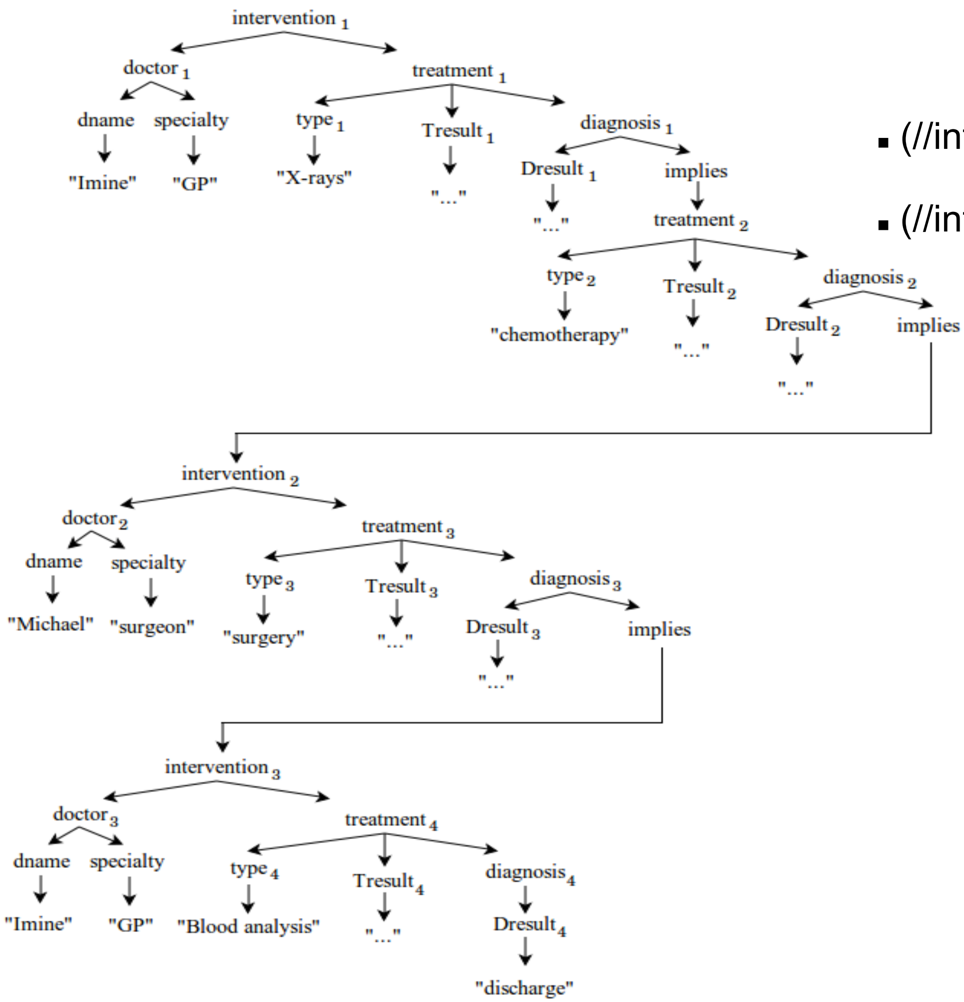
Each doctor can update only data of treatments that she/he has done.

Some XACU rules:

- (//intervention[doctor/dname='Imine']//*treatment*, delete, +)
- (//intervention[doctor/dname≠'Imine']//*treatment*, delete, -)

Existing Access Control Models

Model of Fundulaki et *al.* [Fun2007]



Some XACU rules:

- (//intervention[doctor/dname='Imine']//*treatment*, delete, +)
- (//intervention[doctor/dname≠'Imine']//*treatment*, delete, -)

Limitation:

Nodes treatment$_3$ and treatment$_4$ are in the scopes of both the two XACU rules.

Grant overrides: node treatment$_3$ becomes updatable for Imine.

Deny overrides: node treatment$_4$ becomes not updatable for Imine.

Existing Access Control Models

Model of Damiani et *al.* [Dam2008]

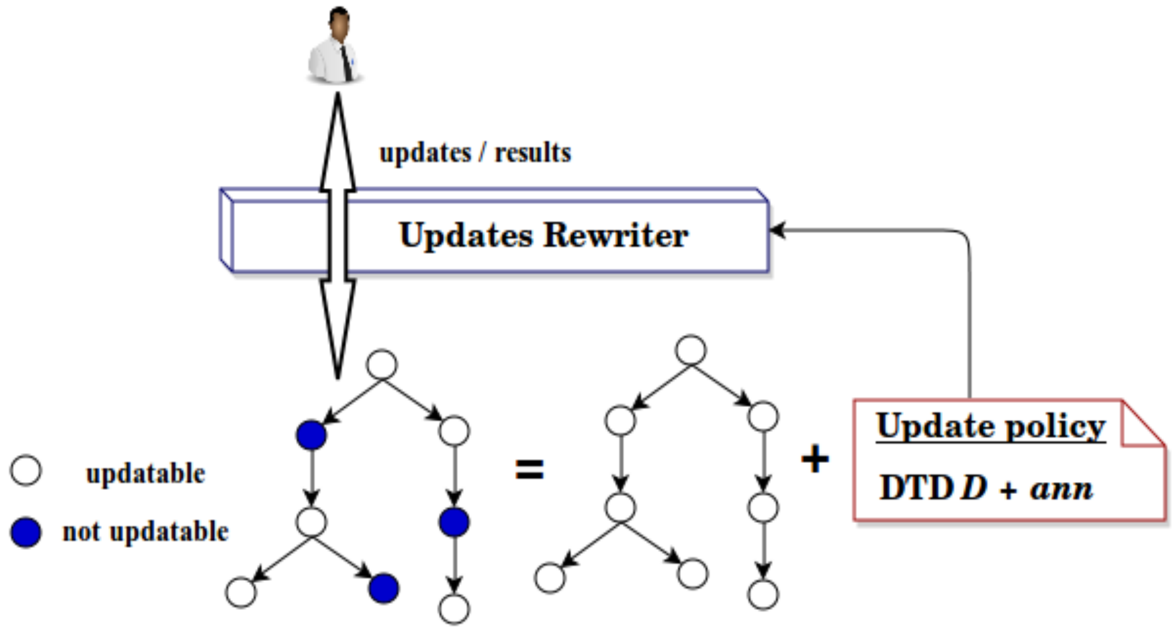
- Update policies are defined by *annotating* element types of the DTD by security attributes.
- E.g., attribute @insert=Y on element type treatment specifies that some nodes can be inserted as children of treatment nodes.
- Update policy is translated into security automaton.
- Each update operation is rewritten into a safe one by parsing this automaton.

Drawbacks

- Query rewriting over automaton is guaranteed only when DTDs are non-recursive
- Update annotations are local which is *insufficient* to specify some update constraints.

Existing Access Control Models

Model of Mahfoud et *al.* [Mah2012]



Security Administrator:

Specifies for each group of users an update policy by annotating the DTD with update constraints (i.e. XPath qualifiers).

Updates Rewriter Module:

Translates each update operation into a *safe* one in order to be performed only over nodes that can be updated w.r.t. the update policy.

Existing Access Control Models

Model of Mahfoud et *al.* [Mah2012]

Update Specification: *Update policy = DTD + XPath Qualifiers*

An update specification *S=(D, Annot)*: *Annot* is a mapping from element types of *D* into: *Y*, *N*, [*Q*].
For an element type *A* in *D*, and an update of type *op*, define *Annot(A, op)* as:

- *Y* : operation of type *op* can be performed at nodes of type *A*.
- *N* : operation of type *op* cannot be performed at nodes of type *A*.
- [*Q*] : operation of type *op* can be performed at node of type *A* iff [*Q*] is valid.

Update types:

We define restricted update operations that can be performed only for some specific element types. E.g. *insertInto*[*B*], *delete*[*B*], *replaceNode*[*Bi,Bj*].

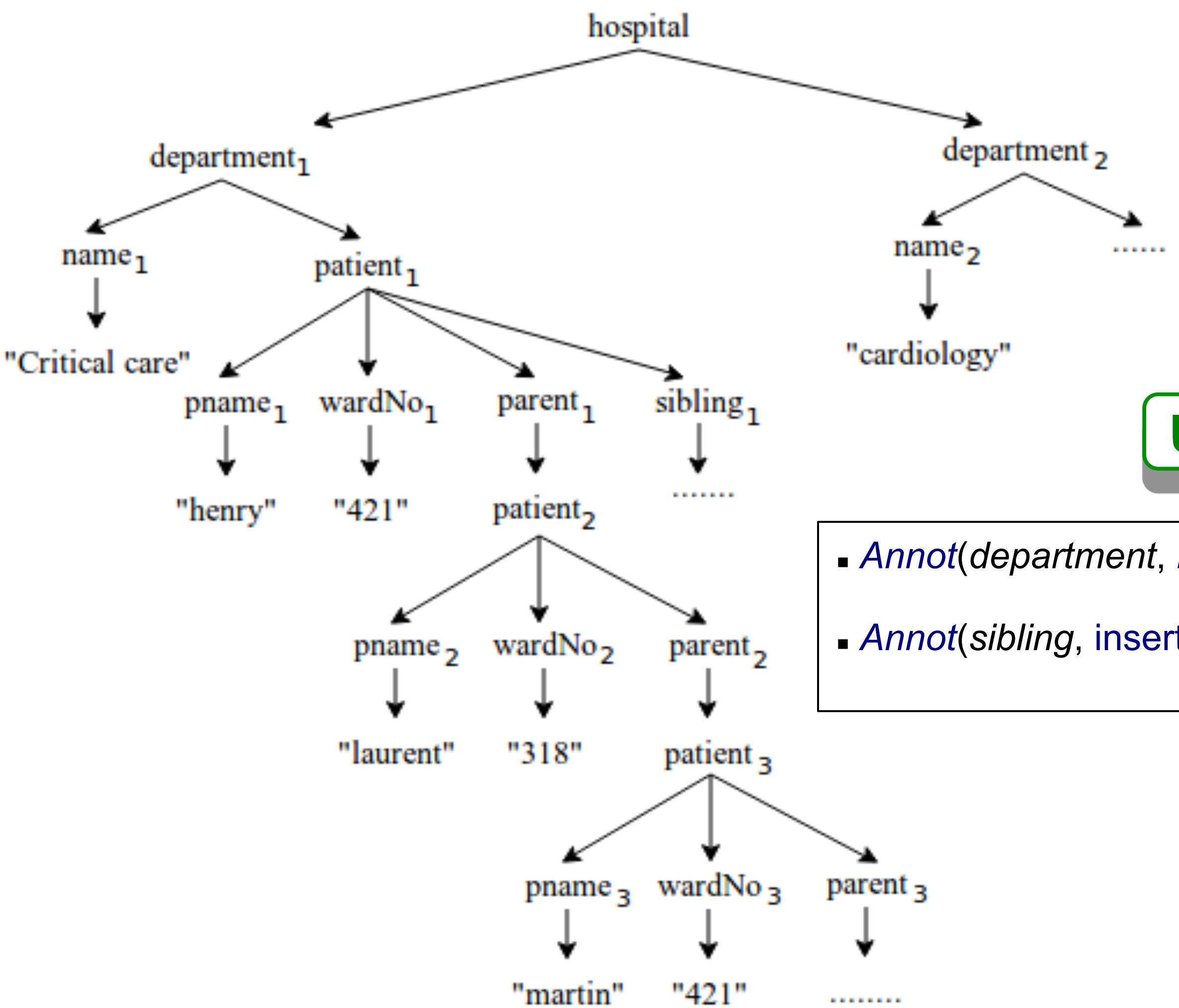
Local and recursive rules:

Inheritance and overriding of update rights

Existing Access Control Models

Model of Mahfoud et *al.* [Mah2012]

Example: Update Policy for Nurses

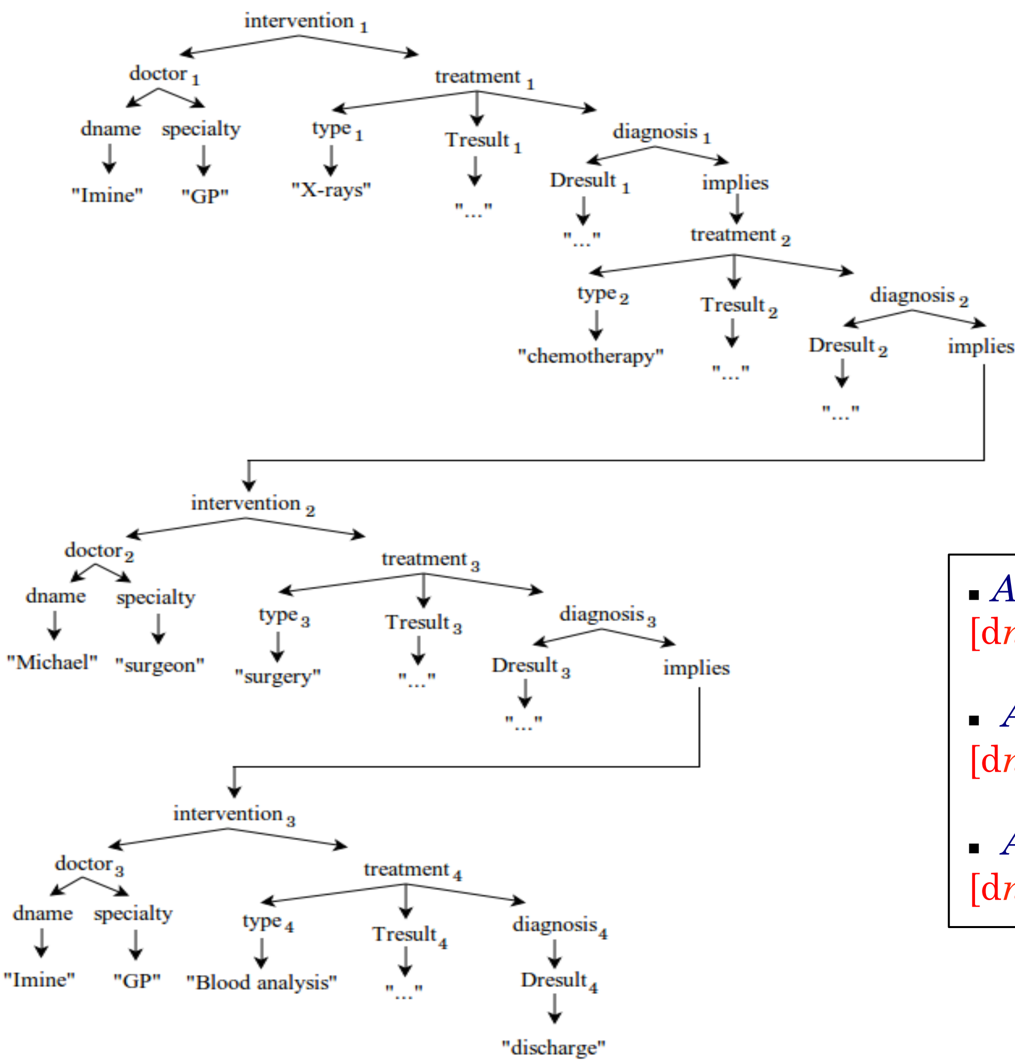


Update specification:

- *Annot*(*department*, *insertInto*[*patient*]) = [*name*='Critical care']
- *Annot*(*sibling*, insertInto[*patient*]) = N

Existing Access Control Models

Model of Mahfoud et *al.* [Mah2012]



Example: Update Policy for Dr Imine

Update Policy:

Each doctor can update only data of treatments that she/he has done.

Update specification:

- $Annot(intervention, replaceValue[Tresult]) =$ $[dname='Imine']$
- $Annot(intervention, insertAfter[\text{type}, Tresult]) =$ $[dname='Imine']$
- $Annot(intervention, delete[Tresult]) =$ $[dname='Imine']$

Existing Access Control Models

Model of Mahfoud et *al.* [Mah2012]

Rewriting principle:

Given an update specification *S*=(*D*, *Annot*) and an update operation *op* over an instance *T* of *D*. We rewrite *op* into a safe one op^t such that executing op^t over *T* has to modify only nodes that are updatable w.r.t. *S*.

Rewriting Problem:

Consider the XPath fragment $\mathcal{X}$ defined as follows:

$$
\begin{array}{lcl}
p & := & \alpha::lab \mid p[q] \mid p/p \mid p \cup p \\
q & := & p \mid p/text()='c' \mid q \ and \ q \mid q \ or \ q \mid not \ (q) \\
\alpha & := & \varepsilon \mid \downarrow \mid \downarrow^{+} \mid \downarrow^{*}
\end{array}
$$

For recursive DTDs, the fragment $\mathcal{X}$ is **not closed** under update operations rewriting.

Existing Access Control Models

Model of Mahfoud et *al.* [Mah2012]

Example: Update Policy for Dr Imine

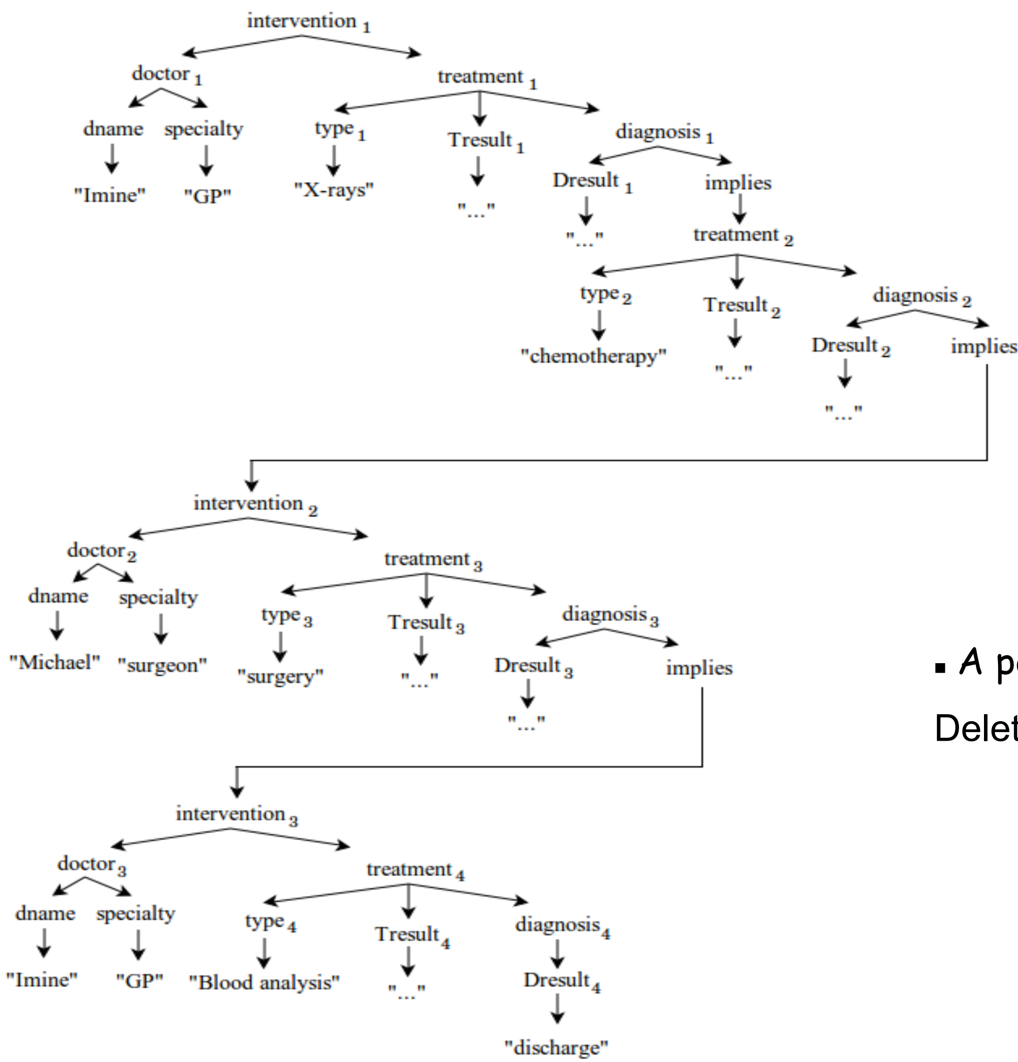
- $Annot(intervention, delete[Tresult])$ = [$dname$='Imine']

User update:

- Delete //Tresult cannot be rewritten in $\mathcal{X}$

Existing Access Control Models

Model of Mahfoud et *al.* [Mah2012]



Example: Update Policy for Dr Imine

- $Annot(intervention,\ delete[Tresult]) = [dname='Imine']$

User update:

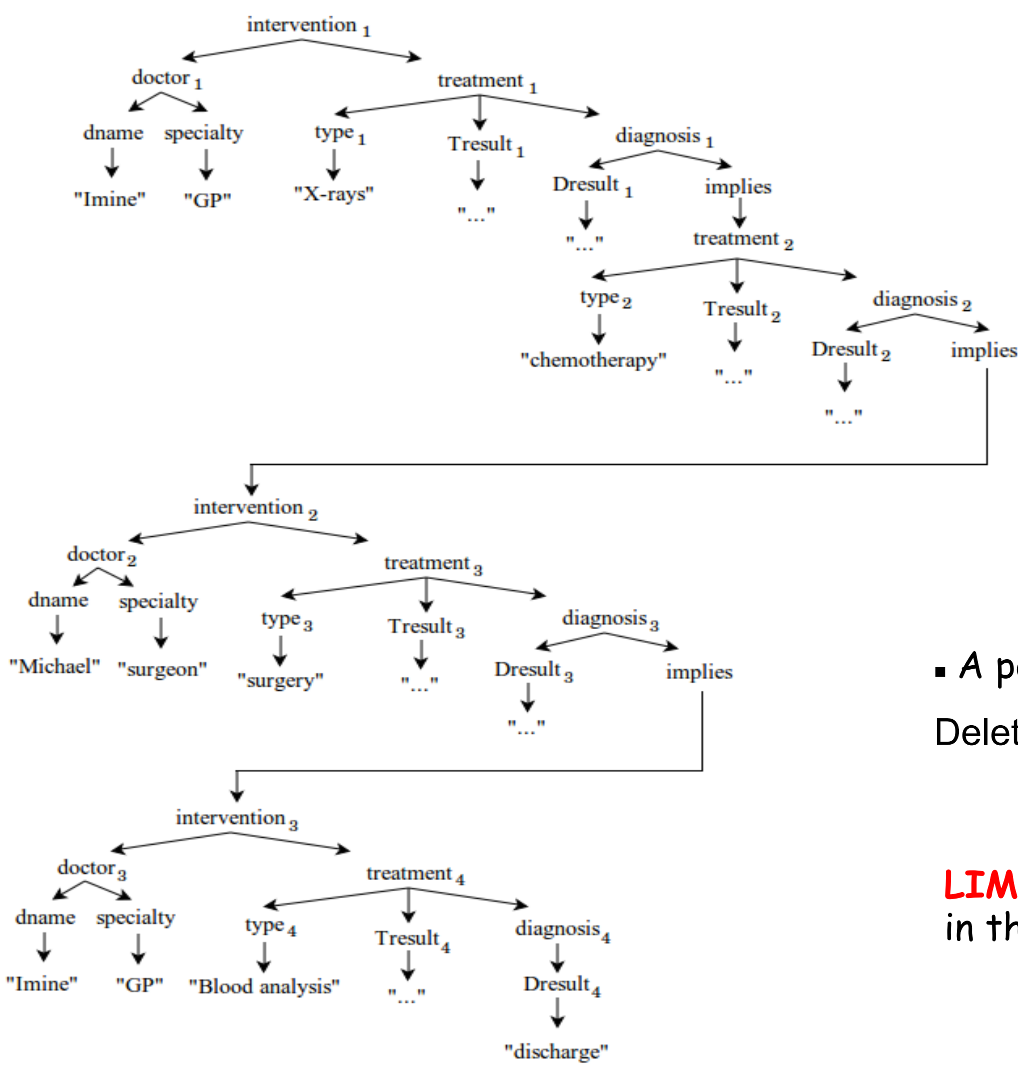
- Delete //Tresult cannot be rewritten in $\mathcal{X}$

- A possible rewriting:

Delete //intervention[doctor/dname='Imine']/treatment/(implies/diagnosis/treatment)*/Tresult

Existing Access Control Models

Model of Mahfoud et *al.* [Mah2012]



Example: Update Policy for Dr Imine

- $Annot(intervention, delete[Tresult]) = [dname='Imine']$

User update:

- Delete //Tresult cannot be rewritten in $\mathcal{X}$

- A possible rewriting:

Delete //intervention[doctor/dname='Imine']/treatment/(implies/diagnosis/treatment)*/Tresult

LIMIT. The kleene star (*) cannot be expressed in the standard XPath.

Existing Access Control Models

Model of Mahfoud et *al.* [Mah2012]

Solution:

We extend fragment $\mathcal{X}$ as follows:

$$\begin{array}{lcl} p & := & \alpha::lab \mid p[q] \mid p/p \mid p \cup p \mid p[n] \\ q & := & p \mid p/text()='c' \mid q \; and \; q \mid q \; or \; q \mid not \; (q) \\ \alpha & := & \varepsilon \mid \downarrow \mid \downarrow^{+} \mid \downarrow^{*} \mid \uparrow \mid \uparrow^{+} \mid \uparrow^{*} \end{array}$$

We extend $\mathcal{X}$ into $\mathcal{X}_{[n]}^{\Uparrow}$ by adding upward axes (*parent*, *ancestor*, and *ancestor-or-self*), and the *position predicate* (i.e., [n]).

For recursive DTDs, the fragment $\mathcal{X}_{[n]}^{\Uparrow}$ is **closed** under update operations rewriting.

Existing Access Control Models

Model of Mahfoud et *al.* [Mah2012]

Update Rewriting Algorithm

- Input:

  An update specification *S*=(*D*, *Annot*) and an update operation *op* defined in $\mathcal{X}$.

- Output:

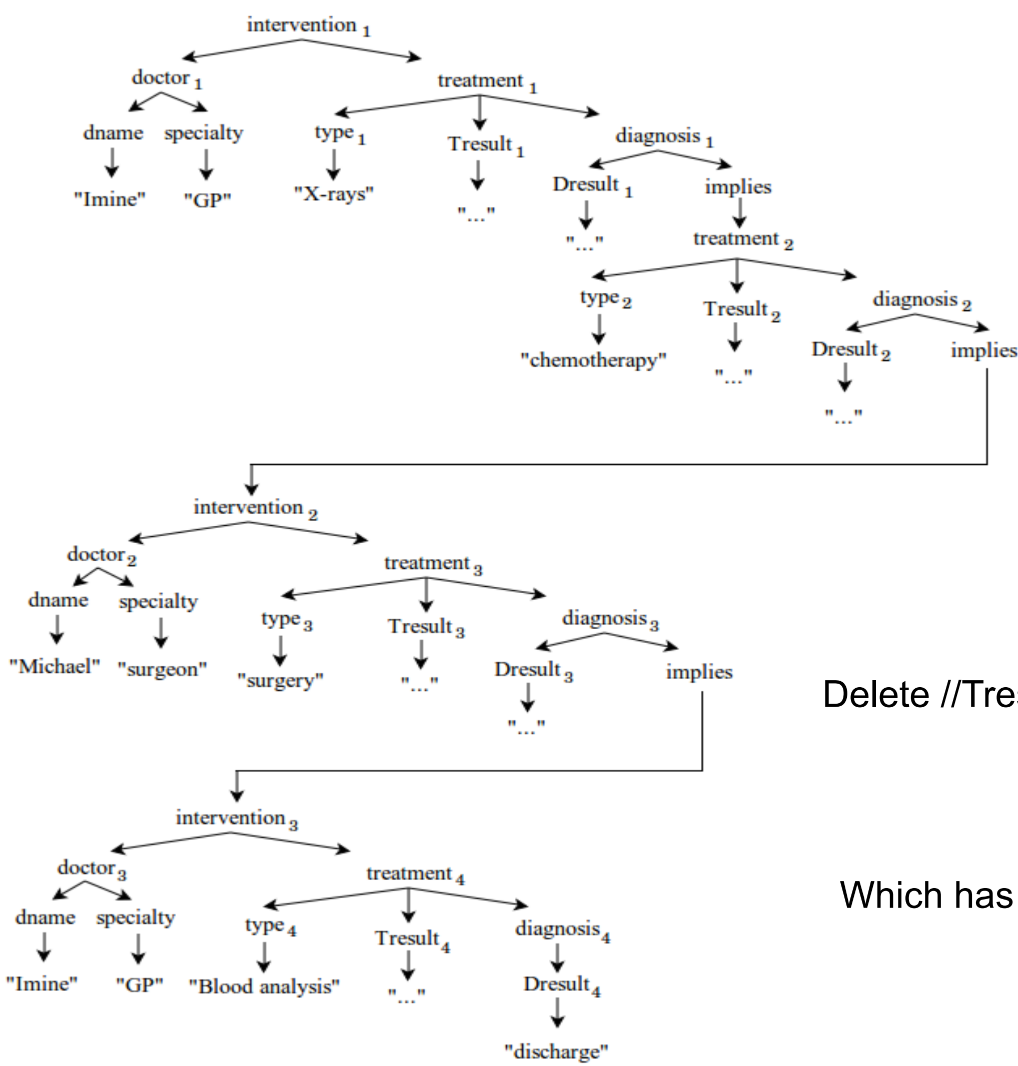
  A safe update op^t defined in $\mathcal{X}_{[n]}^{\Uparrow}$ such that executing op^t over any instance *T* of *D* has to modify only nodes that are updatable w.r.t. *S*.

- Efficiency:

  For any update specification *S*=(*D*, *Annot*) and any update operation *op*, rewriting of *op* can be done in $O(|Annot|)$ time.

Existing Access Control Models

Model of Mahfoud et *al.* [Mah2012]



Example: Update Policy for Dr Imine

- $Annot(intervention, delete[Tresult]) = [dname='Imine']$

User update:

- Delete //Tresult can be rewritten in $\mathcal{X}^{\Uparrow}_{[n]}$

Delete //Tresult[ancestor::intervention[1] [doctor/dname='Imine']]

Which has to delete nodes $Tresult_1$, $Tresult_2$ and $Tresult_3$.

Some References

• Irini Fundulaki, Maarten Marx. Specifying access control policies for XML documents with XPath. In: *SACMAT 2004*.

• Irini Fundulaki, Sebastian Maneth. Formalizing XML Access Control for Update Operations. In: *SACMAT 2007*.

• Wenfei Fan et *al*. Secure XML Querying with Security Views. In: *SIGMOD 2004*.

• Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Rewriting Regular XPath Queries on XML Views. In: *ICDE 2007*.

• Ernesto Damiani, Majirus Fansi, Alban Gabillon, Stefania Marrara. A General Approach to Securely Querying XML. In: *Computer Standards and Interface 2008*.

• Mahfoud Houari, and Abdessamad Imine. Secure querying of recursive XML views: a standard xpath-based technique (short paper).In: *WWW 2012*.

• Mahfoud Houari, and Abdessamad Imine. A General Approach for Securely Updating XML Data. In: *WebDB 2012*.

• Some slides of this course are inspired from lectures taught by Pr Wenfei Fan.